Hierarchical Distributed Repositories in Concurrent Versions System

by

Miroslav Jurišić

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Miroslav Jurišić, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.
Hierarchical Distributed Repositories in Concurrent Versions System

by

Miroslav Jurišić

Submitted to the Department of Electrical Engineering and Computer Science on Jan 19, 2001, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a hierarchical distributed repository system for Concurrent Versions System (CVS). The hierarchical repository system is based on the existing CVS implementation. I modified the repository structure, client-server protocol, working files structure, and user interface to provide the facilities necessary to maintain information about a distributed repository.

Thesis Supervisor: James D. Bruce –Title: Vice President, Information Systems
Acknowledgments

This thesis is my last major work as an MIT student (at least for the foreseeable future), and as such it would not have been possible without those who helped me become an MIT student, and those who helped me as an MIT student. In addition to that, the thesis would have taken an entirely different shape without the influence of the people who helped me bring it to completion. To all of those people, I owe many thanks, even if I can’t list them all. However, some of them deserve a special mention.

My mother’s support for my interests was instrumental in bringing me to MIT. Without all the toys and books she gave me, I would not have discovered my most rewarding interests until too late.

I owe the most crucial seeds of my knowledge and enthusiasm to the people who nourished my interest in math and computer science, and poured their energy into my education: Ivo Melvan, Mea Bombarelli, Snježana Serdar, Ilko Brnetić.

Daria Rovan imbued me with a better understanding of who I am and who I want to become, and for that I will be forever grateful.

Dean Bonnie Walters provided more help in overcoming my culture shock than I ever imagined would be necessary. Her words and support were the root of my strength and enthusiasm, and enabled me to stand up to the challenges I faced at MIT.

Marshall Vale, Scott McGuire, and Alexandra Ellwood provided me with an environment in which I could express my tendency to try to do too much all at once, and I thank them for all the effort they put into providing me with hard problems.

Jim Bruce stepped in at the moment of greatest uncertainty in my life and gave me a wonderful opportunity to do exactly what I wanted.

Greg Hudson reviewed the proposals, draft, and the thesis itself, improving them in countless ways.

Without Jelena’s precious companionship and welcome distractions, many things I had dreamed of would have remained just that.
Kad već postoji planina 

Kad već postoji planina, treba se penjati
Strpljivo i dugo do samoga vrha.
Kad već postoji planina, treba je upoznati
Srecem i umom kao materinsku riječ.
Kad već postoji planina, treba je osvojiti
Polako i mudro kao jedinu ljubav.
Kad već postoji planina, treba je zavoljeti
Predano i nježno kao dijete.
Zato i postoji planina:
Da bismo otkrili
Pute neprohodne.

Ljerka Car Matutinović
Contents

1 Introduction .................................................. 13
1.1 Motivations for hierarchical repositories .................. 13

2 Existing CVS repository model ................................ 17
2.1 Repository structure ....................................... 17
2.2 Client-server protocol ..................................... 19
2.3 Working files .............................................. 19
2.4 User interface ............................................. 20

3 Modifications to the repository model ....................... 21
3.1 Changes to repository structure .......................... 22
3.2 Changes to client-server protocol ........................ 23
3.3 Changes to working files .................................. 25
3.4 Changes to user interface .................................. 25

4 Implementation ................................................ 27
4.1 Repository structure ....................................... 27
4.2 User interface ............................................. 28
4.3 Client-server protocol ..................................... 28
4.3.1 Authentication exchange ............................... 28
4.3.2 Server responses ...................................... 29
4.3.3 Client requests ........................................ 32
5 Discussion

5.1 Implementation analysis ................................................. 37
5.2 Further work ..................................................................... 39

6 Conclusion

A Source code

A.1 add.c .............................................................................. 45
A.2 admin.c ......................................................................... 55
A.3 buffer.c .......................................................................... 65
A.4 buffer.h .......................................................................... 85
A.5 checkin.c ......................................................................... 87
A.6 checkout.c ....................................................................... 90
A.7 classify.c ......................................................................... 104
A.8 client.c ........................................................................... 110
A.9 client.h ........................................................................... 180
A.10 commit.c ......................................................................... 183
A.11 create_adm.c ................................................................. 210
A.12 cvs.h ........................................................................... 213
A.13 cvsrc.c .......................................................................... 223
A.14 diff.c ............................................................................ 225
A.15 edit.c ............................................................................ 237
A.16 edit.h ............................................................................ 250
A.17 entries.c ......................................................................... 251
A.18 error.c ........................................................................... 265
A.19 error.h ........................................................................... 268
A.20 expand_path.c ............................................................... 269
A.21 fileattr.c ......................................................................... 273
A.22 fileattr.h ......................................................................... 281
A.23 filesubr.c ................................................................. 283
A.24 find_names.c ............................................................... 294
A.25 hardlink.c .................................. 299
A.26 hardlink.h .................................. 303
A.27 hash.c ................................... 304
A.28 hash.h ................................... 310
A.29 history.c .................................. 311
A.30 ignore.c .................................. 329
A.31 import.c .................................. 335
A.32 lock.c .................................... 353
A.33 log.c .................................... 362
A.34 login.c ................................... 378
A.35 logmsg.c .................................. 384
A.36 main.c ................................... 394
A.37 mkmodules.c ............................... 406
A.38 modules.c ................................. 416
A.39 myndbm.c ................................. 428
A.40 myndbm.h ................................. 432
A.41 no_diff.c .................................. 433
A.42 parseinfo.c ................................. 435
A.43 patch.c ................................... 440
A.44 rcs.c .................................... 449
A.45 rcs.h .................................... 549
A.46 rcscmds.c .................................. 552
A.47 recurse.c .................................. 560
A.48 release.c .................................. 571
A.49 remove.c .................................. 575
A.50 repos.c ................................... 579
A.51 root.c .................................... 582
A.52 rtag.c .................................... 589
A.53 run.c .................................... 598
A.54 scramble.c .................................. 604
<table>
<thead>
<tr>
<th>A.55 server.c</th>
<th>A.56 server.h</th>
<th>A.57 status.c</th>
<th>A.58 subr.c</th>
<th>A.59 tag.c</th>
<th>A.60 update.c</th>
<th>A.61 update.h</th>
<th>A.62 vers.ts.c</th>
<th>A.63 vers.ts.c</th>
<th>A.64 version.c</th>
<th>A.65 watch.c</th>
<th>A.66 watch.h</th>
<th>A.67 wrapper.c</th>
<th>A.68 zlib.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . .</td>
<td>. . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . .</td>
<td>. . . . . . .</td>
<td>. . . . . .</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

CVS (Concurrent Version System) is a version control system, initially released in 1990 and based on RCS (Revision Control System) from 1982. Building on the basic version control primitives provided by RCS, CVS added many higher-level functions which simplified management of large projects, focusing on manipulating groups of files rather than individual files. CVS provides developers with the ability to track changes to a project over time, to revert parts of the project to their state at an arbitrary point in time, to work on the project concurrently with other developers, and to work on different variants of the project simultaneously. CVS operations provide relatively simple ways of performing most common manipulations on a large project, while still having the flexibility that allows fairly complex version control tasks.

CVS initially did not support client-server operation, but the support was added early on, making CVS very popular in academic environments, as well as many open-source projects. Its popularity increased even more with support for authentication and encryption, and ports to many UNIX platforms, Mac OS and Windows.

1.1 Motivations for hierarchical repositories

Unfortunately, several major features of CVS came about as an afterthought or a side-effect of the original implementation rather than through careful design. As a
result, CVS sometimes falls short of what developers expect from a version control system. One particular area in which CVS is limited is that there is no interaction between different source code repositories. Although there are relatively primitive facilities for manually propagating information between different repositories, their use is generally tedious and error-prone.

The goal of this thesis is to extend CVS and provide the ability to easily propagate information between repositories, possibly stored on different servers, and to establish interconnections between different repositories. These extensions represent the ideas that different groups of people might work on one project, without necessarily sharing the CVS server infrastructure, and that a single project might actively incorporate parts of other projects, without having to share the CVS infrastructure with them. The following situations, for example, might be represented by this model:

- Developer group $G_1$ is working on project $P$. Developer group $G_2$ wants to make private modifications to $P$ for its own purpose, and continue tracking changes $G_1$ makes to $P$.

- Developer group $G_1$ is working on project $P$. Developer group $G_2$ wants to make modifications to $P$, and propagate those modifications back to $G_1$, but it doesn’t want to use the same CVS server infrastructure as $G_1$.

- Developer group $G$ is working on project $P$. It wants to maintain an internal CVS server and a public CVS server; the private server is used for development, and the public CVS server is used to provide the user and the developer community with read-only access to selected versions of the source tree.

- Developer group $G$ is working on project $P$. It wants to maintain a repository with every revision of every source file in a project. It also wants to maintain a separate repository containing only selected versions of source files in the project (for example, only publicly released versions). The second repository would be considerably smaller and could be used for various archival purposes.
• A developer going off-line for an extended period of time could create a local repository on his private computer, and propagate his changes back to the project repository after returning on-line.
Chapter 2

Existing CVS repository model

In order to understand the changes necessary to the existing CVS architecture, some of its key properties need to be specified.

CVS client-server architecture can be thought of as consisting of four parts:

- the repository, stored on a server;
- the client-server protocol, used to communicate between a client and a server;
- the working files, stored on a client; and
- the user interface, used to communicate between a client and a user, or a server and a user (for administration).

2.1 Repository structure

A repository is a hierarchy of files residing on the server. Every file has a history, which encompasses all revisions of that file since its creation.

The history of each file is also hierarchical. The revision tree of a file consists of branches, every branch containing an arbitrary number of revisions. Conceptually, a branch corresponds to a line of development in the life of a project; for example, concurrent development of two separate versions might use two branches to maintain separate histories of the two versions.
Every revision in the tree is uniquely identified with a *revision number*, which is an ordered n-tuple of integers. The last component of a revision number identifies the revision on a branch, and the remaining components identify the branch. Revision numbers are usually written as dot-separated strings. For example, the n-tuple \((1, 2, 3, 4)\) would be written as \(1.2.3.4\), and would represent revision 4 on branch 1.2.3.

Revisions are numbered sequentially, so revision \(x.2\) would immediately precede revision \(x.3\) and immediately follow revision \(x.1\). The first revision of a file has the number \(1.1\). The initial branch of a file (with branch number 1) is called *the trunk*.

Every branch has a *head revision*, which is the latest revision on the branch. Every branch except for the trunk has a *branch point*, which is the revision from which it was derived. The branch point of a branch is a revision on a different branch – its *parent branch*. The branch number of every branch consists of the branch point revision number, followed by its *branch index*, which is an even integer \(^1\). Branch indices are assigned sequentially, starting at 2. For example, the branch with number \(1.2.4.6.8\) would be the fourth branch whose branch point is at the revision \(1.2.4.6\) (which in turn would be the sixth revision on the second branch branched from revision \(1.2\)).

Every revision and branch can carry any number of *symbolic tags*, which are character strings. \(^2\) No two revisions or branches of a single file can carry the same symbolic tag. Symbolic tags are used to label specific revisions to identify them by their function in the project development. For example, “Final:3.0.version” might be a tag assigned to the revisions of all the files in a repository which were part of the final 3.0 version of the product. This allows them to be easily identified later, since individual revision numbers of different files are likely to be significantly different.

---

\(^1\) The only exceptions are the vendor branch, used when importing vendor sources into a repository, whose branch number is 1.1.1, and the trunk, whose branch number is 1.

\(^2\) Only alphanumeric characters, underscores, and hyphens are allowed in symbolic tags.
2.2 Client-server protocol

The first step in the client-server protocol is authentication. CVS supports three different authentication mechanisms: password, Kerberos, and GSSAPI. The server distinguishes between them based on which network port received the authentication request.

After authentication completes successfully, the client sends commands to the server. Every command consists of a command name (such as "checkout"), options (whose allowed values depend on the command), and arguments (usually names of the files the command is to operate on). The server sends a response for every command it receives. Some commands require the client and the server to exchange contents of files. If a file is sent from the client to the server, it is sent before any commands that affect it, and then referred to in the commands. If a file is sent from the server to the client, it is sent as a part of a response.

The client-server protocol shows the roots of CVS. The initial version of CVS required the repository and the working files to be available to the CVS program via the file system. With the addition of the client-server model, the client sends the server enough information to recreate a portion of the client’s file system in a temporary directory on the server. The server then executes the specified commands as if it were not running in the client-server mode, and returns the responses to the server.

2.3 Working files

Working files reside in the client’s file system. For each file present among the working files (not all files from the repository have to be present), the client stores the contents of one revision of the file, possibly with local modifications.

For each directory containing working files, the CVS client maintains information needed to communicate with the server in a subdirectory named "CVS", also known
as the *administrative directory*. The contents of an administrative directory, known as the *administrative files*, contain

- the location of the repository from which the directory was retrieved;
- the location of the directory within the repository; and
- a list of files in the directory, together with per-file bookkeeping information such as the revision number of the revision which was last retrieved from the repository.

### 2.4 User interface

The user interface, in both the command-line versions of the CVS client (UNIX, Windows), and graphical versions of client (Macintosh, Windows), allows the user to specify a number of files in the working directory and a command to perform on them. Where applicable, there might be optional parameters to refine the command.

The client then translates the command to a request the server understands. Since the server essentially executes the commands as if they had been issued locally on the server, most commands require very little translation.
Chapter 3

Modifications to the repository model

The fundamental property of a distributed repository is that it is possible for different parts of the history of a file to be on different servers; at the same time, the client still has to be able to present them as a single entity to the user.

Starting from the existing repository model, the main new concept necessary to support a distributed repository is that a server might not be able to fulfill client’s request, because it needs revision information carried by a different server. However, if the repository is allowed to be distributed arbitrarily, then complexity of any operation on the repository grows beyond practical limits.

To reduce complexity, we need to take advantage of the fact that the full generality of a distributed system is not needed to support objectives outlined in the first chapter. In addition to that, we note the following fact about the way people typically use CVS:

Actions which operate on two or more revisions of a file which are on different branches are rare. The majority of operations only use one or more revisions on one branch.

We can use this observation to form the basic premise of a distributed CVS system which fulfills the described without the maximum possible generality:

All revisions on a particular branch of the revision tree of a file must reside on a single server.
With that restriction, a server will have all the necessary information for most CVS operations. Therefore, most operations in a distributed repository will behave exactly the same as they would in a non-distributed repository.

This restriction simplifies the problem considerably; it becomes unnecessary to consider arbitrary arrangements of servers and arbitrary distributions of revisions among servers. With this restriction, the performance of the client and server are essentially unaffected for vast majority of CVS operations, simply because the majority of CVS operations take place on a single branch. Also, the amount of additional state stored on the server is reduced, because only branches incur additional state, whereas there is no additional state needed for each revision.

Since revisions and branches are arranged in a tree, and a transition from one server to another can only occur at a branch point, servers are effectively also arranged in a tree. Because of that, this is a hierarchically distributed repository.

To implement hierarchical repositories, modifications are necessary to each of the four subdivisions of CVS presented in the previous chapter. Overall, the changes introduce the idea that a server might not know a particular revision's contents or even whether it exists, but instead knows which other server might have that information.

### 3.1 Changes to repository structure

The existing repository structure already encompasses information about branches and branch points. In order to support the notion of a remote branch (i.e. a branch which exists on a different server), the repository structure has to be extended with two pieces of information:

- Each revision has to include locations of all branches at the revision which are not carried by the server
- A branch whose parent branch is carried by a different server has to include the location of the parent branch.

The RCS file format, used by the server for storage of all revision information, is very flexible. It is essentially a tagged data format; it consists of a list of records
corresponding to individual revisions, each record listing a number of fields in the form of key-value pairs. The fields describe properties of each revision; therefore, it is not difficult to add additional fields to list the locations and the branch numbers of remote branches at a branchpoint, and the location of the parent branch.

For each remote branch, in addition to its branch number, the RCS file has to include the location where the branch can be found. This location is a repository root, as already used in CVS, without authentication information (since it is the client, not the server, that determines the authentication protocol to be used).

### 3.2 Changes to client-server protocol

Some commands issued by the CVS client don’t operate on any files. Those commands require no modification, as it is only per-file information which is different in the hierarchical repository model.

The remaining commands operate on one or more files; whenever they operate on more than one file, the operations on different files are independent of each other. Therefore, the commands could just as well be operating on single files, and the ability to operate on multiple files is included mainly for user convenience. Because of that, we can consider the commands as if they operate on single files.

Of the commands that operate on files, some operate on a single revision of a file, some operate on two revisions of a file, and some operate on three revisions of a file. Although some commands’ parameters determine whether the command operates on one, two, or three revisions of a file, those cases can be considered as separate commands, since the combination of command and parameters always uniquely determines how many revisions the command operates on.

Every command operating on a single revision must be modified to accept a new response, by which the server indicates to the client that the requested revision is not carried by that server, and refers the client to a different server. The client can then retry the operation on the other server (which might again redirect the client).
Commands operating on two or three revisions have the potential difficulty that some of those revisions might be carried by different servers. In that case, the client must be able to locate all but one of the revisions on different servers (using the same procedure as it would for single-revision commands), and then pass that information to the server carrying the last one. The server then has the information about all of the revisions, and can complete the command. Putting the burden of locating all the necessary information on the client reduces the impact of distributed model on server performance.

The tag command, used to create branches (and other symbolic tags), has to be modified to allow creation of remote branches. Remote branch creation has to be performed in two steps: first the server carrying the parent branch is contacted to determine the branch number of the new branch, then the server which will carry the new branch is contacted to create the branch. The parent server has to return information to the client about the new branch, which the client then transmits to the child server. This requires modifying the protocol to allow the tag command to return additional information, and to allow the remote branch to be created with this additional information.

Authentication in the existing CVS protocol is inadequate for distributed repositories, because there are no provisions for authentication negotiation between the client and the server. Since it is possible that a client might need to contact several servers to complete an operation, the lack of authentication negotiation means that a client would have to store authentication information for every server. Instead of imposing that restriction, the authentication exchange will be modified to include negotiation of the authentication mechanism; then the client can hop from one server to another without requiring each one of them to be configured on the client. This will work especially well with Kerberos or GSSAPI authentication, as the client can present user’s credentials without prompting the user for a different password for every server.
3.3 Changes to working files

The existing structure of working files stores information per-directory. All the files in a single directory are assumed to come from the same server. However, in a distributed repository, files within one directory might come from different servers. Therefore, this restriction has to be relaxed to allow additional per-file administrative information.

3.4 Changes to user interface

Since remote repositories correspond to branches in the revision tree, the existing facilities for creating a branch can be extended to allow creating remote branches. The tag command creates a new branch of a file, branching off the revision of the file currently in the working files.

To allow creating remote branches, the tag command can be extended to allow an argument specifying the repository on which the branch should be created, as follows:

Create a standard branch: cvs tag -r BranchName

Create a remote branch: cvs tag -r :Server:Repository:BranchName

This syntax also follows the established CVS syntax for specifying a repository location by separating the components with a ‘:’. The branch is created on the specified repository, which must be created separately, using cvs init. Repository creation and branch creation are separate because creating a repository is usually a task which only a small group of trusted people is allowed to perform, and therefore allowing a repository to be implicitly created with the tag command would be inappropriate.
Chapter 4

Implementation

4.1 Repository structure

Every file in the repository consists of a header, which contains information common to all revisions of the file, and a list of revisions.

The header contains the version number of the head revision (the most recent revision on the main branch) and a list of all symbolic tags.

Every revision contains a list of branches at that revision and the revision which immediately precedes it on the branch (or none, if that is the first revision on the branch).

Every field in the revision file has the form \texttt{<key> [<value>] [<value>] [...];}.

The values are optional; the values are separated from the key and from each other with whitespace.

The list of remote branches at a revision is stored in a new \texttt{remote-branches} field. The format of each value in the list of remote branches is \texttt{<revision><location>}, where the revision is the branch number of the remote branch, and the location is its location. The location is given as a CVS root – \texttt{:method:server:repository}. The method field is unused in the current implementation, but is included to avoid introducing a new way to specify CVS repositories.

The location of the parent branch is stored in a new \texttt{parent-branch} field in the RCS file header. This field is also given as a CVS root.
The RCS subsystem of CVS has to be modified to parse the new fields of in the RCS files. The list of branches at a revision was added to the `rcsversnode` structure, where as the parent branch location was added to the `rcsnodes` structure. The `RCS_parse()` function was modified to parse the additional fields and insert them into the structures as appropriate.

### 4.2 User interface

The only change in the user interface is the ability to specify remote branches for the `tag` command. The only change to the corresponding code was to allow the `tag()` function to recognize remote tags as valid.

However, there are several related changes to the client-server protocol, which are described in the next section.

### 4.3 Client-server protocol

#### 4.3.1 Authentication exchange

As metioned previously, the existing authentication exchange is inadequate, because it requires the client to know the authentication method to be used for a particular server before establishing connection with that server (different authentication methods are served on different ports).

In the existing CVS protocol, after connection is established (and authentication method is therefore known), the client authentication request consist of "BEGIN AUTHENTICATION REQUEST" followed by the username, the authentication data, and "END AUTHENTICATION REQUEST". This concludes the authentication request; the server responds with "I LOVE YOU" or "I HATE YOU".

In the modified authentication exchange, the server begins by transmitting a space-separated list of allowed authentication methods. For example, the server might transmit "KERBEROS V4 GSSAPI PASSWORD".
The client then picks one of the authentication methods, and transmits "BEGIN authentication-method AUTHENTICATION REQUEST".

Following that, the client sends the username on a line by itself, followed by the repository root on a line by itself.

Sending the repository root during the authentication exchange allows the server to perform per-repository authentication upfront, rather than allowing the client to access the server at first, but then denying access later (when the Root command is transmitted).

After the repository root, the client sends authentication data, which varies depending on the authentication method. For password authentication, it's the encrypted password; for Kerberos, it's the authenticator.

Finally, the client sends "END AUTHENTICATION REQUEST", thus concluding the authentication request. The server responds with "I LOVE YOU" or "I HATE YOU", as before.

### 4.3.2 Server responses

#### Not-carried

One new server response is required in the protocol to support distributed repositories. Any command which cannot be completed because its completion requires revisions which are not present on the server generates the Not-Carried response.

The Not-Carried response has the following form:

```plaintext
Not-Carried
revision path –server –repository –directory
revision path –server –repository –directory ...
End-Not-Carried
```

The response consists of any number of four-line blocks specifying the file and revision which are not carried by the server, and where the client should proceed to look for them. The blocks are followed by End-Not-Carried to indicate the end of the response.

Most commands only generate a Not-Carried response for only one revision, since most commands operate on a single revision of a file. However, some commands can
return Not-Carried responses with more than one revision – for example, the update command.

This form of the response allows a client to associate all parts of the response with the command it had issued, but also allows the client to treat multiple revisions in the response as separate entities. The client’s implementation might make one interpretation or the other more appropriate. For example, a client implementation which maintains association between requests and responses (common in clients with graphical user interface) would want to know whether two Not-Carried responses are generated in response to the same request or not. On the other hand, a client implementation which handles all responses after sending all requests (common in command-line clients) might not care whether two Not-Carried responses are related or not.

For each revision specified in the response, the following information is included:

- revision: the revision number (or tag) to which this information pertains. This allows the client to determine what to ask of the other server.
- path: the client’s path to the file to which this information pertains. This is the same as the path which the client passed to the server, and is included so that clients do not have to carry such context information from the time that the request is made to the time when the response is received. ¹
- server: the host name of the other carrying the specified revision. The name should be a fully qualified name.
- repository: the location of the repository on the other server.
- directory: the location of the file in the repository on the other server; in general, this is a subdirectory of the repository path.

With that information, the client can proceed to the other server, which might be able to complete the request, or might issue more Not-Carried responses, redirecting

¹ One might argue that a client should carry such information in order to provide better error reporting and user interaction, but the most common CVS client does not do that, and it is by far simpler to include this additional piece of information in the response than it would be to rearchitecture the client.
the client again. In either case, The Not-Carried responses provide the client with enough information to find the needed revisions and complete the desired operation.

The CVS server implementation uses the Classify_File() function to determine the state of a file in the repository. Therefore, introducing the "Not carried" status involved modifying Classify_File() to add the logic to detect when a revision might exist on a different server.

Classify_File() in turn uses Version_Ts() to retrieve RCS information about a file, so Version_Ts() also had to be modified to handle remote revisions.

Version_Ts() internally used RCS_getversion() to establish which RCS version corresponds to a CVS revision or symbolic tag. Since many other places in CVS use RCS_getversion(), it was not modified. A new function, RCS_getremoteversion() was added instead; it performs the same function as RCS_getversion(), but only handles remote revisions.

Once Classify_File() was modified to return the "not carried" status, the places which call Classify_File() had to be modified to handle the new status.

The client handles Not-carried responses in the handle_not_carried() function, which is added to the client's response handler table. Contents of the Not-carried response are added to a queue by handle_not_carried(). After all the responses from the server are received, processing turns to the queue and handles all outstanding Not-carried responses.

In most cases, handling a queued Not-carried response involves repeating the same command on a different server. The only exception are commands which operate on more than one revision of a file: difference and update. For those commands, handling a queued Not-carried response involves retrieving the remote revisions from a different server and then returning to the first server to complete the command.

Queue processing is handled by the client_process_remotes() function, which traverses the queue and performs the appropriate action on each entry.

If the action is to repeat the command on a different server, a new argument list is assembled by the setup_args() function, and a new CVS root is configured by the
setup_root() function, after which the CVS command parsing is invoked to execute
the command as if it had been invoked directly by the user.

If the action is to fetch a revision from a remote server, a new argument list and a
new root are built, but rather than invoking the original command again, the update
command processor is invoked, with arguments specifying that the revision should be
directed to the standard output stream.

After the remote revisions are retrieved, queue processing returns to processing
the original command, by reissuing the command to the first server, but also including
the contents of the remote revisions, as described in the next section.

Create-remote-branch

When a server is asked to create a remote branch in the revision tree (using the
modified tag request described below), it examines the existing revision tree and
determines the branch number of the new branch. It reserves this branch number in
the revision tree of the file, and then returns the revision number of the new branch
in a Create-remote-branch response, which takes the following form:

    Create-remote-branch filename -server -repository -path -revision

The response includes the name of the file, the full location of the new remote
branch (the name of the server, the location of the repository, and the location of the
file in the repository), followed by the branch number of the new branch.

The client handles this new response by enqueuing a new request to be sent to
the server where the remote branch will reside. This request will create the remote
branch itself, as described below.

4.3.3 Client requests

The syntaxes of most existing client requests are unchanged in the distributed reposi-
tory model. However, when different revisions needed to complete a request reside on
different servers, the client has to have a mechanism of transferring this information
from one server to another. In addition to that, the process of remote branch creation requires additional information in some existing requests.

**Remote-revision**

Since CVS already has a mechanism for retrieving arbitrary revisions of a file, it is easy for the client to retrieve a revision from one server in order to transfer it to another server. The existing *update* request is used for that, with the only difference being that the file returned by the server is saved in the administrative *CVS* directory, rather than in the working directory.

However, in order to send the contents of a revision to the server, without having the server change the contents of the repository, the client needs a new request, **Remote-revision**. With this request, the client sends the contents of a remote revision to the server, and the server retains that information until the end of the session. During the session, the server can use the information to complete requests which might otherwise elicit a *Not-carried* response.

In fact, the format of the **Remote-revision** request does not need to differ significantly from the existing *Modified* request, which sends the contents of a client’s working file:

```
Remote-revision filename -revision -mode -size -contents
```

The request includes the name of the file provided in the request, followed by the revision number, the file mode (as defined by the file system), the size (in bytes), and by the contents of the revision. Note that no RCS keyword substitution is performed on the contents of the file.

The CVS client implementation sends information about every working file affected by a command by recursively traversing the working files and directories, and sending the status and, if necessary, the contents of the files. This is done from the *send_fileproc()* function, which determines the status of a file, and sends appropriate information to the server.
Remote-revision is added sent to the server by `send_fileproc()`, along with the other information about working files. `send_fileproc` calls a new function, `send_remoterev()`, which sends the contents of a remote revision, as received while handling a Not-carried response.

On the server, the Remote-revision request is handled by saving the contents of the remote revision in a temporary file in the new `serve_remote_revision` function. Additionally, the logic for determining whether a client command will result in a Not-carried response has to be extended to avoid returning that response when the remote revision has already been provided by the client. This is accomplished by making further modifications to `Classify_File()`.

In order to allow the RCS subsystem to be able to perform operations on the remote revisions, the remote revisions have to be temporarily injected into the RCS revision tree of the file. When `Classify_File()` is asked to classify a remote revision which has been provided by the client in a Remote-revision request, it inserts the contents of the revision into the revision tree on the special branch 1.1.3, and classifies the revision as local. At the same time, it changes the real remote revision number to the revision on the 1.1.3 branch. Therefore, any calls to `Classify_File` are automatically redirected to use the local data for remote revisions.

tag

The `tag` request is used to create symbolic revisions and branches. It directly corresponds to the `tag` command in the user interface. Therefore, augmenting the `tag` command to accept the new syntax for remote branch creation directly affects the `tag` request, which now also has to handle remote branch creation.

The branch name argument, which in the `tag` command is now allowed to be the name of a remote branch, is passed directly to the server in a `tag` request. Other than allowing the new (previously invalid) values for this argument, the request is unchanged.

Server processing of this request is different. In addition to finding a new branch number and creating the branch, the server has to return a Create-remote-branch
response to the client, instructing the client to contact the server on which the branch will be carried. To accomplish this, first the function \texttt{tag\_fileproc()} had to be modified to recognize remote branch names. When a remote branch name is detected by \texttt{tag\_fileproc()}, it calls a new function, \texttt{RCS\_setremotetag()}, which functions similarly to \texttt{RCS\_settag()}: it finds a free branch number at the desired revision, and reserves it in the revision tree, updating the file in the repository.

After \texttt{RCS\_setremotetag()} returns the new remote branch revision, the information is returned to the client by \texttt{tag\_fileproc()} in a \texttt{Create-remote-branch} response, described in the previous section.

\texttt{add}

When the client receives the \texttt{Create-remote-branch} response, it handles it by establishing a connection with the server which will carry the remote branch. This server has to create a new file in the repository, setting its parent branch to the one specified by the client.

To transmit this information from the server to the client, a new argument was added to the \texttt{add} request. In non-distributed repositories, \texttt{add} tells the server a file is about to be added to the repository. This request is augmented with a new argument, which specifies the location of the parent branch. The new argument is of the form

\begin{verbatim}
-b :server :repository :revision
\end{verbatim}

The server and the repository strings specify the location of the parent branch, whereas the revision is the revision number of the new branch.

In response to this request, the server creates an empty revision file in the repository, only setting its parent branch from the information in the request. New revisions committed to this file are committed to the specified branch.

The function \texttt{add\_rcs\_file()} is used to create the new empty revision file. It was already used by \texttt{commit()} and \texttt{import()} to create new files in the repository. It had to be extended to take a new argument specifying the remote branch name. It is then called by \texttt{add()} in response to a request to add a new remote branch.
Chapter 5

Discussion

5.1 Implementation analysis

At every step from the initial design throughout the implementation, the project was constrained by being implemented as an extension to CVS rather than a replacement for CVS.

A revision control system typically maintains many man-years of project history, and therefore development teams are understandably reluctant to make any modifications to their existing version control system, especially if those changes break backwards compatibility with the existing repositories or existing clients. Other things being equal, acceptance of a revision control system is chiefly determined by how difficult it is to switch to it from existing version control systems.

In addition to that, the focus of this thesis is on distributed repositories, and therefore I considered it undesirable to write a complete revision control system merely to serve as the basis for my extensions.

Hence, the intent was to demonstrate that the problems described in Chapter 1 can be solved by extending an existing widely used revision control system. On one hand, this was a tremendous benefit, because many basic problems of revision control had already been solved. File merging, histories, branches, tags, and many other things were already in place.
On the other hand, during the implementation of my thesis, I discovered that the implementation I used as a starting point was not at all designed with future extensions in mind. This turned out to be a major hindrance in my implementation, and is the main reason why some important work had to be left out of my implementation.

The main difficulty with the implementation I used is that it had been worked on by several different people and teams at different times, but there was very little communication among them.

The resulting design is somewhat incoherent and not easily extended. In its current state it seems to be factored and separated into manageable parts, but on a lower level each of those parts is a combination of several incompatible intents, all pulling in different directions. The structure of the code has reached the point where changing one thing can cause seemingly unrelated others to fail. This was a major problem to me, and unfortunately it did not become obvious until it was too late to consider different starting points or even writing everything from scratch.

Regrettably, but inevitably, the changes I made only increased the strain in the code.

My final implementation works sufficiently to demonstrate that the idea of hierarchical repositories can solve the problems which it was intended to solve, and can be implemented as an extension to CVS. However, the implementation is far from robust and not something I would recommend for a production environment. Many of its aspects are untested or unpolished.

However, I do believe that hierarchically distributed repositories are feasible and useful.

Recently, the computing industry has seen a tremendous increase in visibility of open-source software, and in interaction between commercial software and open-source software. Any project faces the possibility of being partly proprietary and partly open-source, either as an open-source project being extended by a commercial vendor, or as a commercial product becoming open-source. This implies that different parts of one project might be managed by different people, teams, or even companies,
and it is therefore paramount that the version control system gives them the ability to decentralize the project to parallel the decentralization of control.

Thus, my recommendation to those interested in distributed version control is:

- If you are designing a new version control system, consider distributed repositories in your design, even if you do not intend to implement them initially. Allowing for that possibility early on will likely greatly reduce the effort needed to implement them when the market demands it.

- If you are extending a new version control system to use distributed repositories, consider whether the implementation you start from is extensible; unfortunately, this is not necessarily easy to determine without trying to extend it.

- If you are extending CVS to use distributed repositories, consider rewriting the rest of CVS before you start, or using a different starting implementation from the one I used. As of this writing, there aren’t any other implementations of both the client and the server used, but there is at least one free, independent, and much better thought out implementation [4] of the client which, in hindsight, might have been a more fruitful starting point.

5.2 Further work

The update command has not been extended to allow merging of revisions from different servers. However, most of the functionality necessary to support update was also necessary to support diff and therefore only a comparably small amount of additional work would be required to fully support update.

The minor problems of which I am aware, but which I have not had the time to resolve are:

- Modify the client to coalesce requests for multiple remote revisions from one server into a single request. For example, if three remote revisions are require to complete a command, and two of them reside on the same server, them both can be retrieved during a single connection to the server, rather than being retrieved with two separate connections to the server. My implementation keeps the out-
standing requests for remote revisions in a double-ended queue, and processes the queue in order. Coallescing the requests could be done by inserting additional outstanding remote revision requests immediately before or after existing requests, rather than at the beginning or the end of the queue.

- Eliminate remaining resource leaks from the client. There are several resource leaks in the client, primarily in constructing argument lists for commands executed in response to `Not-carried` server responses. These leaks can become significant if there are many servers in a distributed repository, because the expected number of Not-Carried responses grows with the size of the server hierarchy.

- Improve error handling in handling of remote revisions. Like the rest of CVS, the code which handles remote revisions aborts client execution when encountering a fatal error. However, some errors which are currently considered fatal don’t need to be fatal if appropriate status information can be returned to the higher-level code, and handled there. Unfortunately, error reporting capabilities in the underlying implementation are somewhat primitive, and therefore improving error handling might require modifications to the error-handling code.

- Add client-side caching of the repository hierarchy. In handling a request for a remote revision, much time is spent determining where the revision is, compared to the amount of time spent handling the request once the revision is found. The time spent discovering the repository hierarchy grows with the size of the hierarchy, and therefore reducing that time can significantly improve client performance in large hierarchies. Since it is impossible for more than one server to carry a particular revision, it is easy to verify whether cache entries are valid or stale simply by attempting to retrieve the revision from the server.

- Improve performance of the authentication exchange. Currently, there is a noticeable delay during the authentication exchange, which hasn’t been fully diagnosed. It seems as if the client and the server spend a brief period of time waiting on each other, when they could just proceed with the rest of the exchange. The delay becomes significant in larger server hierarchies because of the need to authenticate to multiple hosts in order to complete the command.
• Submit the changes back to the CVS maintainers. In the spirit of open source software, the modifications made in this thesis will be submitted back to the maintainers so they can be considered for inclusion in the official release of CVS.
Chapter 6

Conclusion

Distributed repositories can be applied to several important scenarios which occur in software development, especially with the recent surge in interaction between commercial and open-source software. They allow the revision control of a project to be distributed in way which parallels the distribution of project management.

The CVS protocol is suitable for an extension which implements distributed repositories with minimal changes to the protocol; unfortunately, because of inadequacies in the authentication exchange of the CVS protocol, it is infeasible to implement hierarchical repositories in a completely backwards-compatible way. A client which does not understand distributed repositories cannot communicate with a server which only understands distributed repositories; however, it is possible to create a server which understands both (and therefore give client with no knowledge of distributed repositories access to individual servers in a distributed repository) simply by using a different network port for clients which are aware of distributed repositories.

The existing CVS implementation by Cyclic, version 1.10 or older, is unsuitable for significant extensions, and should be avoided as a starting point. Other implementations, such as the one which forms the core of MacCVS Pro, a Mac OS CVS client, might be better suited for such extensions.

The ideas and intents of distributed repositories apply to other version control systems, and I urge the designers of new version control systems (of which there are
several as of this writing, partly due to the state of CVS), to design their systems with distributed repositories in mind.
Appendix A

Source code

This chapter contains the entire source code for the CVS client and server with support for hierarchically distributed repositories. The differences between the version of CVS used as starting point (the Cyclic version 1.10) and the final version with support for distributed repositories are marked with change bars.

A.1 add.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * *
 * Add
 *
 * Adds a file or directory to the RCS source repository. For a file,
 * the entry is marked as "needing to be added" in the user's own CVS
 * directory, and really added to the repository when it is committed.
 * For a directory, it is added at the appropriate place in the source
 * repository and a CVS directory is generated within the directory.
 *
 * The -m option is currently the only supported option. Some may wish to
 * supply standard "rcs" options here, but I've found that this causes more
 * trouble than anything else.
 *
 * The user files or directories must already exist. For a directory, it must
 * not already have a CVS file in it.
 *
 * An "add" on a file that has been "remove"d but not committed will cause the
 * file to be resurrected.
 */

#include "cvs.h"
#include "savecwd.h"
#include "fileattr.h"

static int add_directory PROTO ((struct file_info *finfo));
static int build_entry PROTO ((char *repository, char *user, char *options,
                                char *message, List *entries, char *tag));

static const char *const add_usage[] = {
    "Usage: %s %s [-k rcs-kflag] [-m message] files...\n",
    "\t-k\tUse \"rcs-kflag\" to add the file with the specified kflag.\n",
    "\t-m\tUse \"message\" for the creation log.\n",
    "(Specify the --help global option for a list of other help options)\n"};
46 © Source code

NULL

int
add (argc, argv)
int argc;
char *argv;
{
  char *message = NULL;
  char *revision = NULL;
  int i;
  char *repository;
  int c;
  int err = 0;
  int added_files = 0;
  char *options = NULL;
  List *entries;
  VersTS *vers;
  struct saved_cwd cwd;
  
  /* Nonzero if we found a slash, and are thus adding files in a
   * subdirectory. */
  int found_slash = 0;
  if (argc == 1 || argc == -1)
    usage (add_usage);
  wrap_setup ();
  while ((c = getopt (argc, argv, "+k:m:r:")) != -1)
    switch (c)
    {
    case 'k':
      if (options)
        free (options);
      options = RCS_check_kflag (optarg);
      break;
    case 'm':
      message = xstrdup (optarg);
      break;
    case 'r':
      if (adding_remote)
      { revision = xstrdup (optarg);
      } else {
        error (1, 1, "-r is not valid as a user option to add");
      }
      break;
    case '?':
      default:
      usage (add_usage);
      break;
    }
  optind = optarg = 0;
  argc -= optind;
  argv += optind;
  if (argc <= 0)
    usage (add_usage);
  /* First some sanity checks. I know that the CVS case is (sort of)
   * also handled by add_directory, but we need to check here so the
   * client won't get all confused in send_file_names. */
  for (i = 0, i < argc; i++)
    {
      int skip_file = 0;
      /* If it were up to me I'd probably make this a fatal error.
       * But some people are really fond of their "cvs add *", and
       * don't seem to object to the warnings.
       * Whatever. */
      strip_trailing_slashes (argv[i]);
      if (strcmp (argv[i], "*") == 0 ||
          strcmp (argv[i], "..") == 0 ||
          fncmp (argv[i], CVSADM) == 0)
        { error (0, 0, "cannot add special file \"*\"; skipping", argv[i]);
          skip_file = 1;
        } else {
        char *p;
        p = argv[i];
        while (*p != '\0')
          {
            if (ISDIRSEP (*p))
              skip_file = 1;
            p++;
          }
      }
{  
  foundslash = 1;
  break;
}  
++p;
}
}

if (skip_file)
{
  int j;

  /* FIXME: We don't do anything about free'ing argv[i]. But
   * the problem is that it is only sometimes allocated (see
   * cvs.c). */
  for (j = i; j < argc - 1; ++j)
    argv[j] = argv[j + 1];
  --argc;
  /* Check the new argv[i] again. */
  --i;
  ++err;
}
}
#endif CLIENT_SUPPORT
if (client_active)
{
  int i;

  if (argc == 0)
    /* We snipped out all the arguments in the above sanity
       check. We can just forget the whole thing (and we
       better, because if we fired up the server and passed it
       nothing, it would spit back a usage message). */
    return err;

  start_server ();
  ign_setup ();

  if (options) send_arg(options);
  options = arg("-m", message);

  /* If !foundslash, refrain from sending "Directory", for
   * CVS 1.9 compatibility. If we only tried to deal with servers
   * which are at least CVS 1.9.26 or so, we wouldn't have to
   * special-case this. */
  if (foundslash)
    {  
      repository = NameRepository (NULL, NULL);
      send_a_repository ("", repository, "");
      free (repository);
    }

  for (i = 0, i < argc; ++i)
    /* FIXME: Does this erroneously call CreateAdmin in error
       conditions which are only detected once the server gets its
       hands on things? */
    if (isdir (argv[i]))
      {
        char *tag;
        char *date;
        int nonbranch;
        char *rcsdir;
        char *p;
        char *update_dir;
        /* This is some mungeable storage into which we can point
           with p and/or update_dir. */
        char *filendir;

        if (save_cwd (&cwd))
          error_exit (1);

        filendir = xstrdup (argv[i]);
        p = last_component (filendir);
        if (p == filendir)
          {
            update_dir = "";
          }
        else
          {
            p[-1] = '\0';
            update_dir = filendir;
            if (CURLCHDIR (update_dir) < 0)
              error (1, errno,
              "could not chdir to \x22", update_dir);
          }

        /* find the repository associated with our current dir */
        repository = NameRepository (NULL, update_dir);
        }  

/* before we do anything else, see if we have any
per-directory tags */
ParseTag (&tag, &date, &nonbranch);
rcsdir = xmalloc(strlen(repository) + strlen(p) + 5);
sprintf(rcsdir, "%s/%s", repository, p);
CreateAdmin(p, argv[i], rcsdir, tag, date,
nonbranch, 0);

if (found slash)
    send_repository ("", repository, update_dir);
if (restore_cwd (&cwd, NULL))
    error_exit (1);
free_cwd (&cwd);
if (tag)
    free (tag);
if (date)
    free (date);
free (rcsdir);
if (p == filedir)
    Subdir_register ((List *) NULL, (char *) NULL, argv[i]);
else
    {
        Subdir_register ((List *) NULL, update_dir, p);
    }
free (repository);
free (filedir);
}
sendfile

#define SERVER_SUPPORT

int begin_err = err;

beginAdds = added_files;

struct fileinfo finfo;

char *p;
memset(&finfo, 0, sizeof finfo);
if (save_cwd (&cwd))
    error_exit (1);
finfo.fullname = xstrdup(argv[i]);
p = last_component(argv[i]);
if (p == argv[i])
    {
        finfo.update_dir = "*";
        finfo.file = p;
    }
else
    {
        p[-1] = '\0';
        finfo.update_dir = argv[i];
        finfo.file = p;
    }
if (CVS_CHDIR (finfo.update_dir) < 0)
    error (1, errno, "could not chdir to \%s", finfo.update_dir);

/* Add wrappers for this directory. They exist only until
the next call to wrap.addfile. */
wrap_addfile (CVSDOTWRAPPER, 1);
finfo.res = NULL;

/* Find the repository associated with our current dir. */
repository = Name_Repository (NULL, finfo.update_dir);
entries = Entries_Open [0, NULL];
finfo.repository = repository;
finfo.entries = entries;

/* We pass force_tag_match as 1. If the directory has a
sticky branch tag, and there is already an RCS file which
does not have that tag, then the head revision is
meaningless to us. */
vers = Version;TS (&info, options, NULL, NULL, 1, 0);
if (vers->version == NULL)
{
    /* No entry available, tag is invalid */
    if (vers->version == NULL)
    {
        /* There is no RCS file either */
        if (vers->version == NULL)
        {
            /* There is no user file either */
            error (0, 0, "nothing known about %s", info.fullname);
            err++;
        }
    }
    else if (!isdir (info.file))
    {
        /* See if a directory exists in the repository with */
        /* the same name. If so, blow this request off. */
        char *dname = xmalloc (strlen (repository) + strlen (info.file) + 10);
        (void) sprintf (dname, "%s/%s", repository, info.file);
        if (isdir (dname))
        {
            error (0, 0, "cannot add file \"%s\" since the directory",
                  info.fullname);
            error (0, 0, "\"%s\" already exists in the repository",
                  dname);
            error (1, 0, "illegal filename overlap");
            free (dname);
        }
    }
    if (vers->options == NULL || vers->options == \"\")
    {
        /* No options specified on command line (or in */
        /* rcs file if it existed, e.g. the file exists */
        /* on another branch). Check for a value from */
        /* the wrapper stuff. */
        if (wrap_name_has (info.file, WRAP_TOCVS))
        {
            /* There is a user file, so build the entry for it */
            if (build_entry (repository, info.file, vers->options,
                            message, entries, vers->tag) != 0)
                err++;
        }
    }
    else
    {
        /* There is a user file, so build the entry for it */
        if (build_entry (repository, info.file, vers->options,
                        message, entries, vers->tag) != 0)
            err++;
    }
    if (quiet)
    {
        if (vers->tag && !revision) {
            error (0, 0, \"\n
            scheduling \"%s\" for addition on branch \"%s\",
            (wrap_name_has (info.file, WRAP_TOCVS)
            ? "wrapper" : "file"), info.fullname, vers->tag);
        }
    }
    else if (!vers->tag) {
        error (0, 0, "scheduling \"%s\" for addition",
               (wrap_name_has (info.file, WRAP_TOCVS)
               ? "wrapper" : "file"), info.fullname);
    }
    else {
        /* New case: the user file exists, but we are not really */
        /* adding it -- we are adding a new remote branch. In this case, */
        /* we create a new empty RCS file with the appropriate */
        /* remote branchpoint */
        add_rcs_file (NULL, vers->srcfile->path, info.fullname, NULL,
                     NULL, NULL, NULL, 0, NULL, NULL, 0, revision, NULL);
```c
else if (RCS_isdead (vers->srcfile, vers->vn_rcs))
{
    if (isdir (finfo.file) & & (wrap_name_matches (finfo.file, WRAP_TOCVS))
        error (0, 0, "] cannot be added because a file of the", info.fullname);
    error (1, 0, "] same name already exists in the repository.");
} else if (vers->nonbranch)
{
    if (isdir (finfo.file))
        error (0, 0, "cannot add file on non-branch tag "]", vers->tag);
    ++err;
} else
{
    if (vers->tag)
        error (0, 0, " cannot add file on non-branch tag "]", vers->tag);
    ++err;
} else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '0')
{
    if (vers->tag_user == NULL)
        error (0, 0, " has already been entered", info.fullname);
    err++;  
} else if (vers->vn_user[0] == '1')
{
    if (vers->tag_user == NULL)
        error (0, 0, "] was added independently by second party", info.fullname);
    err++;  
} else if (vers->vn_user[0] == '1')
{
    if (vers->tag_user == NULL)
        error (0, 0, " cannot resurrect "]", info.fullname);
    err++;  
} else
{
    /* There is an RCS file already, so somebody else must've added it */
    error (0, 0, " has already been entered", info.fullname);
    err++;  
} else
{
    if (vers->srcfile == NULL)
        error (0, 0, " has already been entered", info.fullname);
    err++;  
} else if (vers->tag_user == NULL)
{
    if (vers->vn_rcs == NULL)
        error (0, 0, " cannot resurrect "]", info.fullname);
    err++;  
} else
{
    /* There is an RCS file, so remove the "]" from the */
    vers->version number and restore the file
```
```c
char *tmp = xmalloc(strlen(finfo->file) + 50);

(void) strcpy(tmp, vers->vn_user + 1);
(void) strsep(vsn->vn_user, tmp);
(void) sprintf(tmp, "Resurrected %s", finfo->file);
Register(entries, finfo->file, vers->vn_user, tmp,
vers->options, vers->tag, vers->date, vers->ts_conflict, CVSSroot_directory, finfo->repository);
free(tmp);

/* XXX - bugs here; this really resurrect the head */
/* Note that this depends on the Register above actually
   having written Entries, or else it won't really
   check the file out. */
if (update(2, argv + i - 1) == 0)
{
  error(0, 0, "%s version %s, resurrected",
        finfo->file, vers->vn_user);
}
else
{
  error(0, 0, "could not resurrect %s", finfo->file);
  err++;
}

#else

#ifdef SERVER_SUPPORT
        if (server_active && begin_added_files != added_files)
        server_checked_in (finfo->file, finfo->update_dir, repository);
#endif

free(repository);
EntriesClose(entries);
if (restore_cwd (cwd, NULL))
  error_exit ();
free_cwd (cwd);
free(finfo->file);

if (added_files)
  error(0, 0, "use 'ls commit' to add %s permanently",
        program_name, (added_files == 1) ? "this file" : "these files");
if (message)
  free(message);
return (err);

/*
   * The specified user file is really a directory. So, let's make sure that
   * it is created in the RCS source repository, and that the user's directory
   * is updated to include a CVS directory.
   *
   * Returns 1 on failure, 0 on success.
   */
static int
add_directory (finfo)
struct file_info *finfo;
```
The code snippet is from the CVS (Concurrent Versions System) source code. It is part of a function that handles repository operations and file attributes. The code is written in C and includes macros and functions for managing file attributes, directories, and repository operations. The snippet includes error handling, directory creation, and file attribute manipulation. The comments are in the code, indicating the purpose of certain sections and variables. The code is intricate and relies on several macros and functions, which are likely defined elsewhere in the CVS source code. The snippet provides a glimpse into the inner workings of CVS, including how directories are handled, file attributes are managed, and error checking is performed.
client/server, doesn't ask in the right way for GUIs, etc.
A better way of making it harder to accidentally add directories would be to have to add and commit directories like for files. The code was #if 0'd at least since CVS 1.5. */

if (!noexec)
{
    omask = umask (cvsvumask);
    if (CVS_MKDIR (rcsdir, 0777) < 0)
    {
        error (0, errno, "cannot mkdir %s", rcsdir);
        (void) umask (omask);
        goto out;
    }
    (void) umask (omask);
}

/* Now set the default file attributes to the ones we inherited from the parent directory. */

fileattr_setdir (rcsdir);
fileattr_setall (NULL, atts);
fileattr_write ();
fileattr_free (atts);

if (atts != NULL)
free (atts);

/* *
Set up an update list with a single title node for UpdateLogFile
*/

ulist = getlist ();
p = getnode ();
p->type = UPDATE;
p->delproc = update_delproc;
p->key = xstrdup ("\# New directory");
li = (struct logfile_info *) malloc (sizeof (struct logfile_info));
li->type = T_TITLE;
li->tag = xstrdup (tag);
li->rev = li->rev giveaways = NULL;
p->data = (char *) li;
(void) addnode (ulist, p);
UpdateLogFile (rcsdir, message, (FILE *) NULL, ulist);
dellist (*ulist);
}

#define SERVER_SUPPORT
if (server == active)
    CreateAdmin (".*", info->fullname, rcsdir, tag, date, nonbranch, 0);
else
    CreateAdmin (".*", info->fullname, rcsdir, tag, date, nonbranch, 0);

check :
if (tag)
    free (tag);
if (date)
    free (date);
if (restorecwd (cwd, NULL))
    error_exit (1);
free_cwd (cwd);

SubdirRegister (entries, (char *) NULL, dir);
cvsoutput (message, 0);
free (rcsdir);
free (message);
return (0);

out:
if (restorecwd (cwd, NULL))
    error_exit (1);
free_cwd (cwd);
if (rcsdir != NULL)
    free (rcsdir);
return (0);
}

/* * Builds an entry for a new file and sets up "CVS/[file].[tag] by interrogating the user. Returns non-zero on error. */
*/

static int buildentry (repository, user, options, message, entries, tag)
char *repository;
char *user;
char *options;
char *message;
List *entries;
char *tag;

{

char *fname;
char *line;
FILE *fp;

if (noexec)
    return (0);

/*
 * The requested log is read directly from the user and stored in the
 * file user,t. If the "message" argument is set, use it as the
 * initial creation log (which typically describes the file).
 */
fname = xmalloc(strlen(user) + 80);
(void) sprintf(fname, "%s/%s%s", CVSADM, user, CVSEXT_LOG);
fp = open_file (fname, "w+");
if (message && fputs (message, fp) == EOF)
    error (1, errno, "cannot write to %s", fname);
if (fclose(fp) == EOF)
    error(1, errno, "cannot close %s", fname);
free (fname);

/*
 * Create the entry now, since this allows the user to interrupt us above
 * without needing to clean anything up (well, we could clean up the
 * .t file, but who cares).
 */
line = xmalloc(strlen(user) + 20);
(void) sprintf(line, "Initial %s", user);
Register (entries, user, "0", line, options, tag, (char *) 0, (char *) 0, CV$root_directory, repository);
free (line);
return (0);
A.2 admin.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk  
 * Copyright (c) 1989-1992, Brian Berliner  
 * You may distribute under the terms of the GNU General Public License as  
 * specified in the README file that comes with the CVS source distribution.  
 * Administration ("cvs admin")  
 */

#include "cvs.h"
#endif
#include <grp.h>
#include <assert.h>

static Dtype admin_dirproc PROTO ((void * callerdat, char * dir,  
    char * repos, char * update_dir,  
    List * entries));

static int admin_fileproc PROTO ((void * callerdat, struct file info * finfo));

static const char * const admin_usage[] = {
    "Usage: %s %s rcs-options files...
    (Specify the --help global option for a list of other help options)
    NULL
  
	*/
  

struct admin data {
  /* Set default branch (-b). It is "-b" followed by the value  
     given, or NULL if not specified, or merely "-b" if -b is  
     specified without a value. */
  char * branch;
  /* Set comment leader (-c). It is "-c" followed by the value  
     given, or NULL if not specified. The comment leader is  
     relevant only for old versions of RCS, but we let people set it  
     anyway. */
  char * comment;
  /* Set strict locking (-L). */
  int set strict;
  /* Set nonstrict locking (-U). */
  int set nonstrict;
  /* Delete revisions (-o). It is "-o" followed by the value specified. */
  char * delete revs;
  /* Keyword substitution mode (-k), e.g. "-kb". */
  char * kflag;
  /* Description (-t). See sanity.sh for various meanings about  
     files and stdin and such. "" if -t specified without an  
     argument. It is "-t" followed by the argument. */
  char * desc;
  /* Interactive (-I). Problematic with client/server. */
  int interactive;
  /* Quiet (-q). Not the same as the global -q option, which is a bit  
     on the confusing side, perhaps. */
  int quiet;
  /* This is the cheesy part. It is a vector with the options which  
     we don't deal with above (e.g. "-afoo", "-abar,baz"). In the future  
     this presumably will be replaced by other variables which break  
     out the data in a more convenient fashion. AV as well as each of  
     the strings it points to is malloc'd. */
  int ac;
  char ** av;
  int av alloc;
};

/* Add an argument. OPT is the option letter, e.g. 'a'. ARG is the  
   argument to that option, or NULL if omitted (whether NULL can actually  
   happen depends on whether the option was specified as optional to  
   getopt). */
static void
arg_add (dat, opt, arg)  
struct admin data * dat;  
int opt;  
char * arg;  
{
  char * newelt = xmalloc ((arg == NULL ? 0 : strlen (arg)) + 3);
```c
strcpy (newelt, "-");  
newelt[1] = opt;  
if (arg == NULL)  
   newelt[2] = '\0';  
else  
   strcpy (newelt + 2, arg);  
if (dat->av_alloc == 0)  
{  
dat->av_alloc = 1;  
dat->av = (char **) malloc (dat->av_alloc + sizeof (*dat->av));  
}  
else if (dat->ac >= dat->av_alloc)  
{  
dat->av_alloc += 2;  
dat->av = (char **) realloc (dat->av,  
dat->av_alloc + sizeof (*dat->av));  
}  
else  
   dat->av[dat->ac++] = newelt;  

int
admin (argc, argv)  
int argc;  
char ** argv;  
{  
int err;  
#define CVS_ADMIN GROUP  
struct group * grp;  
struct group * getgrnam();  
#undef CVS_ADMIN GROUP  
int c;  
int i;  
if (argc < = 1)  
   usage (admin_usage);  
#define CVS_ADMIN GROUP  
grp = getgrnam(CVS_ADMIN GROUP);  
/* skip usage right check if group CVS_ADMIN GROUP does not exist */  
if (grp != NULL)  
{  
   char * me = getcaller();  
   char * grnam = grp->gr_mem;  
   int denied = 1;  
   while (*grnam)  
   {  
      if (strcmp (*grnam, me) == 0)  
      {  
          denied = 0;  
          break;  
      }  
      grnam++;  
   }  
   if (denied)  
      error (1, 0, "usage is restricted to members of the group %s",  
           CVS_ADMIN GROUP);  
}  
#undef CVS_ADMIN GROUP  
wrap_setup ();  
memset (&admin_data, 0, sizeof admin_data);  
/* TODO: get rid of '-' switch notation in admin_data.  For  
   example, admin_data->branch should be not '-bfoo' but simply 'foo'. */  
optind = 0;  
while ((c = getopt (argc, argv,  
           "+ib::c:a:A:e:l::u::LUn:N:m:o:s:t::IqxV:k:")) != -1)  
{  
   switch (c)  
   {  
   case 'i':  
      /* This has always been documented as useless in cvs.texinfo  
       and it really is admin_fileproc silently does nothing  
       if version user is NULL. */  
      error (0, 0, "the -i option to admin is not supported");  
      goto usage_error;  
   case 'b':  
   if (admin_data.branch != NULL)  
      {  
       error (0, 0, "duplicate 'b' option");  
       goto usage_error;  
   }  
```

if (optarg == NULL)
    admin_list.branch = xstrdup (*argv);
else
    {
        admin_list.branch = xmalloc (strlen optarg + 5);
        strcpy (admin_list.branch, *argv);
        strcat (admin_list.branch, optarg);
    }
break;

case 's':
    if (admin_list.comment != NULL)
        {
            error (0, 0, "duplicate 'c' option");
            goto usageerror;
        }
    admin_list.comment = xmalloc (strlen optarg + 5);
    strcpy (admin_list.comment, optarg);
    strcat (admin_list.comment, optarg);
break;

case 'a':
    add (admin_list, 'a', optarg);
    break;

case 'A':
    /* In the client/server case, this is cheesy because
     * we just pass along the name of the RCS file, which
     * then will want to exist on the server. This is
     * accidental; having the client specify a pathname on
     * the server is not a design feature of the protocol. */
    add (admin_list, 'A', optarg);
    break;

case 'e':
    add (admin_list, 'e', optarg);
    break;

case 'l':
    /* Note that multiple -l options are legal. */
    add (admin_list, 'l', optarg);
    break;

case 'u':
    /* Note that multiple -u options are legal. */
    add (admin_list, 'u', optarg);
    break;

case 'L':
    /* Probably could also complain if -L is specified multiple
     * times, although RCS doesn't and I suppose it is reasonable
     * just to have it mean the same as a single -L. */
    if (admin_list.set_nonstrict)
        {
            error (0, 0, "-0 and -L are incompatible");
            goto usageerror;
        }
    admin_list.set_nonstrict = 1;
break;

case 'U':
    /* Probably could also complain if -U is specified multiple
     * times, although RCS doesn't and I suppose it is reasonable
     * just to have it mean the same as a single -U. */
    if (admin_list.set_nonstrict)
        {
            error (0, 0, "-0 and -L are incompatible");
            goto usageerror;
        }
    admin_list.set_nonstrict = 1;
break;

case 'n':
    /* Mostly similar to cvs tag. Could also be parsing
     * the syntax of optarg, although for now we just pass
     * it to rcs as-is. Note that multiple -n options are
     * legal. */
    add (admin_list, 'n', optarg);
    break;

case 'N':
    /* Mostly similar to cvs tag. Could also be parsing
     * the syntax of optarg, although for now we just pass
     * it to rcs as-is. Note that multiple -N options are
     * legal. */
    add (admin_list, 'N', optarg);
    break;

case 'c':
    /* Change log message. Could also be parsing the syntax
     * of optarg, although for now we just pass
     * it to rcs as-is. Note that multiple -N options are
     * legal. */
    break;
of optarg, although for now we just pass it to rcs
as-is. Note that multiple -m options are legal. */
arg_add (&admin_data, 'm', optarg);
break;
case 'o':
/* Delete revisions. Probably should also be parsing
the syntax of optarg, so that the client can give errors
rather than making the server take care of that.
Other than that I'm not sure whether it matters much
whether we parse it here or in admin/apply.
Note that multiple -a options are illegal, in RCS
as well as here. */
if (admin_data.delete_revs != NULL)
{
  error (0, 0, "duplicate -o option");
goto usage_error;
}
if (optarg == NULL)
admin_data.delete_revs = xmalloc (strlen(optarg) + 5);
strcpy (admin_data.delete_revs, "-a");
strcat (admin_data.delete_revs, optarg);
break;
case 's':
/* Note that multiple -s options are legal. */
arg_add (&admin_data, 's', optarg);
break;
case 't':
if (admin_data.desc != NULL)
{
  error (0, 0, "duplicate 't' option");
goto usage_error;
}
if (optarg == NULL)
admin_data.desc = xstrdup("-t");
else
{
  admin_data.desc = xmalloc (strlen(optarg) + 5);
  strcpy (admin_data.desc, "-t");
  strcat (admin_data.desc, optarg);
}
break;
case 'v':
/* At least in RCS this can be specified several times,
with the same meaning as being specified once. */
admin_data.interactive = 1;
break;
case 'q':
admin_data.quiet = 1;
break;
case 'x':
error (0, 0, "the "x option has never done anything useful");
error (0, 0, "RCS files in CVS always end in .v");
goto usage_error;
break;
case 'y':
/* No longer supported. */
error (0, 0, "the -y option is obsolete");
break;
case 'k':
if (admin_data.kflag != NULL)
{
  error (0, 0, "duplicate '-k' option");
goto usage_error;
}
admin_data.kflag = RCS::check_kflag (optarg);
break;
default:
case '?':
/* getopt will have printed an error message. */
usage_error:
/* Don't use command_name; it might be "server". */
error (1, 0, "specify 2s -H admin for usage information",
program_name);
break;
}
argv = argv + optind;
for (i = 0; i < admin_data.ac; ++i)
{
  assert (admin_data.argv[i][0] == '-');
switch (admin_data.av[i][2])
{
    case 'n':
    case 'r':
    case 'a':
        check_numeric (&admin_data.av[i][2], argc, argv);
        break;
    default:
        break;
}

370  if (admin_data.branch != NULL)
    check_numeric (admin_data.branch + 2, argc, argv);
if (admin_data.delete_revs != NULL)
{
    char *p;
    check_numeric (admin_data.delete_revs + 2, argc, argv);
    p = strchr (admin_data.delete_revs + 2, ' ');
    if (p != NULL && digit (p[1]))
        check_numeric (p + 1, argc, argv);
    else if ((p != NULL && p[1] == '1' && digit (p[2]))
        check_numeric (p + 2, argc, argv);
}
#endif
if (client_active)
{
    /* We're the client side. Fire up the remote server. */
    start_server () ;

390  ign_setup ();

        /* Note that option with arg does not work for us, because some 
        of the options must be sent without a space between the option 
        and its argument. */
        if (admin_data.interactive)
            error (1, 0, "-I option not useful with client/server");
    sendarg (admin_data.branch);
    sendarg (admin_data.branch);
    sendarg (admin_data.comment);
    if (admin_data.set_nonstrict)
        sendarg ("-2");
    if (admin_data.set_nonstrict)
        sendarg ("-2");
    if (admin_data.data_delete_revs != NULL)
        sendarg (admin_data.data_delete_revs);
    if (admin_data.data_desc != NULL)
        sendarg (admin_data.data_desc);
    if (admin_data.data_kflag != NULL)
        sendarg (admin_data.data_kflag);
    for (i = 0; i < admin_data.ac; ++i)
        sendarg (admin_data.av[i]);
        sendarg (admin_data.av[i]);
        sendarg (admin_data.av[i]);
        sendarg (admin_data.av[i]);

        err = sendresponses_and_close ();
    goto return1;
}
#endif
/* CLIENT_SUPPORT */

lock->free_for_write (argc, argv, 0, 0);

    err = start_recursion (admin_fileproc, (FILESONEPROC) NULL, admin_dirproc,
    (DIRLEAVEPROC) NULL, (void *)&admin_data, 
    argc, argv, 0,
    LOCAL, 0, 0, (char *) NULL, 1);

430  LockCleanup ();

    return 1;
    if (admin_data.branch != NULL)
        free (admin_data.branch);
    if (admin_data.comment != NULL)
        free (admin_data.comment);
    if (admin_data.data_delete_revs != NULL)
        free (admin_data.data_delete_revs);
    if (admin_data.data_kflag != NULL)
        free (admin_data.data_kflag);
    if (admin_data.data_desc != NULL)
        free (admin_data.data_desc);
    for (i = 0; i < admin_data.ac; ++i)
        free (admin_data.av[i]);
    if (admin_data.av != NULL)
        free (admin_data.av);
return (err);

/* Called to run "rcs" on a particular file.
 * ARGUSED */
static int adminfileproc (callerdat, finfo)
    void *callerdat;
struct filedata *finfo = finfo;
{
    struct admindata *admindata = (struct admindata *) callerdat;
    VersionTS *vers;
    char version;
    int i;
    int status = 0;
    RCSNode *rcs, *rcs2;
    vers = Version_TLS (finfo, NULL, NULL, NULL, 0, 0);

    version = vers->vnum;
    if (version == NULL)
        goto exitfunc;
    else if (!strcmp (version, "0") == 0)
    {
        error (0, 0, "cannot admin newly added file 'a'", finfo->file);
        goto exitfunc;
    }
    rcs = vers->rcsfile;

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    status = 0;
    if (admindata->quiet)
    {
        cvs_output ("RCS file: ", 0);
        cvs_output (rcs->path, 0);
        cvs_output ("\n", 1);
    }
    if (admindata->branch != NULL)
    {
        char *branch = admindata->branch[2];
        if (branch[1] == '0' && !isdigit (branch[1]))
        {
            branch = RCS_whatbranch (rcs, admindata->branch + 2);
            if (branch == NULL)
            {
                error (0, 0, "Symbolic name \%s is undefined.",
                        rcs->path, admindata->branch + 2);
                status = 1;
            }
        }
        if (status == 0)
            RCS_setbranch (rcs, branch);
        if (branch != NULL && branch != &admindata->branch[2])
            free (branch);
    }

    if (admindata->comment != NULL)
    {
        if (rcs->comment != NULL)
            free (rcs->comment);
        rcs->comment = xstrdup (admindata->comment + 1);
    }
    if (admindata->setstrict)
        rcs->strict_locks = 1;
    if (admindata->setnonstrict)
        rcs->strict_locks = 0;

    if (admindata->deletecevs != NULL)
    {
        char ss, st, *rev1, *rev2;
        /* Set for ", clear for :- */
        int inclusive;
        char *t2;
        s = admindata->deletecevs + 1;
        inclusive = 1;
        t = strchr (s, ',');

        if (t != NULL)
        {
            if (t[1] == ',')
            {
                inclusive = 0;
                t2 = t + 2;
            } else
                t2 = t + 1;
if (t == NULL) {
    /* -rev */
    rev1 = xstrdup (s);
    rev2 = xstrdup (p);
}
else if (t == s) {
    /* -o.rev2 */
    rev1 = NULL;
    rev2 = xstrdup (t2);
} else {
    t = '0';
    rev1 = xstrdup (s);
    t = '1'; /* probably unnecessary */
    if (t2 == '0')
        /* -rev1 */
        rev2 = NULL;
    else
        /* -rev1:rev2 */
        rev2 = xstrdup (t2);
}

if (rev1 == NULL && rev2 == NULL) {
    /* RCS segfaults if -o: is given */
    error (0, 0, "no valid revisions specified in \"%s\" option",
          admin_data->delete_revs);
    status = 1;
} else {
    status |= RCS_delete_revs (rcs, rev1, rev2, inclusive);
}

if (rev1) {
    free (rev1);
    free (rev2);
}
}

if (admin_data->dec != NULL) {
    free (rcs->dec);
    rcs->dec = NULL;
}

if (admin_data->desc[3] == '0')
    rcs->desc = xstrdup (admin_data->desc + 3);
else {
    char *descfile = admin_data->desc + 2;
    size_t bufsize = 0;
    size_t len;

    /* If -t specified with no argument, read from stdin. */
    if (descfile == '\0')
        descfile = NULL;
    getfile (descfile, descfile, "r", &rcs->desc, &bufsize, &len);
}

if (admin_data->kflag != NULL) {
    char *kflag = admin_data->kflag + 2;
    if (rcs->expand || strcmp (rcs->expand, kflag) != 0) {
        if (rcs->expand)
            free (rcs->expand);
        rcs->expand = xstrdup (kflag);
    }
}

/* Handle miscellaneous options. TODO: decide whether any or all
of these should have their own fields in the admin_data
structure. */
for (i = 0; i < admin_data->ac; ++i) {
    char *arg;
    char *users;
    int argc, u;
    Node *av;
    RCSVers *delta;
    arg = admin_data->av[i];
    switch (arg[0])
    {...
\{ 
  case 'a': /* fall through */
  case 'e':
    line2argv (&argc, &users, arg + 2, " ,\t\n");
    if (argc[1] == "\'")
      for (u = 0; u < argc; ++u)
        RCS_setaccess (rcs, users[u]);
    else
      for (u = 0; u < argc; ++u)
        RCS_setaccess (rcs, users[u]);
    free (argc);
    freerccnode (&rcs2);
    break;
  case 'A':
    /* See admin-19a-admin and friends in sanity.sh for 
     * relative pathnames. It makes sense to think in 
     * terms of a syntax which give pathnames relative to 
     * the repository or repository corresponding to the 
     * current directory or some such (and perhaps don't 
     * include \), but trying to worry about such things 
     * is a little pointless unless you first worry about 
     * whether "cvs admin -A" as a whole makes any sense 
     * (currently probably not, as access lists don't 
     * affect the behavior of CVS). */
    rcs2 = RCS_parsercsfile (arg + 2);
    if (rcs2 == NULL)
      error (1, 0, "cannot continue");
    p = xstrdup (RCSt_getaccess (rcs2));
    line2argv (&argc, &users, p, " ,\t\n");
    free (p);
    freerccnode (&rcs2);
    for (u = 0; u < argc; ++u)
      RCS_setaccess (rcs, users[u]);
    free (users);
    break;
  case 'n': /* fall through */
  case 'N':
    if (argc[2] == ''\n'')
      {
        cvouterr ("missing symbolic name after ", 0);
        cvouterr (arg[0], 0);
        cvouterr ("'\n', 1);
        break;
      }
    p = strchr (arg, '\
');
    if (p == NULL)
      {
        if (rCSt_deltag (rcs, arg + 2) != 0)
          {
            error (0, 0, "\n: Symbolic name \n is undefined.",
                   rcs->path,
                   arg + 2);
            status = 1;
            continue;
          }
        break;
      }
    *p = ''\n'';
    tag = xstrdup (arg + 2);
    *tag++ = ''\n'';
    /* Option 'n' signals an error if this tag is already bound. */
    if (argc[1] == ''n'')
      {
        n = findnode (RCSt_symbols (rcs), tag);
        if (n != NULL)
          {
            error (0, 0, 
                      "\n: symbolic name \n already bound to \n",
                     rcs->path,
                     tag, n->data);
            status = 1;
            free (tag);
            continue;
          }
      }
  } /* Attempt to perform the requested tagging. */
  if ((*p == 0) && (rev = RCS_head (rcs)))
    { /* tag2rev may exit */
      RCS_checktag (tag); /* exit if not a valid tag */
      RCS_settag (rcs, tag, rev);
      free (rev);
    }
else
  {
    /* See admin-19a-admin and friends in sanity.sh for 
     * relative pathnames. It makes sense to think in 
     * terms of a syntax which give pathnames relative to 
     * the repository or repository corresponding to the 
     * current directory or some such (and perhaps don't 
     * include \), but trying to worry about such things 
     * is a little pointless unless you first worry about 
     * whether "cvs admin -A" as a whole makes any sense 
     * (currently probably not, as access lists don't 
     * affect the behavior of CVS). */
    rcs2 = RCS_parsercsfile (arg + 2);
    if (rcs2 == NULL)
      error (1, 0, "cannot continue");
    p = xstrdup (RCSt_getaccess (rcs2));
    line2argv (&argc, &users, p, " ,\t\n");
    free (p);
    freerccnode (&rcs2);
    for (u = 0; u < argc; ++u)
      RCS_setaccess (rcs, users[u]);
    free (users);
    break;
  }

{
    error(0, 0,
        "%s: Symbolic name or revision %s is undefined",
        rcs->path, p);
    status = 1;
} free(tag);
break;
case 's':
    p = strchr(arg, '1');
    if (p == NULL)
    {
        tag = xstrdup(arg + 2);
        rev = RCS_head(rcs);
    }
    else
    {
        *p = '\0';
        tag = xstrdup(arg + 2);
        *p++ = '1';
        rev = xstrdup(p);
    }
    revnum = RCS_gettag(rcs, rev, 0, NULL);
    free(rev);
    if (revnum != NULL)
        n = findnode(rcs->versions, revnum);
    if (revnum == NULL || n == NULL)
    {
        error(0, 0,
            "%s: can't get state of nonexisting revision %s",
            rcs->path);
        rev);
    }
    if (revnum != NULL)
        free(revnum);
    status = 1;
    continue;
    delta = (RCSVers *) n->data;
    free(delta->state);
    delta->state = tag;
    break;
}
case 'n':
    p = strchr(arg, '1');
    if (p == NULL)
    {
        error(0, 0,
            "%s: -n option lacks revision number",
            rcs->path);
        status = 1;
        continue;
    }
    *p = '\0';
    rev = RCS_gettag(rcs, arg + 2, 0, NULL);
    if (rev == NULL)
    {
        error(0, 0,
            "%s: no such revision %s", rcs->path, rev);
        status = 1;
        continue;
    }
    *p++ = '1';
    msg = p;
    n = findnode(rcs->versions, rev);
    delta = (RCSVers *) n->data;
    if (delta->text == NULL)
    {
        delta->text = (Deltatext *) xmalloc(sizeof(Deltatext));
        memset((void *) delta->text, 0, sizeof(Deltatext));
        delta->text->version = xstrdup(delta->version);
        delta->text->log = make_message_legal(msg);
    }
    break;
case 'l':
    status |= RCS_lock(rsc, arg[2] + arg + 2 : NULL, 0);
    break;
case 'u':
    status |= RCS_unlock(rsc, arg[2] + arg + 2 : NULL, 0);
    break;
case 'v':
    default: assert(0); /* can't happen */
}
/* TODO: reconcile the weird discrepancies between
   admin->data->quiet and quiet. */
if (status == 0)
{
    RCS_write(rsc, NULL, NULL);
    if (shaming(data->quiet))
        cvo_output("done\n", 5);
810  }
else
{
    
    /* Note that this message should only occur after another 
    message has given a more specific error. The point of this 
    additional message is to make it clear that the previous problems 
    caused CVS to forget about the idea of modifying the RCS file. */
    error (0, 0, "cannot modify RCS file for '%s'", info->file);

    /* Upon failure, we want to abandon any changes made to the 
    RCS data structure. Forcing a reparse does the trick,
    but leaks memory and is kludgey. Should we export
    free_rcsnode_contents for this purpose? */
    RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);
}
exitfunc:
    freevers (&vers);
    return status;
}

830  
    
    /* Print a warm fuzzy message 
    */
    
    static Dtype
admin_dirproc (callerdat, dir, repos, update_dir, entries)
    
    void *callerdat;
    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
    
    { 
    if (!quiet)
        error (0, 0, "Administrating %s", update_dir);
    
        return (R_PROCESS);
    }
A.3 buffer.c

/* Code for the buffer data structure. */

#include <assert.h>
#include "sys.h"
#include "buffer.h"

#ifdef (SERVER_SUPPORT) || defined (CLIENT_SUPPORT)
/* OS/2 doesn't have EIO.FIXME: this whole notion of turning
a different error into EIO strikes me as pretty dubious. */
#ifdef (EIO)
define EIO EBADPOS
#endif

/* Leaked list of available buffer data structures. */
static struct bufferdata *freebufferdata;

/* Local functions. */
static void buf_default_memory_error PROTO ((struct buffer *));
static void allocatebufferdata PROTO ((void));
static struct bufferdata *getbufferdata PROTO ((void));

/* Initialize a buffer structure. */
struct buffer *
buf_initialize (input, output, flush, block, shutdown, memory, closure)
int (*input) PROTO ((void *, char *, int, int, int *));
int (*output) PROTO ((void *, const char *, int, int *));
int (*flush) PROTO ((void *));
int (*block) PROTO ((void *));
int (*shutdown) PROTO ((void *));
void (*memory) PROTO ((struct buffer *));
void *closure;
{
struct buffer *buf;
buf = (struct buffer *) xmalloc (sizeof (struct buffer));
buf->data = NULL;
buf->last = NULL;
buf->nonblocking = 0;
buf->input = input;
buf->output = output;
buf->flush = flush;
buf->block = block;
buf->shutdown = shutdown;
buf->memory_error = memory ? memory : buf_default_memory_error;
buf->closure = closure;
return buf;
}

/* Free a buffer structure. */
void
buf_free (buf)
{
struct buffer *buf;
if (buf->data != NULL)
{
buf->last->next = freebufferdata;
freebufferdata = buf->data;
}
free (buf);
}

/* Initialize a buffer structure which is not to be used for I/O. */
struct buffer *
buf_nonio_initialize (memory)
void (*memory) PROTO ((struct buffer *));
{
return (buf_initialize
(int (*) PROTO ((void *, char *, int, int, int *))) NULL,
(int *) PROTO ((void *, const char *, int, int *))) NULL,
(int *) PROTO ((void *)) NULL,
(int *) PROTO ((void *, int)) NULL,
(int *) PROTO ((void *)) NULL,
(void *) NULL);
}

/* Default memory error handler. */
static void
buf_default_memory_error (buf)
struct buffer *buf;
{
error (1, 0, "out of memory");
}
/∗ Allocate more buffer data structures. ∗/  
static void  
allocate_buffer()  
{  
    struct buffer_data *alc;  
    char space;  
    int i;  
    /∗ Allocate buffer data structures in blocks of 16. ∗/  

#define ALLOC_COUNT (16)  

alc = { (struct buffer_data *)  
    malloc (ALLOC_COUNT + sizeof (struct buffer_data));  
    space = (char *) valloc (ALLOC_COUNT + BUFFER_DATA_SIZE);  
    if (alc == NULL || space == NULL)  
        return;  
    for (i = 0; i < ALLOC_COUNT; i++, alc++, space += BUFFER_DATA_SIZE)  
    {  
        alc->next = free_buffer_data;  
        free_buffer_data = alc;  
        alc->text = space;  
    }  
    /∗ Get a new buffer data structure. ∗/  
    static struct buffer_data *  
    get_buffer_data()  
    {  
        struct buffer_data *ret;  
        if (free_buffer_data == NULL)  
        {  
            allocate_buffer();  
            if (free_buffer_data == NULL)  
                return NULL;  
        }  
        ret = free_buffer_data;  
        free_buffer_data = ret->next;  
        return ret;  
    }  
    /∗ See whether a buffer is empty. ∗/  
    int  
    buf_empty_p (buf)  
    {  
        struct buffer_data *data;  
        for (data = buf->data; data != NULL; data = data->next)  
            if (data->size > 0)  
                return 1;  
    }  

#define SERVER_FLOWCONTROL  
/∗ Count how much data is stored in the buffer.  
∗ Note that each buffer is a malloc'ed chunk BUFFER_DATA_SIZE. ∗/  

int  
buf_count_mem (buf)  
{  
    struct buffer_data *data;  
    int mem = 0;  
    for (data = buf->data; data != NULL; data = data->next)  
        mem += BUFFER_DATA_SIZE;  
    return mem;  
}  

#undef SERVER_FLOWCONTROL  
/∗ Add data DATA of length LEN to BUF. ∗/  

void  
buf_output (buf, data, len)  
{  
    struct buffer *buf;  
    const char *data;  
    int len;  
    if (buf->data != NULL  
        && ((buf->last->text + BUFFER_DATA_SIZE)  
            - (buf->last->bufp + buf->last->size))
180  
{ 
    memcpy (buf->last->bufp + buf->last->size, data, len); 
    buf->last->size += len; 
    return; 
}

while (1)
{ 
    struct buffer *newdata; 
    
    newdata = getbufferdata (); 
    if (newdata == NULL) 
    { 
        (buf->memory_error) (buf); 
        return; 
    }
    if (buf->data == NULL) 
        buf->data = newdata; 
    else 
    
        buf->last->next = newdata; 
        newdata->next = NULL; 
        buf->last = newdata; 
        newdata->bufp = newdata->text; 
    if (len <= BUFFER_DATA_SIZE) 
    { 
        newdata->size = len; 
        memcpy (newdata->text, data, len); 
        return; 
    }
    newdata->size = BUFFER_DATA_SIZE; 
    memcpy (newdata->text, data, BUFFER_DATA_SIZE); 
    data += BUFFER_DATA_SIZE; 
    len -= BUFFER_DATA_SIZE; 
}

220  /*NOTREACHED*/
} 

/* Add a \0 terminated string to BUF. */

void 
buf_output0 (buf, string)
    
    struct buffer *buf; 
    const char *string; 
{
    buf_output (buf, string, strlen (string)); 
}

/* Add a single character to BUF. */

void 
buf_append_char (buf, ch)
    
    struct buffer *buf; 
    int ch; 
{
    if (buf->data != NULL 
        && (buf->last->text + BUFFER_DATA_SIZE 
            != buf->last->bufp + buf->last->size)) 
    { 
        (buf->last->bufp + buf->last->size) = ch; 
        ++buf->last->size; 
    } 
    else 
    { 
        char b; 
        
        b = ch; 
        buf_output (buf, &b, 1); 
    }

    /* Send all the output we’ve been saving up. Returns 0 for success or 
    * errno code. If the buffer has been set to be nonblocking, this 
    * will just write until the write would block. */

int 
buf_sendoutput (buf)
    
    struct buffer *buf; 
{
    if (buf->output == NULL) 
        abort (); 

}
while (buf->data != NULL)
{
    struct buffer_data *data;
    data = buf->data;
    if (data->size > 0)
    {
        int status, nbytes;
        status = (+buf->output) (buf->closure, data->bufp, data->size, &nbytes);
        if (status != 0)
        {
            /* Some sort of error. Discard the data, and return. */
            buf->last->next = free_buffer(data);
            free_buffer(data) = buf->data;
            buf->data = NULL;
            buf->last = NULL;
            return status;
        }
        if (nbytes != data->size)
        {
            /* Not all the data was written out. This is only permitted in nonblocking mode. Adjust the buffer, and return. */
            assert (buf->nonblocking);
            data->size -= nbytes;
            data->bufp += nbytes;
            return 0;
        }
    }
    buf->data = data->next;
    data->next = free_buffer(data);
    free_buffer(data) = data;
}
buf->last = NULL;
return 0;
/*
 * Flush any data queued up in the buffer. If BLOCK is nonzero, then
 * if the buffer is in nonblocking mode, put it into blocking mode for
 * the duration of the flush. This returns 0 on success, or an error
 * code.
 */
int buf_flush (buf, block)

struct buffer *buf;
int block;
{
    int nonblocking;
    int status;
    if (buf->flush == NULL)
        abort ();
    nonblocking = buf->nonblocking;
    if (nonblocking & block)
    {
        status = set_block (buf);
        if (status != 0)
            return status;
    }
    status = buf_end_output (buf);
    if (status == 0)
        status = (+buf->flush) (buf->closure);
    if (nonblocking & block)
    {
        int blockstat;
        blockstat = set_nonblock (buf);
        if (status == 0)
            status = blockstat;
    }
    return status;
}
/* Set buffer BUF to nonblocking I/O. Returns 0 for success or errno code. */

int set_nonblock (buf)

    struct buffer *buf;
{
    int status;
    if (buf->nonblocking)
        return 0;
    if (buf->block == NULL)
        abort();
    status = (*buf->block) (buf->closure, 0);
    if (status != 0)
        return status;
    buf->nonblocking = 1;
    return 0;
}

/* Set buffer BUF to blocking I/O. Returns 0 for success or errno code. */

int set_block (buf)

    struct buffer *buf;
{
    int status;
    if (!buf->nonblocking)
        return 0;
    if (buf->block == NULL)
        abort();
    status = (*buf->block) (buf->closure, 1);
    if (status != 0)
        return status;
    buf->nonblocking = 0;
    return 0;
}

/* Send a character count and some output. Returns errno code or 0 for success. *
   Sending the count in binary is OK since this is only used on a pipe within the same system. */

int buf_send_counted (buf)

    struct buffer *buf;
{
    int size;
    struct buffer data *data;
    size = 0;
    for (data = buf->data; data != NULL; data = data->next)
        size += data->size;
    data = get_buffer_data ();
    if (data == NULL)
        { (*buf->memory_error) (buf);
            return ENOMEM;
        }
    data->next = buf->data;
    buf->data = data;
    if (buf->last == NULL)
        buf->last = data;
    data->bufp = data->text;
    data->size = sizeof (int);
    *((int *) data->text) = size;
    return buf_send_output (buf);
}

/* Send a special count. COUNT should be negative. It will be handled specially by buf_copy_counted. This function returns 0 or an errno code. */

*send_counted (buf, COUNT)}
within the same system.

```c
int
bufsendspecialcount (buf, count)
struct buffer *buf;
int count;
{
struct buffer *data;
data = getbufferdata ();
if (data == NULL)
{ /*buf->memory_error (buf);
  return ENOMEM; */
data->next = buf->data;
buf->data = data;
if (buf->last == NULL)
buf->last = data;
}
data->bufp = data->text;
data->size = sizeof (int);
*((int *) data->text) = count;
return bufsendoutput (buf);
}
/* Append a list of buffer data structures to an buffer. */
void
bufappenddata (buf, data, last)
struct buffer *buf;
struct buffer data *data;
struct buffer last *last;
{
if (data != NULL)
{
if (buf->data == NULL)
  buf->data = data;
else
  buf->last->next = data;
  buf->last = last;
}
/* Append the data on one buffer to another. This removes the data
from the source buffer. */
void
bufappendbuffer (to, from)
struct buffer *to;
struct buffer *from;
{
bufappenddata (to, from->data, from->last);
from->data = NULL;
from->last = NULL;
}
/* Copy the contents of file F into buffer data structures. We can't
* copy directly into an buffer, because we want to handle failure and
* success differently. Returns 0 on success, or -2 if out of
* memory, or a status code on error. Since the caller happens to
* know the size of the file, it is passed in as SIZE. On success,
* this function sets *RETP and *LASTP, which may be passed to
* bufappenddata.
*/
int
bufreadfile (f, size, retp, lastp)
FILE *f;
long size;
struct buffer data **retp;
struct buffer data **lastp;
{
int status;
*retp = NULL;
*lastp = NULL;
while (size > 0)
{
struct buffer data *data;
int get;
data = getbufferdata ();
if (data == NULL)
{ /*
  *
  *
  *
  */
```
{ status = -2;  
goto error_return; }

if (*retp == NULL)  
  *retp = data;
else  
  (*lastp)->next = data;
  data->next = NULL;
  *lastp = data;

  data->bufp = data->text;
  data->size = 0;

if (size > BUFFER_DATA_SIZE)  
  get = BUFFER_DATA_SIZE;
else  
  get = size;

  errno = EIO;

if (fread (data->text, get, 1, f) != 1)  
  { status = errno;  
    goto error_return;
  }

  data->size += get;
  size += get;

return 0;

error_return:  
if (*retp == NULL)  
  { (*lastp)->next = free_buffer_data;
    free_buffer_data = *retp;
  }

return status;

/*  
* Copy the contents of file F into buffer data structures. We can't  
* copy directly into an buffer, because we want to handle failure and  
* success differently. Returns 0 on success, or -2 if out of  
* memory, or a status code on error. On success, this function sets  
* *RETP and *LASTP, which may be passed to buf_append_data.  
*/

int buf_read_file_to_eof (f, retp, lastp)
FILE *f,
struct buffer_data **retp,
struct buffer_data **lastp;
{
  int status;

  *retp = NULL;
  *lastp = NULL;

  while (!feof (f))  
  {  
    struct buffer_data +data;
    int get, nread;

    data = get_buffer_data ();
    if (data == NULL)  
      { status = -2;  
        goto error_return;
      }

    if (*retp == NULL)  
      *retp = data;
    else  
      (*lastp)->next = data;
    data->next = NULL;
    *lastp = data;

    data->bufp = data->text;
    data->size = 0;

    get = BUFFER_DATA_SIZE;

    errno = EIO;

    nread = fread (data->text, 1, get, f);
    if (nread == 0 && !feof (f))  
      { status = errno;

goto error; return;
}
data->size = nread;
}
return 0;
error; return:
if (*((retp) != NULL)
    { (*lastp)->next = free(bufferdata);
      free(bufferdata = *retp;
    } return status;
}

/* Return the number of bytes in a chain of buffer-data structures. */
int
bufchain_length (buf)
struct buffer-data *buf;
{
    int size = 0;
    while (buf)
    {
        size += buf->size;
        buf = buf->next;
    }
    return size;
}

/* Return the number of bytes in a buffer. */
int
buf_length (buf)
struct buffer *buf;
{
    return bufchain_length (buf->data);
}

/* Read an arbitrary amount of data into an input buffer. The buffer
* will be in nonblocking mode, and we just grab what we can. Return
* 0 on success, or -1 on end of file, or -2 if out of memory, or an
* error code. If COUNTP is not NULL, *COUNTP is set to the number of
* bytes read. */

int
buf_input_data (buf, countp)
struct buffer *buf;
int *countp;
{
    if (buf->input == NULL)
        abort ();
    if (countp != NULL)
        *countp = 0;

    while (1)
    {
        int get;
        int status, nbytes;

        if (buf->data == NULL
            || (buf->last->bfp + buf->last->size
                        == buf->last->text + BUFFER_DATA_SIZE))
            { struct buffer-data *data;
                data = getbufferdata ();
                if (data == NULL)
                    { (*buf->memory_error) (buf);
                        return -2;
                    }
            }
            if (buf->data == NULL)
                buf->data = data;
            else
                { buf->last->next = data;
                  data->next = NULL;
                  buf->last = data;
                  data->bfp = data->text;
                  data->size = 0;
                }
        get = ((buf->last->text + BUFFER_DATA_SIZE)
status = (*buf->input) (buf->closure, 
    buf->last->bufp + buf->last->size, 
    0, get, &nbytes);
if (status != 0)
    return status;
buf->last->size += nbytes;
if (countp != NULL)
    *countp += nbytes;

if (nbytes < get)
    /* If we did not fill the buffer, then presumably we read
     all the available data. */
    return 0;
}
/* NOTREACHED*/

int
buf_readline (buf, line, lenp)
    struct buffer *buf;
    char **line;
    int *lenp;
{
    if (buf->input == NULL)
        abort ();
    *line = NULL;
    while (1)
    {
        int len, finallen = 0;
        struct buffer_data *data;
        char *nl;
        for (data = buf->data; data != NULL; data = data->next)
            {
                nl = memchr (data->bufp, '\012', data->size);
                if (nl != NULL)
                    {
                        finallen = nl - data->bufp;
                        len += finallen;
                        break;
                    }
                len += data->size;
            }
        /* If we found a newline, copy the line into a memory buffer,
         and remove it from BUF. */
        if (data != NULL)
            {
                char *p;
                struct buffer_data *nldata;
                p = malloc (len + 1);
                if (p == NULL)
                    return -2;
                *line = p;
                nldata = data;
                data = buf->data;
                while (data != nldata)
                    {
                        struct buffer_data *next;
                        memcpy (p, data->bufp, data->size);
                        p += data->size;
                        next = data->next;
                        data->next = free(buffer_data);
                        data = next;
                    }
memcpy (p, data->bufp, finallen);
p[finallen] = '\0';
data->size = finallen + 1;
data->bufp = ni + 1;
buf->data = data;
if (lenp != NULL)
  *lenp = len;

return 0;
} /* Read more data until we get a newline. */
while (1)
{
  int size, status, nbytes;
  char *mem;
  if (buf->data == NULL ||
      (buf->last->bufp + buf->last->size == buf->last->text + BUFFER_DATA_SIZE))
  {
    data = get_buffer_data ();
    if (data == NULL)
    {
      (*buf->memory_error) (buf);
      return -2;
    }
  }

  if (buf->data == NULL)
    buf->data = data;
  else
    buf->last->next = data;
  data->next = NULL;
  buf->last = data;
  data->bufp = data->text;
  data->size = 0;
}

mem = buf->last->bufp + buf->last->size;
size = (buf->last->text + BUFFER_DATA_SIZE) - mem;
/* We need to read at least 1 byte. We can handle up to SIZE bytes. This will only be efficient if the underlying communication stream does its own buffering, or is clever about getting more than 1 byte at a time. */
if (status != 0)
  return status;

buf->last->size += nbytes;
/* Optimize slightly to avoid an unnecessary call to memchr. */
if (nbytes == 1)
{
  if (*mem == '\012')
    break;
}
else
{
  if (strchr (mem, '\012', nbytes) != NULL)
    break;
}
}

/* Extract data from the input buffer BUF. This will read up to WANT bytes from the buffer. It will set *RETDATA to point at the bytes, and set *GOT to the number of bytes to be found there. Any buffer call which uses BUF may change the contents of the buffer at *DATA, so the data should be fully processed before any further calls are made. This returns 0 on success, or -1 on end of file, or -2 if out of memory, or an error code. */

int
buf_read_data (buf, want, retdata, got)
struct buffer *buf;
int want, *retdata;
int *got;
{
  if (buf->input == NULL)
    abort ();
while (buf->data != NULL && buf->data->size == 0)
{
    struct buffer_data *next;
    next = buf->data->next;
    buf->data->next = free_buffer_data;
    free_buffer_data = buf->data;
    buf->data = next;
    if (next == NULL)
        buf->last = NULL;
}
if (buf->data == NULL)
{
    struct buffer_data *data;
    int get, status, nbytes;
    data = get_buffer_data ();
    if (data == NULL)
    {
        (*buf->memory_error) (buf);
        return -2;
    }
    buf->data = data;
    buf->last = data;
    data->next = NULL;
    data->bufp = data->text;
    data->size = 0;
    if (want < BUFFER_DATA_SIZE)
        get = want;
    else
        get = BUFFER_DATA_SIZE;
    status = (*buf->input) (buf->closure, data->bufp, get,
                        BUFFER_DATA_SIZE, &nbytes);
    if (status != 0)
        return status;
    data->size = nbytes;
}
    retdata = buf->data->bufp;
if (want < buf->data->size)
{
  *got = want;
  buf->data->size -= want;
  buf->data->bufp += want;
}
else
{
  *got = buf->data->size;
  buf->data->size = 0;
}
    return 0;
}

/*
 * Copy lines from an input buffer to an output buffer.  This copies
 * all complete lines (characters up to a newline) from INBUF to
 * OUTBUF.  Each line in OUTBUF is preceded by the character COMMAND
 * and a space.
 * /
void copy_lines (outbuf, inbuf, command)
struct buffer *outbuf;
struct buffer *inbuf;
int command;
{
    while (1)
    {
        struct buffer_data *data;
        struct buffer_data *nldata;
        char *nl;
        int len;
        /* See if there is a newline in INBUF.  */
        nldata = NULL;
        nl = NULL;
        for (data = inbuf->data; data != NULL; data = data->next)
        {
            nl = memchr (data->bufp, '\n', data->size);
            if (nl != NULL)
            {
                nldata = data;
                break;
            }
        }
        if (data == NULL)
            break;
        if (nl != NULL)
        {
            data->size = nl - data;
            nldata = NULL;
        }
        else
        {
            *data = data->text;
            data->next = NULL;
            free_buffer_data = data;
            inbuf->data = data;
            data->last = data;
            data->bufp = data->text;
            data->size = data->next->size - data->size;
            data->next = data->next->next;
            inbuf->data = inbuf->data->next;
        }
    }
}
if (nldata == NULL) {
    /* There are no more lines in INBUF. */
    return;
}

/* Put in the command. */
bufappend(char (outbuf, command);
bufappend(char (outbuf, ' '));

if (inbuf->data != nldata)
{
    /* Simply move over all the buffers up to the one containing
     * the newline
     */
    for (data = inbuf->data; data->next != nldata; data = data->next)
    {
        data->next = NULL;
        bufappend(data (outbuf, inbuf->data, data);
        inbuf->data = nldata;
    }

    /* If the newline is at the very end of the buffer, just move
     * the buffer onto OUTBUF. Otherwise we must copy the data.
     */
    len = nl + 1 - nldata->bufp;
    if (len == nldata->size)
    {
        inbuf->data = nldata->next;
        if (inbuf->data == NULL)
        {
            nldata->next = NULL;
            bufappend(data (outbuf, nldata, nldata);
        }
        else
        {
            bufoutput(outbuf, nldata->bufp, len);
            nldata->bufp += len;
            nldata->size -= len;
        }
    }
}

/* Copy counted data from one buffer to another. The count is an
 integer, host size, host byte order (it is only used across a
 pipe). If there is enough data, it should be moved over. If there
 is not enough data, it should remain on the original buffer. A
 negative count is a special case. If one is seen, *SPECIAL is set
 to the (negative) count value and no additional data is gathered
 from the buffer; normally *SPECIAL is set to 0. This function
 returns the number of bytes it needs to see in order to actually
 copy something over.
 */

int bufcopy(counted (outbuf, inbuf, special)

struct buffer *outbuf;
struct buffer *inbuf;
int *special;
{
    *special = 0;
    while (1)
    {
        struct buffer_data *data;
        int need;
        union
        {
            char intbuf[sizeof (int)];
            int i;
        } u;
        char *intp;
        int count;
        struct buffer_data *start;
        int startoff;
        struct buffer_data *stop;
        int stopwant;

        /* See if we have enough bytes to figure out the count. */
        intp = u.intbuf;
        for (data = inbuf->data; data != NULL; data = data->next)
        {
            if (data->size >= need)
{  
  memcpy(intp, data->bufp, need);  
  break;  
}  
memcpy(intp, data->bufp, data->size);  
intp += data->size;  
need -= data->size;  
}  
if (data == NULL)  
{  
  /* We don't have enough bytes to form an integer. */  
  return need;  
}  
count = u.i;  
start = data;  
startoff = need;  
if (count < 0)  
{  
  /* A negative COUNT is a special case meaning that we  
  don't need any further information. */  
  stop = start;  
  stopwant = 0;  
}  
else  
{  
  /* We have an integer in COUNT. We have gotten all the  
  data from INBUF in all buffers before START, and we  
  have gotten STARTOFF bytes from START. See if we have  
  enough bytes remaining in INBUF. */  
  need = count - (start->size - startoff);  
  if (need <= 0)  
  {  
    stop = start;  
    stopwant = count;  
  }  
  else  
  {  
    for (data = start->next; data != NULL; data = data->next)  
    {  
      if (need <= data->size)  
        break;  
      need -= data->size;  
    }  
    if (data == NULL)  
    {  
      /* We don't have enough bytes. */  
      return need;  
    }  
    stop = data;  
    stopwant = need;  
  }  
  /* We have enough bytes. Free any buffers in INBUF before  
  START, and remove STARTOFF bytes from START, so that we can  
  forget about STARTOFF. */  
  start->bufp += startoff;  
  start->size -= startoff;  
  if (start->size == 0)  
    start = start->next;  
  if (stop->size == 0)  
    stop = stop->next;  
  stopwant = 0;  
}  
while (inbuf->data != start)  
{  
  data = inbuf->data;  
  inbuf->data = data->next;  
  data->next = free_buffer_data;  
  free_buffer_data = data;  
}  
/* If COUNT is negative, set *SPECIAL and get out now. */  
if (count < 0)  
{  
  *special = count;  
  return 0;  
}  
/*
* We want to copy over the bytes from START through STOP. We
  * only want STOPWANT bytes from STOP.
  */
  if (start != stop)
    
    /* Attach the buffers from START through STOP to OUTBUF. */
    for (data = start; data->next != stop; data = data->next)
      inbuf->data = stop;
    data->next = NULL;
  buf.append(data (outbuf, start, data);
  
  if (stopwant > 0)
    
    buf.output (outbuf, stop->bufp, stopwant);
    stop->bufp += stopwant;
    stop->size -= stopwant;
  }

  /* NOTREACHED*/
}

/* Shut down a buffer. This returns 0 on success, or an errno code. */
int
buf.shutdown (buf)
  struct buffer *buf;
  {
    buf->shutdown (buf);
    return (buf->shutdown) (buf->closure);
    return 0;
  }

/* The simplest type of buffer is one built on top of a stdio FILE. 
For simplicity, and because it is all that is required, we do not 
implement setting this type of buffer into nonblocking mode. The 
closure field is just a FILE * - - . */

static int stdio.buffer.input PROTO((void *, char *, int, int, int *));
static int stdio.buffer.output PROTO((void *, const char *, int, int *));
static int stdio.buffer.flush PROTO((void *));

/* Initialize a buffer built on a stdio FILE. */

struct buffer *stdio.buffer.initialize (fp, input, memory)
  FILE *fp;
  int input;
  
  void (*memory) PROTO((struct buffer *));
  
  return buf.initialize (input ? stdio.buffer.input : NULL,
    input ? NULL : stdio.buffer.output,
    input ? NULL : stdio.buffer.flush,
    (int *) PROTO((void *, int))))) NULL,
    (int *) PROTO((void *)) NULL,
    memory,
    (void *) fp);

/* The buffer input function for a buffer built on a stdio FILE. */

static int
stdio.buffer.input (closure, data, need, size, got)
  void *closure;
  char *data;
  int need;
  int size;
  int *got;
  
  {
    FILE *fp = (FILE *) closure;
    int nbytes;
    
    /* Since stdio does its own buffering, we don't worry about 
    getting more bytes than we need. */
    if (need == 0 || need == 1)
      
        int ch;
        ch = getc (fp);
        if (ch == EOF)
          
            if (feof (fp))
              return -1;
            else if (errno == 0)
              return EIO,
else
    return errno;

    data = ch;
    got = 1;
    return 0;
}

nbytes = fread (data, 1, need, fp);

if (nbytes == 0)
{
    got = 0;
    if (feof (fp))
        return -1;
    else if (errno == 0)
        return EIO;
    else
        return errno;
}

    got = nbytes;
    return 0;
}

/* The buffer output function for a buffer built on a stdio FILE. */

static int
stdio_buffer_output (closure, data, have, wrote)
{
    FILE *fp = (FILE *)closure;
    *wrote = 0;
    while (have > 0)
{
    int nbytes;
    nbytes = fwrite (data, 1, have, fp);
    if (nbytes != have)
{
        if (errno == 0)
            return EIO;
        else
            return errno;
    }
    *wrote += nbytes;
    have -= nbytes;
    data += nbytes;
}
    return 0;
}

/* The buffer flush function for a buffer built on a stdio FILE. */

static int
stdio_buffer_flush (closure)
{
    FILE *fp = (FILE *)closure;
    if (fflush (fp) != 0)
{
        if (errno == 0)
            return EIO;
        else
            return errno;
    }
    return 0;
}

/* Certain types of communication input and output data in packets,
   where each packet is translated in some fashion. The packetizing
   buffer type supports that, given a buffer which handles lower level
   I/O and a routine to translate the data in a packet.

   This code uses two bytes for the size of a packet, so packets are
   restricted to 65536 bytes in total.

   The translation functions should just translate; they may not
significant increase or decrease the amount of data. The actual size of the initial data is part of the translated data. The output translation routine may add up to PACKET_SLOP additional bytes, and the input translation routine should shrink the data correspondingly. */

#define PACKET_SLOP (100)

/* This structure is the closure field of a packetizing buffer. */

struct packetizing_buffer

{  /* The underlying buffer. */
    struct buffer *buf;
    / * The input translation function. Exactly one of inpfn and outfn will be NULL. The input translation function should untranslate the data in INPUT, storing the result in OUTPUT. SIZE is the amount of data in INPUT, and is also the size of OUTPUT. This should return 0 on success, or an errno code. */
    int (*inpfn) PROTO(void *fnclosure, const char *input, char *output, int size);
}

/* The output translation function. This should translate the data in INPUT, storing the result in OUTPUT. The first two bytes in INPUT will be the size of the data, and so will SIZE. This should set *TRANSLATED to the amount of translated data in OUTPUT. OUTPUT is large enough to hold SIZE + PACKET_SLOP bytes. This should return 0 on success, or an errno code. */
int (*outfn) PROTO(void *fnclosure, const char *input, char *output, int size, int *translated);

/* A closure for the translation function. */
void *fnclosure;

/* For an input buffer, we may have to buffer up data here. */
/* This is non-zero if the buffered data has been translated. Otherwise, the buffered data has not been translated, and starts with the two byte packet size. */
int translated;
/* The amount of buffered data. */
int holdsize;
/* The buffer allocated to hold the data. */
char *holdbuf;
/* The size of holdbuf. */
int holdbufsize;
/* If translated is set, we need another data pointer to track where we are in holdbuf. If translated is clear, then this pointer is not used. */
char *holddata;
};

static int packetizing_buffer_input PROTO((void *, char *, int, int, int *));
static int packetizing_buffer_output PROTO((void *, const char *, char *, int, int *));
static int packetizing_buffer_flush PROTO((void *));
static int packetizing_buffer_block PROTO((void *, int));
static int packetizing_buffer_shutdown PROTO((void *));

/* Create a packetizing buffer. */

struct buffer *packetizing_buffer_initialize (buf, inpfn, outfn, fnclosure, memory)

struct buffer *buf;
int (*inpfn) PROTO((void *, const char *, char *, int));
int (*outfn) PROTO((void *, const char *, char *, int, int *));

void *fnclosure;
void (*memory) PROTO((struct buffer *));

{  struct packetizing_buffer *pb;
    pb = (struct packetizing_buffer *) malloc (sizeof *pb);
    memset (pb, 0, sizeof *pb);
    pb->buf = buf;
    pb->inpfn = inpfn;
    pb->outfn = outfn;
    pb->fnclosure = fnclosure;

    if (inpfn != NULL)
    {
        /* Add PACKET_SLOP to handle larger translated packets, and add 2 for the count. This buffer is increased if necessary. */
        pb->holdbufsize = BUFFER_DATA_SIZE + PACKET_SLOP + 2;
        pb->holdbuf = (char *) malloc (pb->holdbufsize);
    }

    return buf_initialize (inpfn != NULL ? packetizing_buffer_input : NULL,
                            inpfn != NULL ? packetizing_buffer_output, inpfn != NULL ? packetizing_buffer_flush, packetizing_buffer_block, packetizing_buffer_shutdown, memory, pb);
}
1440 /* Input data from a packetizing buffer. */

static int
packetizing_buffer_input (closure, data, need, size, got)

    void *closure;
    char *data;
    int need;
    int size;
    int *got;

    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;
    *got = 0;

    if (pb->holdsize > 0 && pb->translated) {
        int copy;
        copy = pb->holdsize;

        if (copy > size) {
            memcpy (data, pb->holddata, size);
            pb->holdsize -= size;
            *got = size;
            return 0;
        }

        memcpy (data, pb->holddata, copy);
        pb->holdsize = 0;
        pb->translated = 0;

        data += copy;
        need -= copy;
        size -= copy;
        *got = copy;
    }

    while (need > 0 || *got == 0) {
        int get, status, nread, count, tcount;
        char *bytes;
        char stackoutbuf[BUFFER_DATA_SIZE + PACKET_SLOP];
        char *inbuf, *outbuf;

        /* If we don't already have the two byte count, get it. */
        if (pb->holdsize < 2) {
            get = 2 - pb->holdsize;
            status = buf_read_data (pb->buf, get, &bytes, &nread);
            if (status == 0) {
                /* buf_read_data can return -3, but a buffer input
                 * function is only supposed to return -1, 0, or an
                 * error code. */
                if (status == -2)
                    status = ENOMEM;
                return status;
            }

            pb->holdbuf[1] = bytes[0];
            else
                pb->holdbuf[0] = bytes[0];
                if (nread < 2) {
                    /* We only got one byte, but we needed two. Stash
                     * the byte we got, and try again. */
                    pb->holdsize = 1;
                    continue;
                }

            pb->holdbuf[1] = bytes[1];
            pb->holdsize = 2;
        }

        /* Read the packet. */
        count = (((int)pb->holdbuf[0] & 0xff) << 8)
+ (pb->holdbuf[1] & 0xff));

if (count + 2 > pb->holdbufsize)
{
    char *n;
    /* We didn’t allocate enough space in the initialize
     * function. */
    n = realloc(pb->holdbuf, count + 2);
    if (n == NULL)
    {
        (pb->buf->memory_error) (pb->buf);
        return ENOMEM;
    }
    pb->holdbuf = n;
    pb->holdbufsize = count + 2;
}

get = count = (pb->holdsize - 2);

status = buf_readdata (pb->buf, get, &bytes, &nread);
if (status != 0)
{
    /* buf_readdata can return -2, but a buffer input
     * function is only supposed to return -1, 0, or an error
     * code. */
    if (status == -2)
        status = ENOMEM;
    return status;
}

if (nread == 0)
{
    /* We did not get any data. Presumably the buffer is in
     * nonblocking mode. */
    return 0;
}

if (nread < get)
{
    /* We did not get all the data we need to fill the packet.
     * buf_readdata does not promise to return all the bytes
     * requested, so we must try again. */
    memcpy (pb->holdbuf + pb->holdsize, bytes, nread);
    pb->holdsize += nread;
    continue;
}

/* We have a complete untranslated packet of COUNT bytes. */

if (pb->holdsize == 2)
{
    /* We just read the entire packet (the 2 bytes in
     * PB->HOLDBUF are the size). Save a memcpy by
     * translating directly from BYTES. */
    inbuf = bytes;
}
else
{
    /* We already had a partial packet in PB->HOLDBUF. We
     * need to copy the new data over to make the input
     * contiguous. */
    memcpy (pb->holdbuf + pb->holdsize, bytes, nread);
    inbuf = pb->holdbuf + 2;
}

if (count <= sizeof stackoutbuf)
    outbuf = stackoutbuf;
else
{
    outbuf = malloc (count);
    if (outbuf == NULL)
    {
        (pb->buf->memory_error) (pb->buf);
        return ENOMEM;
    }
}

status = (*pb->inpfn) (pb->fnclosure, inbuf, outbuf, count);
if (status != 0)
    return status;

/* The first two bytes in the translated buffer are the real
 * length of the translated data. */
tcount = ((outbuf[0] & 0xff) << 8) + (outbuf[1] & 0xff);

if (tcount > count)
    error (1, 0, "Input translation failure");
if (tcount > size)
{
    /* We have more data than the caller has provided space
       for. We need to save some of it for the next call. */
    memcpy(data, outbuf + 2, size);
    *got += size;
    pb->holdsize = tcount - size;
    memcpy(pb->holdbuf, outbuf + 2 + size, tcount - size);
    pb->holddata = pb->holdbuf;
    pb->translated = 1;
    if (outbuf != stackoutbuf)
        free(outbuf);
    return 0;
}
memcopy(data, outbuf + 2, tcount);

if (outbuf != stackoutbuf)
    free(outbuf);
pb->holdsize = 0;
data += tcount;
need -= tcount;
size -= tcount;
*got += tcount;

return 0;

/* Output data to a packetizing buffer. */

static int
packetizing_buffer_output(closure, data, have, wrote)
    void *closure;
    const char *data;
    int have;
    int *wrote;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;
    char inbuf[BUFFER_DATA_SIZE + 2];
    char stack_outbuf[BUFFER_DATA_SIZE + PACKET_SLOP + 4];
    struct buffer_data *outdata;
    char *outbuf;
    int size, status, translated;

    if (have > BUFFER_DATA_SIZE)
    {
        /* It would be easy to malloc a buffer, but I don't think this
           case can ever arise. */
        abort();
    }
    inbuf[0] = (have >> 8) & 0xff;
    inbuf[1] = have & 0xff;
    memcpy(inbuf + 2, data, have);

    size = have + 2;
    /* The output function is permitted to add up to PACKET_SLOP
       bytes, and we need 2 bytes for the size of the translated data.
       If we can guarantee that the result will fit in a buffer.data,
       we translate directly into one to avoid a memcpy in buf_output. */
    if (size + PACKET_SLOP + 2 > BUFFER_DATA_SIZE)
        outbuf = stack_outbuf;
    else
    {
        outdata = get_buffer_data();
        if (outdata == NULL)
        {
            (*pb->buf->memory_error)(pb->buf);
            returnENOMEM;
        }
        outdata->next = NULL;
        outdata->bufp = outdata->text;
    }
    outbuf = outdata->text;

    status = (*pb->outfn)(pb->fnclosure, inbuf, outbuf + 2, size,
        &translated);
    if (status != 0)
        return status;
/* The output function is permitted to add up to PACKET_SLOP bytes. */
if (translated > size + PACKET_SLOP)
    abort();
outbuf[0] = (translated >> 8) & 0xff;
outbuf[1] = translated & 0xff;
if (outbuf == stack_outbuf)
    buf_output (pb->buf, outbuf, translated + 2);
else
    {
        outdata->size = translated + 2;
        buf_append_data (pb->buf, outdata, outdata);
    }
*wrote = have;
/* We will only be here because buf_output was called on
the packetizing buffer. That means that we should now call
buf_end_output on the underlying buffer. */
return buf_end_output (pb->buf);
/* Flush data to a packetizing buffer. */
static int
packetizing_buffer_flush (closure)
void *closure;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;
    /* Flush the underlying buffer. Note that if the original call to
buf/flush passed 1 for the BLOCK argument, then the buffer will
already have been set into blocking mode, so we should always
pass 0 here. */
    return buf_flush (pb->buf, 0);
}
/* The block routine for a packetizing buffer. */
static int
packetizing_buffer_block (closure, block)
void *closure;
int block;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;
    if (block)
        return set_block (pb->buf);
    else
        return set_nonblock (pb->buf);
}
/* Shut down a packetizing buffer. */
static int
packetizing_buffer_shutdown (closure)
void *closure;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;
    return buf_shutdown (pb->buf);
}
endif /* defined (SERVER_SUPPORT) | | defined (CLIENT_SUPPORT) */
A.4 buffer.h

/* Declarations concerning the buffer data structure. */

#ifdef (SERVER_SUPPORT) || defined (CLIENT_SUPPORT)

/*
 * We must read data from a child process and send it across the
 * network. We do not want to block on writing to the network, so we
 * store the data from the child process in memory. A BUFFER
 * structure holds the status of one communication, and uses a linked
 * list of buffer data structures to hold data.
 */

struct buffer
{
    /* Data. */
    struct buffer_data *data;

    /* Last buffer on data chain. */
    struct buffer_data *last;

    /* Nonzero if the buffer is in nonblocking mode. */
    int nonblocking;

    /* Functions must be provided to transfer data in and out of the
     * buffer. Either the input or output field must be set, but not
     * both. */

    /* Read data into the buffer DATA. There is room for up to SIZE
     * bytes. In blocking mode, wait until some input, at least NEED
     * bytes, is available (NEED may be 0 but that is the same as NEED
     * == 1). In non-blocking mode return immediately no matter how
     * much input is available; NEED is ignored. Return 0 on success,
     * or -1 on end of file, or an errno code. Set the number of
     * bytes read in *GOT.
     * If there are a nonzero number of bytes available, less than NEED,
     * followed by end of file, just read those bytes and return 0. */
    int (*input) (void *closure, char *data, int need, int size,
                  int *got);

    /* Write data. This should write up to HAVE bytes from DATA.
     * This should return 0 on success, or an errno code. It should
     * set the number of bytes written in *WROTE. */
    int (*output) (void *closure, const char *data, int have,
                   int *wrote);

    /* Flush any data which may be buffered up after previous calls to
     * OUTPUT. This should return 0 on success, or an errno code. */
    int (*flush) (void *closure);

    /* Change the blocking mode of the underlying communication
     * stream. If BLOCK is non-zero, it should be placed into
     * blocking mode. Otherwise, it should be placed into
     * non-blocking mode. This should return 0 on success, or an
     * errno code. */
    int (*block) (void *closure, int block);

    /* Shut down the communication stream. This does not mean that it
     * should be closed. It merely means that no more data will be
     * read or written, and that any final processing that is
     * appropriate should be done at this point. This may be NULL.
     * It should return 0 on success, or an errno code. This entry
     * point exists for the compression code. */
    int (*shutdown) (void *closure);

    /* This field is passed to the INPUT, OUTPUT, and BLOCK functions. */
    void *closure;

    /* Function to call if we can't allocate memory. */
    void (*memory_error) (struct buffer *);
};

/* Data is stored in lists of these structures. */

struct buffer_data
{
    /* Next buffer in linked list. */
    struct buffer_data *next;

    /* A pointer into the data area pointed to by the text field. This
     * is where to find data that has not yet been written out.
     */
    char *bufp;

    /* The number of data bytes found at BUFP. */
    int size;

*/
```c
/*
 * Actual buffer. This never changes after the structure is
 * allocated. The buffer is BUFFER_DATA_SIZE bytes.
 */
char *text;
}

/* The size we allocate for each buffer_data structure. */
#define BUFFER_DATA_SIZE (4096)

/* The type of a function passed as a memory error handler. */
typedef void (*BUFMEMERRPROC) (struct buffer *);

extern struct buffer *buf_initialize();
extern struct buffer *buf_nonio_initialize (void (*) (struct buffer *));
extern struct buffer *stdio_buffer_initialize
    (FILE *, int, void (*) (struct buffer *));
extern struct buffer *compress_buffer_initialize
    (struct buffer *, int, int, void (*) (struct buffer *));
extern struct buffer *packetizing_buffer_initialize
    (struct buffer *, int (*) (void *, const char *, char *, int),
     int (*) (void *, const char *, char *, int, int *),
     void (*) (struct buffer *),
     void *);

extern void buf_free (struct buffer *);
extern void buf_output (struct buffer *, const char *, int);
extern void buf_output0 (struct buffer *, const char *, int);
extern void buf_append_char (struct buffer *, int);
extern void buf_flush (struct buffer *, int);
extern void buf_nonblock (struct buffer *, int);
extern void buf_send_counted (struct buffer *, int);

extern void buf_send_special_count (struct buffer *, int);
extern void buf_send_append_data (struct buffer *,
                                  struct buffer_data *);
extern void buf_append_buffer (struct buffer *, struct buffer *);
extern int buf_read_file (FILE *, long, struct buffer_data *,
                          struct buffer_data *);
extern int buf_read_file_line (FILE *, struct buffer_data *,
                              struct buffer_data *);
extern int buf_input (struct buffer *, int *);
extern int buf_read_line (struct buffer *, char **, int *);
extern int buf_read_data (struct buffer *, int, char **, int *);
extern void buf_copy_lines (struct buffer *, struct buffer *, int);
extern int buf_copy_counted (struct buffer *, struct buffer *, int *);
extern int buf_length (struct buffer *,
                        struct buffer_data *);
extern int buf_shutdown (struct buffer *);

ifdef SERVER_FLOWCONTROL
external int buf_count_lines (struct buffer *);
endif

ifdef SERVER_FLOWCONTROL */
#endif
define (SERVER_SUPPORT) || define (CLIENT_SUPPORT) */
```
A.5 checkin.c

/ * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Check In
 * Does a very careful checkin of the file “user”, and tries not to spoil its
 * modification time (to avoid needless recomputations). When RCS ID keywords
 * get expanded on checkout, however, the modification time is updated and
 * there is no good way to get around this.
 * Returns non-zero on error.
 */

#include "cvs.h"
#include "fileattr.h"
#include "edit.h"

int
checkin(type, finfo, rcs, rev, tag, options, message)
int type;
struct file_info *finfo;
char *rcs;
char *rev;
char *tag;
char *options;
char *message;

// Hmm. This message goes to stdout and the “foo,v < foo”
// message from “ci” goes to stderr. This doesn’t make a whole
// lot of sense, but doing everything go to stdout can only be
// gracefully achieved once RCS/checkin is librified. */
cvs_output("Checking in ", 0);
cvs_output(finfo->fullname, 0);
cvs_output("Checkin ", 0);
tocvsPath = wrap_tocvs_process_file (finfo->file);
if (finfoex) {
    if (tocvsPath)
        if (unlink_file_dir (finfo->file) < 0)
            error (1, errno, "cannot remove \%s", info->fullname);
        rename_file (tocvsPath, info->file);
    }
if (finfo->rcs == NULL)
    finfo->rcs = RCS_parse (info->file, info->repository);
switch (RCS_checkin (finfo->rcs, NULL, message, rev, RCS_FLAGS(KEEPFILE)))
{
case 0:
    /* everything normal */
    /* The checkin succeeded. If checking the file out again
    would not cause any changes, we are done. Otherwise,
    we need to check out the file, which will change the
    modification time of the file.
    The only way checking out the file could cause any
    changes is if the file contains RCS keywords. So we if
    we are not expanding RCS keywords, we are done. */
    if (strcmp (options, "-V4") == 0) /* upgrade to V5 now */
        options[0] = 'V';
    /* FIXME: If PreservePermissions is on, RCS/cmp file is
    going to call RCS/checkout into a temporary file
    anyhow. In that case, it would be more efficient to
    call RCS/checkout here, compare the resulting files
    using cmp, and rename if necessary. I think this
    should be fixed in RCS/cmp file. */
    if (!preserveperms
        && options != NULL
        && (strcmp (options, "-kb") == 0
            || strcmp (options, "-ko") == 0)
            || RCScmpfile (finfo->rcs, rev, options, finfo->file) == 0)
        {
            /* The existing file is correct. We don’t have to do
            anything. */
        }
set_time = 0;
}
else
{
  /* The existing file is incorrect. We need to check
     out the correct file contents. */
  if (RCS_checkout (finfo->rcs, finfo->file, rev, (char *) NULL,
                  options, RUNTTY, (RCSCHECKOUTPROC) NULL, (void *) NULL) != 0)
    error (1, 0, "failed when checking out new copy of %s",
           finfo->fullname);
  xchmod (finfo->file, 1);
  set_time = 1;
}

wrap_fromcvs_processfile (finfo->file);

/*
 * If we want read-only files, muck the permissions here, before
 * getting the file time-stamp.
 */

if (fcvwrite || fileattrget (finfo->file, "_watched"))
  xchmod (finfo->file, 0);

/* Re-register with the new data. */
if (strcmp (vers->options, "-vR") == 0)
  vers->options[0] = 'W';
Register (finfo->entries, finfo->file, vers->vn_recs, vers->ts_user,
       vers->options, vers->tag, vers->date, (char *) 0, CVSroot_directory,
       finfo->repository);
history_write (type, NULL, vers->vn_recs,
               finfo->file, finfo->repository);

if (tocvsPath)
  if (unlink (tocvsPath) < 0)
    error (0, errno, "cannot remove %s", tocvsPath);
  break;

case -1:
  /* fork failed */
if (tocvsPath)
  if (unlink (tocvsPath) < 0)
    error (0, errno, "cannot remove %s", tocvsPath);
  if (noexec)
    error (1, errno, "could not check in %s -- fork failed",
           finfo->fullname);
  return (1);

default:
  /* csi failed */

  /* The checkin failed, for some unknown reason, so we
   print an error, and return an error. We assume that
   the original file has not been touched. */
  if (tocvsPath)
    if (unlink (tocvsPath) < 0)
      error (0, errno, "cannot remove %s", tocvsPath);
    if (noexec)
      error (0, 0, "could not check in %s", finfo->fullname);
    return (1);

  }

  /* When checking in a specific revision, we may have locked the wrong
   * branch, so to be sure, we do an extra unlock here before
   * returning. */
  if (rev)
    { (void) RCS_unlock (finfo->rcs, NULL, 1);
      RCS_rewrite (finfo->rcs, NULL, NULL);
    }
#endif SERVER_SUPPORT
if (server_active)
{
  if (set_time)
    /* Need to update the checked out file on the client side. */
  server_updated (finfo, vers, SERVERUPDATED,
                 (mod_vn) - 1, (unsigned char *) NULL,
                 (struct buffer *) NULL);
  else
    server_checked_in (finfo->file, finfo->update_dir, finfo->repository);
  else
    sendif mark_update (finfo->file);
  freevers (&vers);
    return (0);
}
A.6 checkout.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Create Version
 * "checkout" creates a "version" of an RCS repository. This version is owned
 * totally by the user and is actually an independent copy, to be dealt with
 * as seen fit. Once "checkout" has been called in a given directory, it
 * never needs to be called again. The user can keep up-to-date by calling
 * "update" when he feels like it; this will supply him with a merge of his
 * own modifications and the changes made in the RCS original. See "update"
 * for details.
 * "checkout" can be given a list of directories or files to be updated and in
 * the case of a directory, will recursively create any sub-directories that
 * exist in the repository.
 * When the user is satisfied with his own modifications, the present version
 * can be committed by "commit"; this keeps the present version in tact,
 * usually.
 * The call is cvs checkout [options] <module-name>...
 * "checkout" creates a directory in CVS, in which it keeps its administration,
 * in two files, Repository and Entries. The first contains the name of the
 * repository. The second contains one line for each registered file,
 * consisting of the version number it derives from, its time stamp at
 * derivation time and its name. Both files are normal files and can be
 * edited by the user, if necessary (when the repository is moved, e.g.)
 */
#include "assert.h"
#include "cvs.h"

static char *findlash PROTOTO(char *start, char *p);

static int checkout_proc PROTOTO(int *pargc, char ***argv, char *where, char *where, char *module, int localspecified, char *module, char *module);

static int sub_location PROTOTO(void);

static const char *const checkout_usage[] = {
    "Usage: %s %s [options] <module-name>
          [-j rev] [-j rev2] [-k kopt] modules...
    -r rev
    "[-j rev1] [-j rev2] [-k kopt] modules..."
    
    "-c	"cat" the module database.
    "-C	Create Version
    "-D date	Export revisions as of date.
    "-r rev	Export revision or tag.
    "-n	Do not run module program (if any).
    "-R	Process directories recursively (default).
    "-d loc	Export into dir instead of module name.
    "-D date	Export revisions as of date.
    "-f	Force a head revision match if tag/date not found.
    "-j rev	Merge in changes made between current revision and rev.
    "-k kopt	Use RCS kopt -k option on checkout.
    "-N	Don't shorten module paths if -d specified.
    "-P	Prune empty directories.
    "-R	Process directories recursively.
    "-s	Like -c, but include module status.
    "-t	Local directory only, not recursive.
    "-v
    "-f	Force a head revision match if tag/date not found.
    "-j rev	Merge in changes made between current revision and rev.
    "[Specify the --help global option for a list of other help options]"
    NULL
};

static const char *const export_usage[] = {
    "Usage: %s %s [options] <module-name>
          [-j rev] [-j rev2] [-k kopt] modules...
    -r rev
    "[-j rev1] [-j rev2] [-k kopt] modules..."
    
    "-c	"cat" the module database.
    "-C	Create Version
    "-D date	Export revisions as of date.
    "-r rev	Export revision or tag.
    "-n	Do not run module program (if any).
    "-R	Process directories recursively (default).
    "-d loc	Export into dir instead of module name.
    "-D date	Export revisions as of date.
    "-f	Force a head revision match if tag/date not found.
    "-j rev	Merge in changes made between current revision and rev.
    "[Specify the --help global option for a list of other help options]"
    NULL
};

static int checkout_prune.isdir;
static int forcechecktagmatch = 1;
static int pipeout;
static int aflag;
static char *options = NULL;
static char *tag = NULL;
static int tag_validated = 0;
static char *date = NULL;
static char *join_rev1 = NULL;
static char *join_rev2 = NULL;
static int join_tags_validated = 0;
static char *preload_update_dir = NULL;
static char *history_name = NULL;

int checkout(int argc, char **argv)
{
    int i;
    int c;
    DBM *db;
    int cat = 0, err = 0, status = 0;
    int run_module_prog = 1;
    int local = 0;

    int shorten = -1;
    char *where = NULL;
    char *valid_options;
    const char *const *valid_usage;

    enum mtype m_type;

    /*
     * A smaller subset of options are allowed for the export command, which
     * is essentially like checkout, except that it hard-codes certain
     * options to be default (like -k) and takes care to remove the CVS
     * directory when it has done its duty
     */
    if (strcmp(command_name, "export") == 0)
    {
        m_type = EXPORT;
        valid_options = "+Nnk:d:flRQqr:D:1;";
        valid_usage = export_usage;
    }
    else
    {
        m_type = CHECKOUT;
        valid_options = "+Nnk:d:flpqr:Drj:j:P:2;";
        valid_usage = checkout_usage;
    }

    if (argc == -1)
        usage(valid_usage);

    ign_setup();
    wrap_setup();

    optind = 0;
    while ((c = getopt(argc, argv, valid_options)) != -1)
    {
        switch (c)
        {
        case 'A':
            aflag = 1;
            break;
        case 'N':
            shorten = 0;
            break;
        case 'k':
            if (options)
                free(options);
            options = RCS_checkkflag(optarg);
            break;
        case 'n':
            run_module_prog = 0;
            break;
        case 'Q':
            break;
        case 'q':
            break;
        case 'l':
            local = 1;
            break;
        case 'R':
            local = 0;
            break;
        case 'P':
            checkout_prune_dirs = 1;
            break;
        #ifdef SERVER_SUPPORT
        /* The CVS 1.5 client sends these options (in addition to
         * Global option requests), so we must ignore them. */
        #endif
        if (server_active)
        {
            error(1, 0,
                "-q or -Q must be specified before "
                command_name);
            break;
        }
        case '1':
            local = 1;
            break;
        case '8':
            local = 0;
            break;
        case 'P':
            checkout_prune_dirs = 1;
            break;
        case 'Q':
            break;
        case 'q':
            break;
        #endif
    }
break;
case 'p':
pipout = 1;
run_module = prog = 0; /* don't run module prog when piping */
noexec = 1; /* so no locks will be created */
break;
case 'c':
cat = 1;
break;
case 'd':
where = optarg;
break;
if (shorten == -1)
    shorten = 1;
case 's':
    status = 1;
    break;
case 'f':
    force_tag_match = 0;
    break;
case 'r':
tag = optarg;
    checkout_prune_dirs = 1;
    break;
case 'D':
date = Make_Date(optarg);
    checkout_prune_dirs = 1;
    break;
case 'j':
    if (join_rev2)
        error(1, 0, "only two -j options can be specified");
    if (join_rev1)
        join_rev2 = optarg;
    else
        join_rev1 = optarg;
break;
case '?':
default:
    usage (valid_usage);
break;
}
} 
argc -= optind;
argv += optind;
if (shorten == -1)
    shorten = 0;
if ((cat || status) && argc == 0)
    error(1, 0, "-c and -s must not get any arguments");
if (! (cat || status) && argc == 0)
    error(1, 0, "must specify at least one module or directory");
if (where && pipout)
    error(1, 0, "-d and -p are mutually exclusive");
if (strcmp (command_name, "export") == 0)
    
    if (tag && date)
        error(1, 0, "must specify a tag or date");
if (tag && isdigit (tag[0]))
    error(1, 0, "tag 's' must be a symbolic tag", tag);
#endif
if (server_active && where != NULL)
    {
        server_pathname = check (where);
    }
#endif
if (! safe_location())
    { 
        error(1, 0, "Cannot check out files into the repository itself");
    }
#endif
if (client_active)
    {
        start_server();
        ign_setup();
        }
        
        /* We have to expand names here because the "expand-modules" directive to the server has the side-effect of having the server send the check-in and update programs for the various modules/dirs requested. If we turn this off and
simply request the names of the modules and directories (as below in \texttt{expand-modules}), those files (CVS/\texttt{Checkout} or CVS/\texttt{Updateprog}) don't get created. \texttt{Grr!} */

```c
expand-modules = (\texttt{cat \&\& (status \&\& !pipeout
\&\& supported)} \texttt{\&\& request ("expand-modules")});

if (expand-modules)
  { /* This is done here because we need to read responses
      from the server before we send the command checkout or
      export files. */

    client\_expand\_modules (argc, argv, local);
  }

if (\texttt{run\_module\_prog})
  send\_arg (\texttt{"-n"});
if (local)
  send\_arg (\texttt{"-l"});
if (pipeout)
  send\_arg (\texttt{"-p"});
if (\texttt{force\_tag\_match})
  send\_arg (\texttt{"-t"});
if (\texttt{alltag})
  send\_arg (\texttt{"-a"});
if (\texttt{shorten})
  send\_arg (\texttt{"-s"});
if (\texttt{check\_change\_dirs \&\& \texttt{strcmp (command\_name, "expert") \(! = 0}})
  send\_arg (\texttt{\"-p\"});

client\_change\_dirs = checkout\_change\_dirs;

if (\texttt{cat})
  send\_arg (\texttt{"-c"});
if (where \(! = NULL)
  option\_with\_arg (\texttt{"-\&", where});
if (status)
  send\_arg (\texttt{"-d"});
if (options \(! = NULL \&\& options[0] \(! = \\"\\"\")
  send\_arg (options);
  option\_with\_arg (\texttt{"-c", tag});
if (\texttt{date})
  client\_send\_date (date);
if (\texttt{join\_rev1} \(! = NULL)
  option\_with\_arg (\texttt{"-j", join\_rev1});
if (\texttt{join\_rev2} \(! = NULL)
  option\_with\_arg (\texttt{"-j", join\_rev2});

if (expand\_modules)
  { client\_send\_expansions (local, where, 1);
  }
else
  { int i;
    for (i = 0; i < argc; \texttt{++i})
      send\_arg (argv[i]);
    client\_nonexpanded\_setup ();

    send\_\_server (\texttt{strcmp (command\_name, "expert") \(! = 0} ?
      \texttt{"export\_\_012": \"co\_\_012\", 0);};
  }

  return get\_responses\_and\_lose ();
  }

send\_\_df /\texttt{CLIENT\_\_SUPPORT} */

if (\texttt{cat \| status})
  { cat\_module (status);
    if (options)
      free (options);
    return (0);
  }

db = open\_module ();

/* If we've specified something like \texttt{\"cos co foo/bar baz/quux\"}
   don't try to shorten names. There are a few cases in which we
   could shorten (e.g. \texttt{\"cos co foo/bar baz\"}), but we don't
   handle those yet. Better to have an extra directory created
   than the thing checked out under the wrong directory name. */

if (\texttt{argc > 1})
  shorten = 0;

/* If we will be calling \texttt{history\_write}, work out the name to pass
it. */
if (\texttt{strcmp (command\_name, "expert") \(! = 0 \&\& !pipeout})
360  
361  if (tag & date)  
362     {  
363       sprintf(history_name, "%s:", tag, date);  
364     }  
365  else if (tag)  
366     history_name = tag;  
367  else  
368     history_name = date;  
369    
370  for (i = 0; i < argc; i++)  
371     err += do_module(db, argv[i], mtype, "Updating", checkout, proc,  
372                   where, shorten, local, run_module, prog,  
373                   (char *) NULL);  
374  close_module(db);  
375  if (options)  
376    free (options);  
377  return (err);  
378  
379 static int  
380 safe_location ()  
381  {  
382    char *current;  
383    char hardpath[PATH_MAX+5];  
384    size_t hardpath_len;  
385    int x;  
386    int retval;  
387  
388  ifdef HAVE_READLINK  
389     /* FIXME-arbitrary limit: should be retrying this like xgetwd.  
390      But how does readlink let us know that the buffer was too small?  
391      (by returning sizeoff hardpath - 1?). */  
392     x = readlink(CVSroot directory, hardpath, sizeoff hardpath - 1);  
393     
394     else  
395     
396      x = -1;  
397  endif  
398  
399  if (x == -1)  
400     {  
401       strcpy(hardpath, CVSroot directory);  
402     }  
403  else  
404     {  
405       hardpath[x] = '\0';  
406  
407       current = xgetwd ();  
408     
409     
410     }  
411  if (current == NULL)  
412     error (1, errno, "could not get working directory");  
413  
414  hardpath_len = strlen (hardpath);  
415  if ((strlen (current) >= hardpath_len && strncmp (current, hardpath, hardpath_len) == 0)  
416     {  
417       /* Current is a subdirectory of hardpath. */  
418       current [hardpath_len] = '/'  
419     }  
420  else  
421     {  
422       /* Current is hardpath itself. */  
423       current [hardpath_len] = '\0';  
424       retval = 0;  
425     }  
426  if (current[0] == '/')  
427     {  
428       /* It isn't a problem. For example, current is  
429       "foo/cvsroot-bar" and hardpath is "foo/cvsroot". */  
430       retval = 1;  
431     }  
432     else  
433     
434     
435     retval = 1;  
436     free (current);  
437     return retval;  
438   
439 struct dir* build  
440  {  
441     /* What to put in CVS/Repository. */  
442     char *repository;  
443     /* The path to the directory. */  
444     char *dirdir;  
445     
446     /* If set, don't build the directory, just change to it.  
447      The caller will also want to set REPOSITORY to NULL. */  
448     int just_chdir;  
449   
450   struct dir* next;  
451  };  
452  static int build() and chdir PROTO ((struct dir * and dir PROTO ((char *, char *, int)));  
453  static void build() PROTO ((char *, char *, int));
static void
build_one_dir (repository, dirpath, sticky)
{
    FILE *fp;
    if (lseek (CVSADM) &amp; stremap (command, "export") != 0)
        error (1, 0, "there is no repository %s", repository);
    if (++dirpath == dirpath)
        error (1, 0, "there is no repository %s", repository);
    if (CreateAdmin [".", dirpath, repository, sticky ? (char *) NULL : tag, sticky ? (char *) NULL : date,
        /* FIXME? This is a guess. If it is important
        for nonbranch to be set correctly here I
        think we need to write it one way now and
        then rewrite it later via WriteTag, once
        we've had a chance to call RCS_nodeisbranch
        on each file. */
        0, 1))
        return;
    if (!close (fp) == EOP)
        error (1, errno, "cannot close %s", CVSADMIN_ENTSTAT);
    ifdef SERVER_SUPPORT
        server_set_entstat (dirpath, repository);
    endif
}

/* process_module calls us back here so we do the actual checkout stuff
*/
/* ARGUSED */
static int
checkout_proc (pargc, argv, where_orig, mwhere, mfile, shorten,
        local_specified, omodule, msg)
{
    int *pargc;
    char **argv;
    char *where_orig;
    char *mwhere;
    char *mfile;
    int shorten;
    int local_specified;
    char *omodule;
    char *msg;
{
    int err = 0;
    int which;
    char *cp;
    char *repository;
    char *oldupdate = NULL;
    char *where;

    /* OK, so we're doing the checkout! Our args are as follows:
    * argc, argv contain either dir or dir followed by a list of files
    * where contains where to put it (if supplied by checkout)
    * mwhere contains the module name or -d from module file
    * mfile says do only that part of the module
    * shorten = 1 says shorten as much as possible
    * omodule is the original arg to do_module()
    */

    /* Set up the repository (maybe) for the bottom directory.
    * Allocate more space than we need so we don't need to keep
    * reallocating this string. */
    repository = xmalloc (strlen (CVSroot_directory) +
        strlen (argv[0]) +
        strlend (NULL ? 0 : strlend (mfile)) +
        10);
    (void) sprintf (repository, "%s/%s", CVSroot_directory, argv[0]);
    Sanitize_Repository_Name (repository);

    /* save the original value of preload/update/dir */
    if (preload/update/dir != NULL)
        oldupdate = xstrdup (preload/update/dir);
/* Allocate space and set up the where variable. We allocate more space than necessary here so that we don't have to keep reallocating it later on. */

where = xmalloc(strlen(argv[0])
+ (mfile == NULL ? 0 : strlen(mfile))
+ (mwhere == NULL ? 0 : strlen(mwhere))
+ (where_orig == NULL ? 0 : strlen(where_orig))
+ 10);

/* Yes, this could be written in a less verbose way, but in this form it is quite easy to read.

FIXME? The following code that sets should probably be moved to do_module in modules.c, since there is similar code in patch.c and ring.c. */

if (shorten)
{
  if (where_orig != NULL)
  {
    /* If the user has specified a directory with -d on the command line, use it preferentially, even over the -d flag in the modules file. */
    (void) strcpy(where, where_orig);
  } else if (mwhere != NULL)
  {
    /* Second preference is the value of mwhere, which is from the -d flag in the modules file. */
    (void) strcpy(where, mwhere);
  } else
  {
    /* Third preference is the directory specified in argv[0] which is this module's directory in the repository. */
    (void) strcpy(where, argv[0]);
  }
}
else
{
  /* Use the same preferences here, but don't shorten — that is, tack on where_orig if it exists. */
  *where = '\0';
  if (where_orig != NULL)
  {
    (void) strcat(where, where_orig);
    (void) strcat(where, "/");
  }
  /* If the -d flag in the modules file specified an absolute directory, let the user override it with the command-line -d option. */
  if ((mwhere != NULL) && (!isabsolute(mwhere)))
  {
    (void) strcat(where, mwhere);
  } else
  {
    (void) strcat(where, argv[0]);
  }
  strip_trailing_slashes(where); /* necessary? */

  /* At this point, the user may have asked for a single file or directory from within a module. In that case, we should modify where, repository, and argv as appropriate. */

  if (mfile != NULL)
  {
    /* The mfile variable can have one or more path elements. If it has multiple elements, we want to tack those onto both repository and where. The last element may refer to either a file or directory. Here's what to do:
      it refers to a directory
        -> simply tack it on to where and repository
      it refers to a file
        -> munge argv to contain 'basename mfile' */

    char *cp;
    char *path;

    /* Paranoia check. */
if (mfile[strlen (mfile) − 1] == '/')
{
    error (0, 0, "checkout_proc: trailing slash on mfile (%s)!",
            mfile);
}

/* Does mfile have multiple path elements? */
cp = strrchr (mfile, '/');
if (cp != NULL)
{
    *cp = '\0';
    (void) strcat (repository, '*');
    (void) strcat (repository, mfile);
    mfile = cp + 1;
}

/* Now mfile is a single path element. */
path = xmalloc (strlen (repository) + strlen (mfile) + 5);
(void) sprintf (path, "%s/%s", repository, mfile);
if (isdir (path))
{
    /* It's a directory, so tack it on to repository and
     * where, as we did above. */
    (void) strcat (repository, '/');
    (void) strcat (repository, mfile);
    (void) strcat (where, '/');
    (void) strcat (where, mfile);
}
else
{
    /* It's a file, which means we have to screw around with
     * argv. */

    int i;

    /* Paranoia check. */
    if (*pargc > 1)
    {
        error (0, 0, "checkout_proc: trashing argv elements!");
        for (i = 1; i < *pargc; i++)
        {
            error (0, 0, "checkout_proc: argv[%d] '%s'",
                    i, argv[i]);
        }
    }
    for (i = 1; i < *pargc; i++)
    {
        free (argv[i]);
        argv[i] = xstrdup (mfile);
        *(pargc) = 2;
    }
    free (path);
}
if (preload_update_dir != NULL)
{
    preload_update_dir =
        xrealloc (preload_update_dir,
                  strlen (preload_update_dir) + strlen (where) + 5);
    strcat (preload_update_dir, '/');
    strcat (preload_update_dir, where);
}
else
    preload_update_dir = xstrdup (where);

    /*
    * At this point, where is the directory we want to build, repository is
    * the repository for the lowest level of the path.
    * We need to tell build_dirs not only the path we want it to
    * build, but also the repositories we want it to populate the
    * path with. To accomplish this, we walk the path backwards, one
    * pathname component at a time, constructing a linked list of
    * struct dir_to_build.
    */

    /*
    * If we are sending everything to stdout, we can skip a whole bunch of
    * work from here
    */
    if (pipeout)


720 {
    struct dir_to_build *head;
    char *reposcopy;

    if (strncmp (repository, CVSroot, directory,
                strlen (CVSroot) - directory)) != 0)
        error (1, 0, \": internal error: %s doesn't start with %s in checkout_proc
            ", repository, CVSroot);

        /* We always create at least one directory, which corresponds to
          the entire strings for WHERE and REPOSITORY. */
        head = (struct dir_to_build *) xmalloc (sizeof (struct dir_to_build));
        /* Special marker to indicate that we don't want build\dirname and .chdir
         to create the CVSADM directory for us. */
        head->repository = NULL;
        head->dirpath = xstrdup (where);
        head->next = NULL;
        head->justchdir = 0;

    740 /* Make a copy of the repository name to play with. */
    reposcopy = xstrdup (repository);

    /* FIXME: this should be written in terms of last_component
       instead of hardcoded '/'. This presumably affects OS/2,
       NT, &c., if the user specifies \\'. Likewise for the call
       to findslash. */
    cp = where + strlen (where);

    while (1)
    {
        struct dir_to_build *new;

        cp = findslash (where, cp - 1);
        if (cp == NULL)
            break; /* we're done */

        new = (struct dir_to_build *)
            xmalloc (sizeof (struct dir_to_build));
        new->dirpath = xmalloc (strlen (where));

    760 /* If the user specified an absolute path for where, the
       last path element we create should be the top-level
       directory. */
    if (cp == where)
    {
        strcpy (new->dirpath, where);
        new->dirpath[cp - where] = '\0';
    }
    else
    {
        /* where should always be at least one character long. */
        assert (strlen (where));
        strcpy (new->dirpath, "/");
    }

    770 new->next = head;
    head = new;

    /* If where consists of multiple pathname components,
       then we want to just cd into it, without creating
       directories or modifying CVS directories as we go.
       In CVS 1.9 and earlier, the code actually does a
       CVS\dirname up-front; I'm not going to try to go back
       to that exact code but this is somewhat similar
       in spirit. */
    if (where == NULL
       && cp == where < strlen (where))
    {
        new->repository = NULL;
        new->justchdir = 1;
        continue;
    }

    790 new->justchdir = 0;

    /* Now figure out what repository directory to generate.
       The most complete case would be something like this:

       The modules file contains
         foo -d bar/baz quux

       The command issued was:
         cvs co -d what/ever -N foo

       The results in the CVS/Repository files should be:
         . -> (don't touch CVS/Repository)
           (I think this case might be buggy currently)
         what -> (don't touch CVS/Repository)
         ever -> .. (same as "cd what/ever; cvs co -N foo")

    800
}
If (strcpy (reposcopy, CVSroot_directory) == 0)
{
    /* We can't walk up past CVSROOT. Instead, the
       repository should be Emptydir. */
    new->repository = emptydir_name ();
}
else
{
    if (where ![= NULL)
        & & (strcmp (new->dirpath, where) == 0))
        {
            /* It's the case that the user specified a
               destination directory with the "-d" flag. The
               * repository in this directory should be "." *
               * since the user's command is equivalent to:
               * cd <dir>; cvs co blah */
            strcpy (reposcopy, CVSroot_directory);
            goto allocate_repo;
        }
    else if (mwhere ![= NULL)
        { /* This is a generated directory, so point to
            CVSNULLREPOS. */
            new->repository = emptydir_name ();
        }
    else
        {
            /* It's a directory in the repository! */
            char *rp = strchr (reposcopy, '/');
            /* We'll always be below CVSROOT, but check for
               paranoia's sake. */
            if (rp ![= NULL)
                error (1, 0,
                    "internal error: %s doesn't contain a slash",
                    reposcopy);
                *rp = '\0';
            allocate_repo:
            new->repository = xmalloc (strlen (reposcopy) + 5);
            (void) strcpy (new->repository, reposcopy);
            if (strcmp (reposcopy, CVSroot_directory) == 0)
                { /* Special case - the repository name needs
                    to be "/path/to/repos/", (the trailing dot
                    is important). We might be able to get rid
                    of this after the we check out the other
                    code that handles repository names. */
                    (void) strcat (new->repository, "/.");
                }
        }
    }
/* clean up */
free (reposcopy);
{
    int where ![= absolute = isabsolute (where);
    /* The top-level CVSAADM directory should always be
       CVSroot_directory. Create it, but only if WHERE is
       relative. If WHERE is absolute, our current directory
       may not have a thing to do with where the sources are
       being checked out. If it does, build_dir_and_chdir
       will care of creating adm files here. */
    /* FIXME: checking where ![= absolute is a horrid kludge;
       I suspect we probably can just skip the call to
       build_dir_and_chdir whenever the -d command option
       was specified to checkout. */
    if (! where ![= absolute & & top_level_dir)
        { /* It may be argued that we shouldn't set any sticky
            bits for the top-level repository. FIXME? */
            build_dir (CVSroot_directory, "/", pargc <= 1);
        }
    ifdef SERVER_SUPPORT
    /* We always want to have a top-level admin
       directory. If we're running in client/server mode,
       send a "Clear-static-directory" command to make
sure it is created on the client side. (See 5.10
in cofficient.dvi to convince yourself that this is
OK.) If this is a duplicate command being sent, it
will be ignored on the client side. */

if (server_active)
  server_clear_entstat (*.*, CVSroot directory);

}  {

if (build_dirs && chdir (head, *pargc < 1) != 0)
{
  error (0, 0, "ignoring module %s", omodule);
  err = 1;
  goto out;
}

}  {

if (!isfile (CVSADM))
{
  FILE *fp;
  if (!noexec && *pargc > 1)
  {
    /* I'm not sure whether this check is redundant. */
    if (lslidir (repository))
      error (1, 0, "there is no repository %s", repository);
    Create_Admin (".*", preload, update_dir, repository,
      (char *) NULL, (char *) NULL, 0, 0);
    fp = open_file (CVSADM_ENTSTAT, *w*);
    if (fclose (fp) == EOF)
      error (1, errno, "cannot close %s", CVSADM_ENTSTAT);
    ifdef SERVER_SUPPORT
    server_set_entstat (where, repository);
    ifdef }
    else
    {
      /* I'm not sure whether this check is redundant. */
      if (lslidir (repository))
        error (1, 0, "there is no repository %s", repository);
      Create_Admin (".*", preload, update_dir, repository, tag, date,
        /* FIXME? This is a guess. If it is important */
        /* for nonbranch to be set correctly here I */
        /* think we need to write it one way now and */
        /* then rewrite it later via WriteTag, once */
        /* we've had a chance to call RCSnodeisbranch */
        /* on each file. */
        0, 0);
    }
  }
}

}  {
  char *repos;
  /* get the contents of the previously existing repository */
  repos = Name_Repository ((char *) NULL, preload, update_dir);
  if (fncmp (repository, repos) != 0)
  {
    error (0, 0, "existing repository %s does not match %s",
      repos, repository);
    error (0, 0, "ignoring module %s", omodule);
    free (repos);
    err = 1;
    goto out;
  }
  free (repos);
}

}  {

/* If we are going to be updating to stdout, we need to cd to the */
/* repository directory so the recursion processor can use the current */
/* directory as the place to find repository information */
if (pipeout)
{
  if (CVS_CHDIR (repository) < 0)
  {
error (0, errno, "cannot chdir to %s", repository);
err = 1;
goto out;
} which = W_REPO:
if (tag != NULL && tag->validated)
{
tag->check_valid (tag, *pargc - 1, argv + 1, 0, aflag, NULL);
tag->validated = 1;
}
} else
{
which = W_LOCAL | W_REPO:
if (tag != NULL && tag->validated)
{
tag->check_valid (tag, *pargc - 1, argv + 1, 0, aflag, repository);
tag->validated = 1;
}
if (tag != NULL || date != NULL || join_rev1 != NULL) which [= W_ATTIC):
if (!join_tags_validated)
{
if (join_rev1 != NULL)
tag->check_valid (join_rev1, *pargc - 1, argv + 1, 0, aflag, repository);
if (join_num2 != NULL)
tag->check_valid (join_rev2, *pargc - 1, argv + 1, 0, aflag, repository);
join->tags_validated = 1;
} /* if we are going to be recursive (building dirs), go ahead and call the
* update recursion processor. We will be recursive unless either local
* only was specified, or we were passed arguments
*/
if ((local_specified || *pargc > 1))
{
if (strcmp (command, "export") != 0 && (pipeout
    history_write ("0", preload_update_dir, history_name, where, repository);
else if (strcmp (command, "export") == 0 && (pipeout
    history_write ("E", preload_update_dir, tag ? tag : date, where, repository);
err += do_update (0, (char **) NULL, options, tag, date,
force_tag_match, 0 /* local */ ,
1 /* update -d */ , aflag, checkout_prune_dirs, pipeout, which, join_rev1, join_rev2, preload_update_dir);
goto out;
}
if (!pipeout)
{ int i;
List *entries;
/* we are only doing files, so register them */
entries = EntriesOpen (0, NULL);
for (i = 1; i < *pargc; i++)
{
char *line;
VersionTS *vers;
struct fileinfo info_info;
memset (&info, 0, sizeof info);
info.file = argv[0];
/* Shouldn't be used, so set to arbitrary value. */
info.update_dir = NULL;
info.file = argv[0];
info.repository = repository;
info.entries = entries;
/* The rcs slot is needed to get the options from the RCS
* file */
info.rcs = RCS_parse (info.file, repository);
version = VersionTS (&info, options, tag, date, force_tag_match, 0);
if (version->user == NULL)
{ line = xmalloc (strlen (info.file) + 15);
(void) sprintf (line, "Initial %s", info.file);
Register (entries, info.file, vers->version ? vers->version : "0", 
line, vers->options, vers->tag,
1080  free (line);}
}freevers=1 (&vers);freeversnode (&info.rcs);
}EntriesClose (entries);

1090  /* Don't log "export", just regular "checkouts" */
if (strcmp (command_name, "export") != 0 && (pipeout)
history_write ('0', preload_update_dir, history_name, where, repository);

1100  /* go ahead and call update now that everything is set */
err += do_update (space, -1, argv + 1, options, tag, date,
force_tag_match, localspecified, 1 /* update -d */,
 aflag, checkout_prune, dirs, pipeout, which, join, ev1,
join_rev2, preload_update_dir);
out:
freesavetrainupdate_dir;free (where);free (repository);
return (err);
}

1110  static char *findslash (start, p)
char *start;
{char *p;
while (p >= start && *p != '/') p--; /
/FIXME: indexing off the start of the array like this is NOT OK according to ANSI, and will break some of the time on certain
segmented architectures. */
if (p < start) return (NULL);
else return (p);
}

1120  /* Return a newly malloc'd string containing a pathname for CVSNULLREPOS,
and make sure that it exists. If there is an error creating the
directory, give a fatal error. Otherwise, the directory is guaranteed
to exist when we return. */
char *emptydir_name ()
{
char *repository;
repository = xmalloc (strlen (CVSroot_directory)
+ sizeof (CVSROOTADM)
+ sizeof (CVSNULLREPOS)
+ 10);
(void) sprintf (repository, "%s/\%s\%s", CVSroot_directory,
CVSROOTADM, CVSNULLREPOS);
if (lufsbuf (repository)) {
mode_t umask;
umask = umask (cvsumask);
if (CVSCHKDIR (repository, 0777) < 0)
error (1, errno, "cannot create \%s", repository);
(void) umask (umask);
}
return repository;
}

1140  /* Build all the dirs along the path to DIRS with CVS subdirs with appropriate
repositories. If >repository is NULL, do not create a CVSADM directory
for that subdirectory; just CVS_CHDIR into it. */
static int build_dirs_and_chdir (dirs, sticky)
struct dir **build +dirs;
int sticky;
{int retval = 0;
struct dir **nextdir;
while (dirs != NULL)
{
char *dir = last_component (dirs->dirpath);
if (dirs->justchdir)
{
mkdir_needed (dir);
Subdir_Register (NULL, NULL, dir);
}
if (CVS_CHDIR (dir) < 0) {
    error (0, errno, "cannot chdir to %s", dir);
    retval = 1;
    goto out;
}
if (dirs->repository != NULL) {
    build_one_dir (dirs->repository, dirs->dirpath, sticky);
    free (dirs->repository);
}
nextdir = dirs->next;
free (dirs->dirpath);
free (dirs);
dirs = nextdir;
}

out:
return retval;
}
A.7 classify.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"

static void sticky ck PROTO ((struct file_info *fiinfo, int aflag,
    Vers_TS *vers));

/*
 * Classify the state of a file
 */

Ctype

Classify_File (fiinfo, tag, date, options, force_tag_match, aflag, versp,
    pipeout)

struct file_info *fiinfo;
char *tag;
char *date;

/* Keyword expansion options. Can be either NULL or "-" to
   indicate none are specified here. */
char *options;

int force_tag_match;
int aflag;
Vers_TS *versp;
int pipeout;

{ Vers_TS *vers;
  Ctype ret;

  /* get all kinds of good data about the file */
  vers = Version_TS (fiinfo, options, tag, date,
                    force_tag_match, 0);

  if (vers->vn_user == NULL)
  { /* No entry available, ts_rcs is invalid */
    if (vers->vn_rcs == NULL)
    { /* there is no RCS file either */
      if (vers->ts_user == NULL)
      { /* there is no user file */
        /* FIXME: Why do we skip this message if vers->tag or
           vers->date is set? It causes "cvs update -r tag98 foo"
           to silently do nothing, which is seriously confusing
           behavior. "cvs update foo" gives this message, which
           is what I would expect. */
        if (force_tag_match || ! (vers->tag || vers->date))
          if (really_quiet)
            error (0, 0, "nothing known about %s", fiinfo->fullname);
        ret = T_UNKNOWN;
      }
    } else
    { /* there is a user file */
      /* FIXME: Why do we skip this message if vers->tag or
         vers->date is set? It causes "cvs update -r tag98 foo"
         to silently do nothing, which is seriously confusing
         behavior. "cvs update foo" gives this message, which
         is what I would expect. */
      if (force_tag_match || ! (vers->tag || vers->date))
        if (really_quiet)
          error (0, 0, "use 'ls add' to create an entry for %s",
                  program_name, fiinfo->fullname);
        ret = T_UNKNOWN;
    }
  } else if (RCS_indir (vers->srcfile, vers->vn_rcs))
  { if (vers->ts_user == NULL)
    ret = T_UPTODATE;
  else
    { error (0, 0, "use 'ls add' to create an entry for %s",
                program_name, fiinfo->fullname);
      ret = T_UNKNOWN;
    }
  } else
  { /* there is an rcs file */

```c
/* Classify the state of a file */

Ctype

Classify_File (fiinfo, tag, date, options, force_tag_match, aflag, versp,
    pipeout)

struct file_info *fiinfo;
char *tag;
char *date;

/* Keyword expansion options. Can be either NULL or "-" to
   indicate none are specified here. */
char *options;

int force_tag_match;
int aflag;
Vers_TS *versp;
int pipeout;

{ Vers_TS *vers;
  Ctype ret;

  /* get all kinds of good data about the file */
  vers = Version_TS (fiinfo, options, tag, date,
                    force_tag_match, 0);

  if (vers->vn_user == NULL)
  { /* No entry available, ts_rcs is invalid */
    if (vers->vn_rcs == NULL)
    { /* there is no RCS file either */
      if (vers->ts_user == NULL)
      { /* there is no user file */
        /* FIXME: Why do we skip this message if vers->tag or
           vers->date is set? It causes "cvs update -r tag98 foo"
           to silently do nothing, which is seriously confusing
           behavior. "cvs update foo" gives this message, which
           is what I would expect. */
        if (force_tag_match || ! (vers->tag || vers->date))
          if (really_quiet)
            error (0, 0, "nothing known about %s", fiinfo->fullname);
        ret = T_UNKNOWN;
      }
    } else
    { /* there is a user file */
      /* FIXME: Why do we skip this message if vers->tag or
         vers->date is set? It causes "cvs update -r tag98 foo"
         to silently do nothing, which is seriously confusing
         behavior. "cvs update foo" gives this message, which
         is what I would expect. */
      if (force_tag_match || ! (vers->tag || vers->date))
        if (really_quiet)
          error (0, 0, "use 'ls add' to create an entry for %s",
                  program_name, fiinfo->fullname);
        ret = T_UNKNOWN;
    }
  } else if (RCS_indir (vers->srcfile, vers->vn_rcs))
  { if (vers->ts_user == NULL)
    ret = T_UPTODATE;
  else
    { error (0, 0, "use 'ls add' to create an entry for %s",
                program_name, fiinfo->fullname);
      ret = T_UNKNOWN;
    }
  } else
  { /* there is an rcs file */
```
if (vers->ts_user == NULL) {
  /* There is no user file; needs checkout */
  if (vers->vn.remote) ret = T_REMOTE;
  else ret = T_CHECKOUT;
} else {
  if (pipeout) {
    /* The user file doesn't necessarily have anything
     * to do with this.
    */
    ret = T_CHECKOUT;
  }
  /* There is a user file; print a warning and add it to the
   * conflict list, only if it is indeed different from what we
   * plan to extract
   */
  else if (No_Difference (finfo, vers)) {
    /* the files were different so it is a conflict */
    if (!really_quiet) error (0, 0, "move away %s; it is in the way",
                         finfo->fullname);
    ret = T_CONFLICT;
  } else {
    /* since there was no difference, still needs checkout */
    ret = T_CHECKOUT;
  }
} else if (strcmp (vers->vn_user, "0") == 0) {
  /* An entry for a new-born file; tsrcs is dummy */
  if (vers->ts_user == NULL) {
    /* There is no user file, but there should be one; remove the
     * entry
    */
    if (!really_quiet) error (0, 0, "warning: new-born %s has disappeared",
                            finfo->fullname);
    ret = T_REMOVE_ENTRY;
  } else {
    /* There is a user file */
    if (vers->vn_rcs == NULL) {
      /* There is no RCS file, added file */
      ret = T_ADDED;
    } else if (RCS_isdead (vers->arcfile, vers->vn_rcs)) {
      /* we are resurrecting. */
      ret = T_ADDED;
    } else {
      if (vers->arcfile->flags & INATTIC
          && vers->arcfile->flags & VALID) {
        /* This file has been added on some branch other than
         * the one we are looking at. In the branch we are
         * looking at, the file was already valid. */
        if (!really_quiet) error (0, 0,
                                 conflict: %s has been added, but already exists",
                                 finfo->fullname);
      } else {
        /* There is an RCS file, so someone else must have checked
         * one in behind our back; conflict
        */
        if (!really_quiet) error (0, 0,
                                 conflict: %s created independently by second party",
                                 finfo->fullname);
      }
      ret = T_CONFLICT;
    }
  }
}
else if (vers->vn_user[0] == '*') {
  /* An entry for a removed file, ts_recs is invalid */
  if (vers->ts_user == NULL) {
    /* There is no user file (as it should be) */
    if (vers->vn_recs == NULL, NULL) {
      /* There is no RCS file; this is all-right, but it has been
       * removed independently by a second party; remove the entry
       * */
      ret = TR_REMOVED_ENTRY;
    }
    else if (vers->vn_recs == NULL, NULL) {
      /* The RCS file is the same version as the user file was, and
       * that's OK; remove it */
      ret = TR_REMOVED;
    } else {
      /* The RCS file is a newer version than the removed user file
       * and this is definitely not OK; make it a conflict. */
      if (!really_quiet) error [0, 0, "conflict: %s is modified by second party",
      .info->fullname());
      ret = TR_CONFLICT;
    }
    /* The user file shouldn't be there */
    if (!really_quiet)
      error [0, 0, "Ts should be removed and is still there",
      .info->fullname());
    ret = TR_REMOVED;
  }
  else if (atrcmp (vers->ts_user, vers->ts_recs) == 0) {
    /* The user file is still unmodified, so just remove it from
     * the entry list */
    if (!really_quiet)
      error [0, 0, "Ts is no longer in the repository",
      .info->fullname());
    ret = TR_REMOVED_ENTRY;
  } else {
    /* The user file has been modified and since it is no longer
     * in the repository, a conflict is raised */
    if (NM_Difference (info, vers)) {
      /* they are different > conflict */
      if (!really_quiet)
        error [0, 0, "conflict: %s is modified but no longer in the repository",
    //...
classify.c • 107

```c
finfo->fullname);
ret = T_CONFLICT;
}
else
{
  /* they weren't really different */
  if (really_quiet)
    error (0, 0, "warning: %s is not (any longer) pertinent",
         finfo->fullname);
  ret = T_REMOVEENTRY;
}
}
else if (strcmp (vers->vn_csa, vers->vn_user) == 0)
{
  /* The RCS file is the same version as the user file */
  if (vers->tv_user == NULL)
  
    /* There is no user file, so note that it was lost and 
       extract a new version */

    /* Comparing the command name against "update", in 
       addition to being an ugly way to operate, means 
       that this message does not get printed by the 
       server. That might be considered just a straight 
       bug, although there is one subtlety; that case also 
       gets hit when a patch fails and the client fetches 
       a file. I'm not sure there is currently any way 
       for the server to distinguish those two cases. */

    if (strcmp (command_name, "update") == 0)
      if (really_quiet)
        error (0, 0, "warning: %s was lost", finfo->fullname);
      ret = T_CHECKOUT;

    else if (strcmp (vers->tv_user, vers->tv_csa) == 0)
    
      /* The user file is still unmodified, so nothing special at 
       * all to do – no lists updated, unless the sticky -k option 
       * has changed. If the sticky tag has changed, we just need 
       * to re-register the entry */

      /* TODO: decide whether we need to check file permissions 
       * for a mismatch, and return T_CONFLICT if so. */

      if (strcmp (vers->entdata->options, vers->options) != 0)
        ret = T_CHECKOUT;

    else
      
      /* The user file appears to have been modified, but we call 
       * NDIFFDifference to verify that it really has been modified 
       */
      if (NDIFFDifference (finfo, vers))

        /* they really are different; modified if we aren't 
         * changing any sticky -k options, else needs merge */

        ifdef XXX_FIXME_WHEN_RCSMERGE_IS_FIXED
          if (strcmp (vers->entdata->options, vers->entdata->options, vers->options) == 0)
            ret = T_MODIFIED;
          else
            ret = T_NEEDS_MERGE;
        else
          ret = T_MODIFIED;
        endif

      sticky_c (finfo, aflag, vers);
      ret = T_UPTODATE;
    }
  }
  
  /* The user file appears to have been modified, but we call 
   * NDIFFDifference to verify that it really has been modified 
   */
  if (NDIFFDifference (finfo, vers))

    /* they really are different; modified if we aren't 
     * changing any sticky -k options, else needs merge */

    ifdef XXX_FIXME_WHEN_RCSMERGE_IS_FIXED
      if (strcmp (vers->entdata->options, vers->entdata->options) == 0)
        ret = T_MODIFIED;
      else
        ret = T_NEEDS_MERGE;
    else
      ret = T_MODIFIED;
    endif

    sticky_c (finfo, aflag, vers);
  }
  
  else
  
    /* file has not changed; check out if -k changed */
    if (strcmp (vers->entdata->options, vers->entdata->options) != 0)
      ret = T_CHECKOUT;
    
```

360 } else {
    /* else -> note that NoDifference will Register the
       file already for us, using the new tag/date. This
       is the desired behaviour */
    ret = T_UPDATE;
}
370 } else /* The RCS file is a newer version than the user file */
    if (vers->ts_user == NULL) {
      /* There is no user file, so just get it */
    } else if (strcmp (vers->ts_user, vers->ts_rca) == 0) {
    /* The user file is still unmodified, so just get it as well */
    }
380 endif SERVER_SUPPORT
    if (vers->vname != NULL) {  
      ret = T_REMOTE;
    } else if (strcmp (vers->entdata->options , vers->entdata->options , "", vers->options) != 0
390 ) {
      ret = T_CHECKOUT;
    } else if (strcmp (vers->vname, vers->vname) != 0)
        ret = T_PATCH;
    endif
  } else /* really modified, needs to merge */
    ret = T_NEEDS_MERGE;
endif SERVER_SUPPORT
else if (vers->vname != NULL) {
    ret = T_REMOTE;
  } else if (strcmp (vers->entdata->options, vers->entdata->options, "", vers->options) != 0
400 ) {
    ret = T_CREATE;
  } else {
    ret = T_PATCH;
  } else /* not really modified, check it out */
        ret = T_CHECKOUT;
endif
410 } else /* free up the vers struct, or just return it */
  if (vers != (Vers_TS **) NULL)  
    versp = vers;
  else
    freesvers (&vers);
420 } /* return the status of the file */
  return (ret);
}

static void
stickyck (finfo, aflag, vers)
struct file_info *finfo, int aflag;
VersTS *vers;

450 {
  if ((aflag || vers->tag || vers->date))
  {
    char *enttag = vers->entdata->tag;
    char *entdate = vers->entdata->date;

    if ((enttag && vers->tag && !strcmp(enttag, vers->tag)))
      ((enttag && vers->tag) || (enttag && vers->tag))
    {
      Register (finfo->entries, finfo->file, vers->tag_user, vers->ts_rcs,
                 vers->options, vers->tag, vers->date, vers->ts_conflict, CVSroot_directory, finfo->repository);
    }
  }
##ifdef SERVER_SUPPORT
  if (server_active)
  {
    /* We need to update the entries line on the client side.
     It is possible we will later update it again via server
     server updated or some such, but that is OK. */
    server_update_entries
      ((finfo->file, finfo->update_dir, finfo->repository,
        !strcmp(vers->ts_rcs, vers->ts_user) == 0)
          SERVER_UPDATED : SERVER_MERGED);
  }
##endif
}
}
A.8  client.c

/* CVS client-related stuff. */

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. */

#ifndef HAVE_CONFIG_H
#include "config.h"
#endif /* HAVE_CONFIG_H */

#include <assert.h>
#include "cva.h"
#include "getline.h"
#include "edit.h"
#include "buffer.h"
#include "winsock.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netinet/tcp.h"
#endif /* No winsock.h */

#endif

/* If SOCK_ERRNO is defined, then send()/recv() and other socket calls do not set errno, but that this macro should be used to obtain an error code. This probably doesn't make sense unless NO_SOCKET_TO_FD is also defined. */

if defined[AUTH_CLIENT_SUPPORT] || HAVE_KERBEROS || defined(SOCKET_ERRNO) || defined(SOCKET_STERROR)
#
if defined[WINSOCK] || HAVE_KERBEROS || defined(SOCKET_ERRNO)
#
include <winsock.h>
#else
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#endif
#endif /* No winsock.h */

endif

/* If SOCK_STERROR is defined, then the error codes returned by socket operations are not known to strerror, and this macro must be used instead to convert those error codes to strings. */

if defined[SOCK_STERROR]
#define SOCKET_STERROR strerror
#endif

if STDC_HEADERS

#endif

if defined[sterror]
extern char *sterror();
#endif

endif /* ! SOCK_STERROR */

if defined[CVS_PORT] 24011

if defined[KRB] || defined[WINSOCK] || HAVE_KERBEROS
#endif

if defined[KRB] || defined[WINSOCK] || HAVE_KERBEROS
#include <krb.h>
 extern char *krb_realmofhost();
#endif

if defined[KRB_GET_ERRTEXT] || defined[KRB_GET_ERRTEXT] || defined[KRB_GET_ERRTEXT]
#define krb_get_errtext(status) krb_err_text[status]
#endif

/* Information we need if we are going to use Kerberos encryption. */
static CBlock kblock;
static Key schedule sched;
#endif /* HAVE_KERBEROS */

#endif /* HAVE_KERBEROS */

if defined[GSSAPI]
#include <gssapi.h>
#endif

if defined[GSSAPI] || defined[GSSAPI_GSSAPI_GSSAPI]
#include <gssapi/gssapi.h>
#endif

if defined[GSSAPI] || defined[GSSAPI_GSSAPI_GSSAPI_GSSAPI]
#include <gssapi/gssapi_generic.h>
#endif
static void add_prune_candidatePROTO((char *));

void add_prune_candidate(char * file, char * server, char * root, char * repository);

/* All the commands. */
int addPROTO((int argc, char ** argv));

int adminPROTO((int argc, char ** argv));

int commitPROTO((int argc, char ** argv));

int diffPROTO((int argc, char ** argv));

int importPROTO((int argc, char ** argv));

int cvslogPROTO((int argc, char ** argv));

int patchPROTO((int argc, char ** argv));

int releasePROTO((int argc, char ** argv));

int cveremovePROTO((int argc, char ** argv));

int rtagPROTO((int argc, char ** argv));

int statusPROTO((int argc, char ** argv));

int tagPROTO((int argc, char ** argv));

int updatePROTO((int argc, char ** argv));

/* All the response handling functions. */
static void handleokPROTO((char * , int));

static void handleerrorPROTO((char * , int));

static void handlevaliquestsPROTO((char * , int));

static void handlecheckPROTO((char * , int));

static void handleentryPROTO((char * , int));

static void handlechecksumPROTO((char * , int));

static void handlecopyfilePROTO((char * , int));

static void handleupdatedPROTO((char * , int));

static void handlemergedPROTO((char * , int));

static void handlepatchedPROTO((char * , int));

static void handlecsdiffPROTO((char * , int));

static void handleremovedPROTO((char * , int));

static void handle_remove_entryPROTO((char * , int));

static void handle_set_directoryPROTO((char * , int));

static void handle_clear_directoryPROTO((char * , int));

static void handle_set_stickyPROTO((char * , int));

static void handle_clear_stickyPROTO((char * , int));

static void handle_set_keepproto((char * , int));

static void handle_clear_keepproto((char * , int));

static void handle_module_keepproto((char * , int));

static void handle_wrapproto((char * , int));

static void handle_module_wrapproto((char * , int));

static void handle_wrapproto((char * , int));

/* Shared with server. */

/*
 * Return a malloc'd, \0-terminated string corresponding to the mode in SB.
 */
char *
static mode_to_string(mode);

static mode_to_string(mode);

/*
endif /* STDCC */

(if defined(CLIENT_SUPPORT) || defined(SERVER_SUPPORT))

*/
if (mode & S_IWUSR) u[i++] = 'u';
if (mode & S_IWGRP) u[i++] = 'u';
if (mode & S_IWOTH) u[i++] = 'u';
in = 0;
if (mode & S_IRUSR) g[i++] = 'r';
if (mode & S_IXUSR) g[i++] = 'x';
if (mode & S_IRGRP) g[i++] = 'r';
if (mode & S_IXGRP) g[i++] = 'x';
in = 0;
if (mode & S_IROTH) o[i++] = 'r';
if (mode & S_IWOTH) o[i++] = 'u';
if (mode & S_IXOTH) o[i++] = 'x';
if (mode & S_IROTH) o[i++] = 'x';
sprintf(buf, "%s,u=%s,g=%s,o=%s", u, g, o);
return xstrdup(buf);
}

/* Change mode of FILENAME to MODESTRING. *
 * Returns 0 for success or errno code.
 * If RESPECT Umask is set, then honor the umask.
 */

int change_mode(filename, mode_string, respect_umask)
char *filename;
char *mode_string;
int respect_umask;
{
#define CHMOD BROKEN
char *p;
int writable = 0;

/* We can only distinguish between
1) readable
2) writeable
3) Picasso’s “Blue Period”
We handle the first two. */
p = mode_string;
while (*p != '\0') {
  if ((p[0] == 'u' | p[0] == 'g' | p[0] == 'o') & & p[1] == '*') {
    char *q = p + 2;
    while (*q != '*' & & *q != '\0') {
      if (*q == 'w')
        writable = 1;
      q++;
    }
  }
  /* Skip to the next field. */
  while (*p == '*' & & *p != '\0')
    ++p;
  if (*p == '*')
    ++p;
}

/* chmod honors the umask for us. In the respect_umask case, we
don’t try to cope with it (probably to handle that well, the server
needs to deal with modes in data structures, rather than via the
modes in temporary files). */

if (xchmod (filename, writable);
return 0;
else /* !CHMOD BROKEN */
char *p;
mode_t mode = 0;
mode_t umask;
p = mode_string;
while (*p != '\0') {
  if ((p[0] == 'u' | p[0] == 'g' | p[0] == 'o') & & p[1] == '*') {
    int can_read = 0, can_write = 0, can_execute = 0;
    char *q = p + 2;
    while (*q != '*' & & *q != '\0') {
      if (*q == 'r')
        can_read = 1;
      else if (*q == 'w')
        can_write = 1;
      else if (*q == 'x')
        can_execute = 1;
      q++;
    }
  }
if (p[0] == 'u')
{
if (can_read)
  mode |= S_IRUSR;
if (can_write)
  mode |= S_IWUSR;
if (can_execute)
  mode |= S_IXUSR;
}
else if (p[0] == 'g')
{
if (can_read)
  mode |= S_IRGRP;
if (can_write)
  mode |= S_IWGRP;
if (can_execute)
  mode |= S_IXGRP;
}
else if (p[0] == 'o')
{
if (can_read)
  mode |= S_IROTH;
if (can_write)
  mode |= S_IWOTH;
if (can_execute)
  mode |= S_IXOTH;
}

/* Skip to the next field. */
while (*p != '!' & & *p != '\0')
++p;
if (*p == 'k')
++p;

if (respect_umask)
{
  umask = umask(0);
  (void) umask(umask);
  mode & "-" umask;
}

if (chmod(filename, mode) < 0)
  return errno;
return 0;
endif /* !CHMOD_BROKEN */
#endif /* CLIENT_SUPPORT or SERVER_SUPPORT */

struct CLIENT_SUPPORT

int client_prune_dirs;
static List *signlist = (List *) NULL;
/* Buffer to write to the server. */
static struct buffer *twoserver;
/* The stream underlying to server, if we are using a stream. */
static FILE *toserver;

/* Buffer used to read from the server. */
static struct buffer *fromserver;
/* The stream underlying from_server, if we are using a stream. */
static FILE *fromserver;
/* Process ID of rsh subprocess. */
static int rshpid = -1;

/* We want to be able to log data sent between us and the server. We do it using log buffers. Each log buffer has another buffer which handles the actual I/O, and a file to log information to.

This structure is the closure field of a log buffer. */
struct log_buffer
{
  /* The underlying buffer. */
  struct buffer *buf;
  /* The file to log information to. */
  FILE *log;
};

static struct buffer *log_buffer_initialize
  PROTO(struct buffer * FILE *, int, void () (struct buffer *));
static int log_buffer_input PROTO(void *, char *, int, int, int *);
static int log_buffer_output PROTO(void *, const char *, int, int *);
static int log_buffer_flush PROTO(void *);
static int log_buffer_block PROTO(void *, int);
**Source code**

```c
static int log_buffer_shutdown PROTO((void *));

/* Create a log buffer. */

static struct buffer *
log_bufferInitialize (buf, fp, input, memory)

    struct buffer *buf;
    FILE *fp;
    int input;

    void (*memory) PROTO((struct buffer *));

    struct log_buffer *n;

    n = (struct log_buffer *) xmalloc (sizeof *n);

    n->buf = buf;

    n->log = fp;

    return bufInitialize (input ? log_buffer_input : NULL,
                             input ? NULL : log_buffer_output,
                             input ? NULL : log_buffer_flush,
                             log_buffer_block,
                             log_buffer_shutdown,
                             memory,
                             n);

/* The input function for a log buffer. */

static int
log_buffer_input (closure, data, need, size, got)

    void *closure;
    char *data;

    int need;
    int size;

    int *got;

    struct log_buffer *lb = (struct log_buffer *) closure;

    int status;

    size_t to_write;

    if ((lb->buf->input == NULL)
                           abort ());

    status = (*lb->buf->input) (lb->buf->closure, data, need, size, got);

    if (status != 0)
        return status;

    if (*got > 0)
    {
        *got = *got - got;

        if (fwrite (data, 1, *to_write, lb->log) != *to_write)
            error (0, errno, "writing to log file");
    }

    return 0;

/* The output function for a log buffer. */

static int
log_buffer_output (closure, data, have, wrote)

    void *closure;
    const char *data;

    int have;

    int *wrote;

    struct log_buffer *lb = (struct log_buffer *) closure;

    int status;

    size_t to_write;

    if ((lb->buf->output == NULL)
                           abort ());

    status = (*lb->buf->output) (lb->buf->closure, data, have, *wrote);

    if (status != 0)
        return status;

    if (*wrote > 0)
    {
        *wrote = *wrote - wrote;

        if (fwrite (data, 1, *to_write, lb->log) != *to_write)
            error (0, errno, "writing to log file");
    }

    return 0;

/* The flush function for a log buffer. */

static int
log_buffer_flush (closure)
```

void +closure;

450 {  
    struct log_buffer *lb = (struct log_buffer *) closure;
    if (lb->buf->flush == NULL)
        abort();
    /* We don't really have to flush the log file here, but doing it
        will let tail -f on the log file show what is sent to the
        network as it is sent. */
    if (fflush(lb->log) != 0)
        error(0, errno, "flushing log file");
    return (lb->buf->flush)(lb->buf->closure);
}

/* The block function for a log buffer. */
static int
log_buffer_block (closure, block)
    void *closure;

470 {  
    struct log_buffer *lb = (struct log_buffer *) closure;
    if (block)
        return set_block (lb->buf);
    else
        return set_nonblock (lb->buf);
}

/* The shutdown function for a log buffer. */
static int
log_buffer_shutdown (closure)
    void *closure;

480 {  
    struct log_buffer *lb = (struct log_buffer *) closure;
    int retval;
    retval = buf_shutdown (lb->buf);
    if (fclose(lb->log) < 0)
        error(0, errno, "closing log file");
    return retval;
}

sifdef NO_SOCKET_TO_FD
/* Under certain circumstances, we must communicate with the server
via a socket using send() and recv(). This is because under some
operating systems (OS/2 and Windows 95 come to mind), a socket
cannot be converted to a file descriptor – it must be treated as a
socket and nothing else.

We may also need to deal with socket routine error codes differently
in these cases. This is handled through the SOCKERRNO and
SOCK_STRError macros. */

sifndef NO_SOCKET_TO_FD
static int unix_socket_style = 0;
static int server_sock;

500 /* These routines implement a buffer structure which uses send and
recv. The buffer is always in blocking mode so we don't implement
the block routine. */

/* Note that it is important that these routines always handle errors
internally and never return a positive errno code, since it would in
general be impossible for the caller to know in general whether any
error code came from a socket routine (to decide whether to use
SOCKSTERROR or simply strerror to print an error message). */

520 /* We use an instance of this structure as the closure field. */

struct socket_buffer
{
    /* The socket number. */
    int socket;
};

static struct buffer *socket_buffer Initialize
PROTO ((int, int, void (*)(struct buffer *)));

530 static int socket_buffer_input PROTO((void *, char *, int, int, int *));
static int socket_buffer_output PROTO((void *, const char *, int, int *));
static int socket_buffer_flush PROTO((void *));

/* Create a buffer based on a socket. */

static struct buffer *
socket_buffer Initialize (socket, input, memory)
    int socket;

int input;
void (*memory) PROTO((struct buffer *));
{
    struct socket_buffer *n;
    n = (struct socket_buffer *) malloc (sizeof *n);
    n->socket = socket;
    return buff_initialize (input, socket_buffer_input : NULL, input ? NULL : socket_buffer_output, input ? NULL : socket_buffer_close, (int (*)(*) PROTO(void *, int)) NULL, (int (*)(*) PROTO(void *)) NULL, memory, n);
}

/* The buffer input function for a buffer built on a socket. */
static int
socket_buffer_input (closure, data, need, size, got)
void *closure;
char *data;
int need;
int size;
int *got;
{
    struct socket_buffer *sb = (struct socket_buffer *) closure;
    int nbytes;

    /* I believe that the recv function gives us exactly the semantics
     we want. If there is a message, it returns immediately with
     whatever it could get. If there is no message, it waits until
     one comes in. In other words, it is not like read, which in
     blocking mode normally waits until all the requested data is
     available. */
    *got = 0;
    do {
        nbytes = recv (sb->socket, data, size, 0);
        if (nbytes < 0) error (1, 0, "reading from server: %s", SOCK_strerror (SOCK_ERRNO));
        if (nbytes == 0)
            { /* End of file (for example, the server has closed
               the connection). If we've already read something, we
               just tell the caller about the data, not about the end of
               file. If we've read nothing, we return end of file. */
                if (*got == 0)
                    return -1;
                else
                    return 0;
            }
        need = need - nbytes;
        size = size - nbytes;
        data += nbytes;
        *got += nbytes;
    } while (need > 0);
    return 0;
}

/* The buffer output function for a buffer built on a socket. */
static int
socket_buffer_output (closure, data, have, wrote)
void *closure;
const char *data;
int have;
int *wrote;
{
    struct socket_buffer *sb = (struct socket_buffer *) closure;

    *wrote = have;

    /* See comment in socket_buffer_input regarding buffer size we pass
to send and recv. */
}
ifdef SEND_NEVER_PARTIAL
  /* If send() never will produce a partial write, then just do it. This
     is needed for systems where its return value is something other than
     the number of bytes written. */
  if (send(ab->socket, data, have, 0) < 0)
    error (1, 0, "writing to server socket: %s", SOCK_strerror(SOCK_errno));
  else
    while (have > 0)
      { int nbytes;
        nbytes = send (ab->socket, data, have, 0);
        if (nbytes < 0)
          error (1, 0, "writing to server socket: %s", SOCK_strerror(SOCK_errno));
          have -= nbytes;
        data += nbytes;
      } sendif
  return 0;
}endif

#elif defined(CLIENT_SUPPORT) || defined(SERVER_SUPPORT)
  /* Zero if compression isn't supported or requested; non-zero to indicate
     a compression level to request from gzip. */
  int gzip_level;
  /* Level of compression to use when running gzip on a single file. */
  int file_gzip_level;
  /
  * The buffer flush function for a buffer built on a socket. */
  /*ARGSUSED*/
  static int socket_buffer_flush (closure)
    { /* Nothing to do. Sockets are always flushed. */
      return 0;
    } sendif /* NO_SOCKET_TO_FD */
  /* Read a line from the server. Result does not include the terminating \n.
   * Space for the result is malloc'd and should be freed by the caller.
   */
  /* Return number of bytes read. */
  static int read_line (resultp)
    { char *resultp;
      int status;
      char *result;
      int len;
      status = buf_flush (to_server, 1);
      if (status != 0)
        error (1, status, "writing to server");
      status = buf_read_line (from_server, &result, &len);
      if (status != 0)
        { if (status == -1)
            error (1, 0, "end of file from server (consult above messages if any)");
          else if (status == -2)
            error (1, 0, "out of memory");
          else
            error (1, status, "reading from server");
        }
      if (resultp != NULL)
        *resultp = result;
      else
        free (result);
      return len;
    } sendif /* CLIENT_SUPPORT */
#endif /* defined(CLIENT_SUPPORT) || defined(SERVER_SUPPORT) */
int filter_through_gzip (fd, dir, level, pidp)
{
    static char buf[5] = "-";
    static char *gzip argv[3] = { "gzip", buf };
    sprintf (buf+1, "%d", level);
    return filter_stream_through_program (fd, dir, &gzip argv[0], pidp);
}

int filter_through_gunzip (fd, dir, pidp)
{
    static char *gunzip argv[3] = { "gzip", "-d" };
    return filter_stream_through_program (fd, dir, &gunzip argv[0], pidp);
}

#endif /* CLIENT_SUPPORT or SERVER_SUPPORT */

static void handle_ok (args, len)
{
    return;
}

static void handle_error (args, len)
{
    int something_printed;
    char *p = strchr (args, ' ');
    if (p == NULL)
    {
        error (0, 0, "invalid data from cvs server");
        return;
    }
    ++p;
    len -= p - args;
    something_printed = 0;
    for (; len > 0; --len)
    {
        something_printed = 1;
        putc (*p++, stderr);
    }
    if (something_printed)
        putc (\n, stderr);
}

static void handle_valid_requests (args, len)
{
    char *p = args;
    char q;
    struct request *rq;
    do
    {
        q = strchr (p, ' ');
        if (q == NULL)
            rq++ = '\0';
        for (rq = requests; rq->name != NULL; ++rq)
        {
            if (strcmp (rq->name, p) == 0)
                break;
        }
    }
if (rq->name == NULL)
  /*
   * It is a request we have never heard of (and thus never
   * will want to use). So don't worry about it.
   * /
   */
  return 0;
else
  if (rq->status == rq->enableme)
    /*
     * Server wants to know if we have this, to enable the
     * feature.
     */
    send_to_server(rq->name, 0);
    send_to_server("request", 0);
  else
    rq->status = rq->supported;
  p = q;
}
while (q != NULL);
for (rq = requests; rq->name != NULL; ++rq)
  if (rq->status == rq->essential)
    error (1, 0, "request " rq->name " not supported by server");
else if (rq->status == rq->optional)
  rq->status = rq->not_supported;
}

/* This variable holds the result of Entries::Open, so that we can
  close Entries::Close on it when we move on to a new directory, or
  when we finish. */

static List *last_entries;

/*
 * Do all the processing for PATHNAME, where pathname consists of the
 * repository and the filename. The parameters we pass to FUNC are:
 * DATA is just the DATA parameter which was passed to
 * call_to_directory; ENT_LIST is a pointer to an entries list (which
 * we manage the storage for); SHORT_PATHTNAME is the pathname of the
 * file relative to the (overall) directory in which the command is
 * taking place; and PILENAME is the filename portion only of
 * SHORT_PATHNAME. When we call FUNC, the current directory points to
 * the directory portion of SHORT_PATHNAME. */

static char *last_dir_name;

static void
call_to_directory (pathname, func, data)
{
  char *pathname;
  void (*func) (char *data, List *ent_list, char *short_pathname,
              char *filename);

  char *data;

  char *dirname;
  char *filename;
  /* This is what we get when we hook up the directory (working directory
   * name) from PATHNAME with the filename from REPOSNAME. For example:
   * pathname: ccvs/src/
   * reposname: /u/src/master/ccvs/src/ChangeLog
   * short_pathname: ccvs/src/ChangeLog
   */

  char *short_pathname;
  char *p;

  /*
   * Do the whole descent in parallel for the repositories, so we
   * know what to put in CVS/Repository files. I'm not sure the
   * full hair is necessary since the server does a similar
   * computation; I suspect that we only end up creating one
   * directory at a time anyway.
   * *
   * Also note that we must only worry about this stuff when we
   * are creating directories; 'cvs co foo/bar; cd foo/bar; cvs co
   * CVSROOT; cvs update' is legitimate, but in this case
   * foo/bar/CVSROOT/CVS/Repository is not a subdirectory of
   * foo/bar/CVS/Repository.
   */

  char *reposname;

  char *short_repos;
  char *reposdirname;
  char *rdirp;
  int repos dirname absolute;

  reposname = NULL;
  realpath (&reposname);
  assert (reposname != NULL);
reposdirname\[absolute = 0);
900  if (strncmp (reposname, toplevel\[repos, strlen (toplevel\[repos)) != 0) {
901      reposdirname\[absolute = 1;
902      short\[repos = reposname;
903  } else {
904      short\[repos = reposname + strlen (toplevel\[repos) + 1;
905      if (short\[repos[-1] != '\n') {
906        reposdirname\[absolute = 1;
907        short\[repos = reposname;
908      }
909      reposdirname = xstrdup (short\[repos);
910      p = strchr (reposdirname, '\n');
911  if (p == NULL) {
912    reposdirname = xrealloc (reposdirname, 2);
913    reposdirname\[0] = '\n'; reposdirname\[1] = '\0';
914  } else {
915    p = '\0';
916    dir_name = xstrdup (pathname);
917    p = strchr (dir_name, '\n');
918  if (p == NULL) {
919    dir_name = xrealloc (dir_name, 2);
920    dir_name\[0] = '\n'; dir_name\[1] = '\0';
921  } else {
922    p = '\0';
923    if (errno == EISDIR)
924      add\[name\[and\]candidate (dir_name);
925    filename = strchr (short\[repos, '\n');
926  if (filename == NULL)
927    filename = short\[repos;
928  else {
929    ++filename;
930    short\[pathname = xmalloc (strlen (pathname) + strlen (filename) + 5);
931    strcpy (short\[pathname, pathname);
932    strcat (short\[pathname, filename);
933  }
934  }
935  if (last\[dir_name == NULL)
936     || strcmp (last\[dir_name, dir_name) != 0) {
937    int newdir;
938  if (strcmp (command\[name, 'export') != 0)
939    if (last\[entries)
940      Entries\[Close (last\[entries);
941    if (last\[dir_name)
942      free (last\[dir_name);
943    last\[dir_name = dir_name;
944  if (toplevel\[wd == NULL)
945    {
946      toplevel\[wd = xgetwd ();
947  if (toplevel\[wd == NULL)
948    error (1, errno, "could not get working directory");
949  }
950  if (CVS\[CHDIR (toplevel\[wd) < 0)
951    error (1, errno, "could not chdir to \"a\", toplevel\[wd);
952    newdir = 0;
953  } /* Create the CVS directory at the top level if needed. The
954  isdir seems like an unneeded system call, but it *does* need to be called both if the CVS\[CHDIR below succeeds
955  (e.g. "cvs co .") or if it fails (e.g. hasch-1a in
956  testsuite). We only need to do this for the "." case,
957  since the server takes care of forcing this directory to be
958  created in all other cases. If we don't create CVSADM
959  here, the call to Entries\[Open below will fail. FIXME:
960  perhaps this means that we should change our algorithm
961  below that calls Create\[Admin instead of having this code
962  here? */
963  if (reposname\[absolute case has to do with
964  things like "cvs update /foo/bar". In any event, the
965  code below which tries to put toplevel\[repos into
966  CVS/Repository is almost surely unsuited to
967  the reposname\[absolute case. */
968  reposname\[absolute
969  && (strcmp (dir_name, \".*\") == 0)
970  && ! isdir (CVSADM))
971  else
972      toplevel\[wd = xrealloc (toplevel\[wd + 1);
973    if (toplevel\[wd == NULL)
974      error (1, errno, "could not realloc working directory");
975  if (toplevel\[wd == NULL)
976    error (1, errno, "could not get working directory");
977  if (CVS\[CHDIR (toplevel\[wd) < 0)
978    error (1, errno, "could not chdir to \"a\", toplevel\[wd);
979    newdir = 0;
newdir = 1;
repo = xmalloc(strlen(toplevelrepos) + 10);
strcpy(repo, toplevelrepos);
r = repo + strlen(repo);
if ((r[-1] != '.' || r[-2] != '/')
strncpy(r, '/../');
CreateAdmin('.', '.', repo, (char *) NULL,
(char *) NULL, 0, 1);
free(repo);
}
if (CVS_CHDIR(dir_name) < 0)
{
     char *dir;
     char *dirp;
     if (!existence_error(errno))
     error(1, errno, "could not chdir to %s", dir_name);
     /* Directory does not exist, we need to create it. */
     newdir = 1;
     /* Provided we are willing to assume that directories get created one at a time, we could simplify this a lot.
     Do note that one aspect still would need to walk the dir_name path: the checking for "fncmp (dir, CVSADM)". */
     dir = xmalloc(strlen(dir_name) + 1);
dirp = dir_name;
rdirp = reposdirname;
/* This algorithm makes nested directories one at a time
     and createCVS administration files in them. For example, we're checking out foo/bar/baz from the repository:
     1) create foo, point CVS/Repository to </root>/foo
     2) .. foo/bar .. <root>/foo/bar
     3) .. foo/bar/baz .. <root>/foo/bar/baz
     As you can see, we're just stepping along DIR_NAME (with DIRP) and REPOSdirname (with RDIRP) respectively.
     We need to be careful when we are checking out a module, however, since DIR_NAME and REPOSdirname are not going to be the same. Since modules will not have any slashes in their names, we should watch the output of STRCHR to decide whether or not we should use STRCHR on the RDIRP. That is, if we're down to a module name, don't keep picking apart the repository directory name. */
do {
     dirp = strchr(dirp, '/');
     if (dirp)
     {
         strncpy(dir, dirp, dirp - dirp);
dirp[dirp = dir_name] = '/0';
         /* Skip the slash. */
         +dirp;
     if (rdirp == NULL)
         /* This just means that the repository string has fewer components than the dir_name string. But
that is OK (e.g. see modules3-8 in testsuite). */
     else
         rdirp = strchr(rdirp, '/');
     }
else
{
     /* If there are no more slashes in the dir name,
we're down to the most nested directory-OR- to
the name of a module. In the first case, we
should be down to a DIRP that has no slashes,
so it won't help/hurt to do another STRCHR call
on DIRP. It will definitely hurt, however, if
we're down to a module name, since a module
name can point to a nested directory (that is,
DIRP will still have slashes in it. Therefore,
we should set it to NULL so the routine below
copies the contents of REMOTEDIRNAME onto the
root repository directory (does this if rdirp

is set to NULL, because we used to do an extra
\texttt{STRCHR} call here). */

\begin{verbatim}
rdirp = NULL;
strcpy (dir, dirname);
}

if (!strcmp(dir, CVSADM) == 0)
{
    error (0, 0, "cannot create a directory named \texttt{\%s}\n", dir);
    error (0, 0, "because CVS uses \texttt{\%s} for its own uses",
CVSADM);
    error (1, 0, "rename the directory and try again");
}

if (mkdir if needed (dir))
{
    /* It already existed, fine. Just keep going. */
}
else if (strcmp (command name, "export") == 0)
    /* Don't create CVSADM directories if this is export. */
else
{
    /* Put repository in CVS/Repository. For historical
     * (pre-CVS/Root) reasons, this is an absolute pathname,
     * but what really matters is the part of it which is
     * relative to cvsroot. */
    char *repo;
    char *r, *b;
    repo = xmalloc(strlen(reposdirname)
                   + strlen(toplevelrepos) + 80);
    if (reposdirname absolute)
        r = repo;
    else
        {
            strcpy (repo, toplevelrepos);
            strcat (repo, "/");
            r = repo + strlen (repo);
        }
    if (rdirp)
    {
        /* See comment near start of function; the only
         way that the server can put the right thing
         in each CVS/Repository file is to create the
         directories one at a time. I think that the
         CVS server has been doing this all along. */
        error (0, 0, ",

        \texttt{warning: server is not creating directories one at a time}
        *);
        strncpy (r, reposdirname, rdirp - reposdirname);
        r[rdirp - reposdirname] = '\0';
    }
    else
        strcpy (r, reposdirname);
    CreateAdmin (dir, dir, repo,
(char *)[NULL], (char *)[NULL], 0, 0);
    free (repo);
    b = strrchr (dir, '/');
    if (b == NULL)
        SubdirRegister ((List *)[NULL], (char *)[NULL], dir);
    else
    {
        *b = '\0';
        SubdirRegister ((List *)[NULL], dir, b + 1);
        *b = '/';
    }
}
if (rdirp != NULL)
{
    /* Skip the slash. */
    ++rdirp;
}
}
while (dirp != NULL);
free (dir);
/* Now it better work. */
if (CVSomidir (dirname) < 0)
    error (1, errno, "could not chdir to \texttt{\%s}\", dirname);
}
if (strcmp (command name, "export") == 0)
{
last_entries = Entries_Open(0, dir_name);

/* If this is a newly created directory, we will record
all subdirectory information, so call SubdirsKnown in
case there are no subdirectories. If this is not a
newly created directory, it may be an old working
directory from before we recorded subdirectory
information in the Entries file. We force a search for
all subdirectories now, to make sure our subdirectory
information is up to date. If the Entries file does
record subdirectory information, then this call only
does list manipulation. */
if (newdir)
    Subdirs_Known(last_entries);
else
    {
        List *dirlist;
        dirlist = Find_Directories([char *] NULL, W_LOCAL,
            last_entries);
    }

static void
copy_file(list, short pathname, filename)
char *data;
char *ent_list;
char *short_pathname;
char *filename;

{ char *newname;
  ifdef USE_VMS_FILENAMES

      read_line(&newname);
  endif

  ifdef USE_VMS_FILENAMES
  /* Mogrify the filename so VMS is happy with it. */
  for(p = newname; *p; p++)
      if(*p == '.' || *p == '$') *p = '_';
  endif

  copy_file(filename, newname);
  free(newname);

  static void
handle_copy_file(args, len)
char *args;
int len;

{ call_in_directory(args, copy_file, [char *] NULL);
}

static void read_counted_file PROTO ( [char *], char * );

/* Read from the server the count for the length of a file, then read
the contents of that file and write them to FILENAME. FULLNAME is
the name of the file for use in error messages. FIXME-someday:
extend this to deal with compressed files and make update_entries
use it. On error, gives a fatal error. */
static void
read_counted_file(filename, fullname)
char *filename;
char *fullname;

{ char *size_string;
  size_t size;
  char *buf;

  /* Pointers in buf to the place to put data which will be read,
  and the data which needs to be written, respectively. */
  char *read;
  char *write;
  /* Number of bytes left to read and number of bytes in buf waiting to
  be written, respectively. */
  size_t nread;
  size_t nwrite;
FILE *fp;

read_line (&size_string);
if (size_string[0] == 'z')
    error (1, 0, " protocol error: compressed files not supported for that operation");
/* FIXME: should be doing more error checking, probably. Like using
    stderr and making sure we used up the whole line. */
size = atoi (size_string);
free (size_string);

/* A more sophisticated implementation would use only a limited amount
of buffer space (8K perhaps), and read that much at a time. We allocate
a buffer for the whole file only to make it easy to keep track what
needs to be read and written. */
buf = xmalloc (size);

/* FIXME-someday: caller should pass in a flag saying whether it
is binary or not. I haven't carefully looked into whether
CVS/Template files should use local text file conventions or
not. */

fp = CVS_OPEN (filename, "wb");
if (fp == NULL)
    error (1, errno, "cannot write %s", fullname);
nread = size;
fwrite = 0;
pread = buf;
pwrite = buf;
while (nread > 0 || nwrite > 0)
{
    size = n;
    if (nread > 0)
    {
        n = try_read_from_server (pread, nread);
        nread -= n;
pread += n;
write += n;
    }
    if (nwrite > 0)
    {
        n = fwrite (pwrite, 1, nwrite, fp);
        if (error (fp))
            error (1, errno, "cannot write %s", fullname);
        nwrite -= n;
pwrite += n;
    }
    free (buf);
    if (fclose (fp) < 0)
        error (1, errno, "cannot close %s", fullname);
}

/* OK, we want to swallow the "U foo.c" response and then output it only
if we can update the file. In the future we probably want some more
systematic approach to parsing tagged text, but for now we keep it
ad hoc. "Why," I hear you cry, "do we not just look at the
Update-existing and Created responses?" That is an excellent question,
and the answer is roughly conservatism/laziness I haven't read through
update.c enough to figure out the exact correspondence or lack thereof
between those responses and a "U foo.c" line (note that Merged, from
join_file, can be either "C foo" or "U foo" depending on the context). */
/* Nonzero if we have been +updated and not -updated. */
static int updated;
/* Filename from an "fname" tagged response within +updated/-updated. */
static char *updated_filename;

/* Nonzero if we should arrange to return with a failure exit status. */
static int failure_exit;

/ *
* The time stamp of the last file we registered.
* /
static time_t last_register_time;

/ *
* The Checksum response gives the checksum for the file transferred
* over by the next Updated, Merged or Patch response. We just store
* it here, and then check it in update_entries.
* /
static int stored_checksum_valid;
static unsigned char stored_checksum[16];

static void handle_checksum (args, len)
    char *args;
int len;

char *s;
char buf[3];
int i;

if (stored.checksum.valid)
    error (1, 0, "Checksum received before last one was used");

s = args;
buf[2] = '\0';

for (i = 0; i < 16; i++)
    {
        char *bufend;
        buf[0] = *s++;
        buf[1] = *s++;
        stored.checksum[i] = (char) strtol (buf, &bufend, 16);
        if (bufend != buf + 2)
            break;
    }

    if (i < 16 || *s != '\0')
        error (1, 0, "Invalid Checksum response: '%s'", args);

stored.checksum_valid = 1;
}

static int stored.mode_valid;
static char *stored.mode;

static void handle_mode PROTO ((char *, int));

static void handle_mode (args, len)
    char *args;
    int len;
    {
        if (stored.mode_valid)
            error (1, 0, "protocol error: duplicate Mode");
        if (stored.mode != NULL)
            free (stored.mode);
        stored.mode = xstrdup (args);
        stored.mode_valid = 1;
    }

    /* Nonzero if time was specified in Mod-time. */
    static int stored.modtime_valid;
    /* Time specified in Mod-time. */
    static time_t stored.modtime;

static void handle_mod_time PROTO ((char *, int));

static void handle_mod_time (args, len)
    char *args;
    int len;
    {
        if (stored.modtime_valid)
            error (0, 0, "protocol error: duplicate Mod-time");
        stored.modtime = getdate (args, NULL);
        if (stored.modtime == (time_t) -1)
            error (0, 0, "protocol error: cannot parse date %s", args);
        else
            stored.modtime_valid = 1;
    }

    /*
     * If we receive a patch, but the patch program fails to apply it, we
     * want to request the original file. We keep a list of files whose
     * patches have failed.
     *
     */

    char **failed_patches;
    int failed_patches_count;

struct update_entries_data
    {
        enum
            {
                /*
                 * We are just getting an Entries line; the local file is
                 * correct.
                 */
                UPDATE_ENTRIES_CHECKIN,
                /* We are getting the file contents as well. */
                UPDATE_ENTRIES_UPDATE,
                /*
                 * We are getting a patch against the existing local file, not
                 * an entire new file.
                 */
            }
UPDATE_ENTRIES_PATCH,
//
/* We are getting an RCS change text (diff -n output) against
 * the existing local file, not an entire new file.
 */
UPDATE_ENTRIES_RCS_DIFF
} contents;

enum {
  /* We are replacing an existing file. */
  UPDATE_ENTRIES_REPLACING,
  /* We are creating a new file. */
  UPDATE_ENTRIES_CREATING,
  /* We don't know whether it is existing or new. */
  UPDATE_ENTRIES_UNKNOWN_OR_NEW,
} existing;

/* Strong to put in the timestamp field or NULL to use the timestamp
 * of the file.
 */
char *timestamp;

/* Update the Entries line for this file. */
static void
update_entries (data_arg, ent_list, short_pathname, filename)
{
  char *dat_arg;
  List *ent_list;
  char *short_pathname;
  char *filename;

  struct update_entries_data *data = (struct update_entries_data *)data_arg;
  char *remote_file;
  char *cp;
  char *user;
  char *vn;

  /* Timestamp field. Always empty according to the protocol. */
  char *ts;

  char *options = NULL;
  char *tag = NULL;
  char *date = NULL;
  char *tag_or_date;
  char *scratch_entries = NULL;
  char *repository = NULL;
  int bin;

  ifdef UTIME_EXPECTS_WRITABLE
  int change_it_back = 0;
  endif

  read_line (&entries_line);

  /*
   * Parse the entries line.
   */
  scratch_entries = xstrdup (entries_line);

  if [scratch_entries[0] == ' ')
    error (1, 0, "bad entries line "ts" from server", entries_line);
  user = scratch_entries + 1;
  if [(cp = strchr (user, ' ')) == NULL)
    error (1, 0, "bad entries line "ts" from server", entries_line);
    cp++; = '\0';
  vn = cp;
  if [(cp = strchr (vn, ' ')) == NULL)
    error (1, 0, "bad entries line "ts" from server", entries_line);
    cp++; = '\0';
  ts = cp;
  if [(cp = strchr (ts, ' ')) == NULL)
    error (1, 0, "bad entries line "ts" from server", entries_line);
    cp++; = '\0';
  options = cp;
  if [(cp = strchr (options, ' ')) == NULL)
    error (1, 0, "bad entries line "ts" from server", entries_line);
    cp++; = '\0';
  tag_or_date = cp;
  if [(cp = strchr (tag_or_date, ' ')) == NULL)
    error (1, 0, "bad entries line "ts" from server", entries_line);
    cp++; = '\0';

  /* Note that the root doesn't come from the server, because the server
   * has no idea it's running as a server -- it thinks it's running
   * in a filesystem. That's hard to fix, so I won't. */

repository = cp;

/* If a slash ends the tag update, ignore everything after it. */
if (cp != NULL)
    *cp = '\0';
    if (*tag == 'D')
        tag = tag + 1;
    else if (*tag == 'U')
        date = tag + 1;
    else if (*tag == '?' 
    tag = tag + 1;
    else if (*tag == '?')

    /* Done parsing the entries line. */
    if (data->contents == UPDATE_ENTRIES_UPDATE
        || data->contents == UPDATE_ENTRIES_PATCH
        || data->contents == UPDATE_ENTRIES_RCS_DIFF
    {
        char *size_string;
        char *mode_string;
        int size;
        char *buf;
        char *filename;
        int use_gzip;
        int patch_failed;

        read_line (&mode_string);
        read_line (&size_string);
        if (*size_string[0] == 'x')
            use_gzip = 1;
        size = atoi (size_string+1);
    } else
        use_gzip = 0;
        size = atoi (size_string);
        free (size_string);

    /* Note that checking this separately from writing the file is
     * a race condition: if the existing or lack thereof of the
     * file changes between now and the actually calls which
     * operate on it, we lose. However (a) there are so many
     * cases, I'm reluctant to try to fix them all, (b) in some
     * cases the system might not even have a system call which
     * does the right thing, and (c) it isn't clear this needs to
     * work. */
    if (data->existp == UPDATE_ENTRIES_EXISTING
        && isfile (filename))
        /* Emit a warning and update the file anyway. */
        error (0, 0, "warning: %s unexpectedly disappeared",
            short_pathname);
        if (data->existp == UPDATE_ENTRIES_NEW
            && isfile (filename))
            /* Emit a warning and refuse to update the file; we don't want
             * to clobber a user's file. */
            size_t read;
            size_t toread;
            /* size should be unsigned, but until we get around to fixing
             * that, work around it. */
            size_t use;
            char *buf[8192];

        /* This error might be confusing; it isn't really clear to
         * the user what to do about it. Keep in mind that it has
         * several causes: (1) something/someone creates the file
         * during the time that CVS is running, (2) the repository
         * has two files whose names clash for the client because
         * of case-insensitivity or similar causes, (3) a special
         * case of this is that a file gets renamed for example
         * from A.c to A.c. A "cvs update" on a case-insensitive
         * client will get this error. Repeating the update takes
         * care of the problem, but is it clear to the user what
         * is going on and what to do about it?, (4) the client
         * has a file which the server doesn't know about (e.g. "foo"
         * file), and that name clashes with a file the
         * server does know about, (5) classify.c will print the same
         * message for other reasons.
         * I hope the above paragraph makes it clear that making this
         * clearer is not a one-line fix. */
        error (0, 0, "move away %s; it is in the way",
            short_pathname);
        if (updated_filename != NULL)
            {cvs_output (*c *, 0);}
cvs_output (updated_filename, 0);
cvs_output ("\n", 1);
}
return;
discard_file_and_return:
    /* Now read and discard the file contents. */
    usize = size;
    nread = 0;
    while (nread < usize)
    {
        toread = usize - nread;
        if (toread > sizeof buf)
            toread = sizeof buf;
        nread += try_read_from_server (buf, toread);
        if (nread == usize)
            break;
    }
free (mode_string);
free (entries_line);
/* The Mode, Mod-time, and Checksum responses should not carry
over to a subsequent Created (or whatever) response, even
in the error case. */
stored_mode_valid = 0;
if (stored_mode != NULL)
    free (stored_mode);
stored_modtime_valid = 0;
stored_checksum_valid = 0;
if (updated_filename != NULL)
{
    free (updated_filename);
    updated_filename = NULL;
}
return;
}
    temp_filename = xmalloc (strlen (filename) + 80);
#endif USE_VMS_FILENAMES
    /* A VMS rename of "blah.dat" to "foo" to implies a
destination of "foo.dat" which is unfortunate for CVS */
sprintf (temp_filename, "%s_new_.%s", filename);
#else
#endif POSIX_NO_TRUNC
    sprintf (temp_filename, "%s.9s", filename);
#endif /* POSIX_NO_TRUNC */
#endif USE_VMS_FILENAMES */
buf = xmalloc (size);
/* Some systems, like OS/2 and Windows NT, end lines with CRLF
instead of just LF. Format translation is done in the C
library I/O functions. Here we tell them whether or not to
convert - if this file is marked "binary" with the RCS -kb
flag, then we don't want to convert, else we do (because
CVS assumes text files by default). */

if (options)
    bin = !(strcmp (options, "-kb"));
else
    bin = 0;

if (data->contents == UPDATE_ENTRIES_RCSDIFF)
    /* This is an RCS change text. We just hold the change
text in memory. */
    else
    /* server error: gzip invalid with RCS change text */;
    read_from_server (buf, size);
}
else
{
    int fd;

    fd = CVS_OPEN (temp_filename,
        (O_WRONLY | O_CREAT | O_TRUNC
        | (bin ? OPEN_BINARY : 0)),
        0777);
    if (fd < 0)
    {
        /* I can see a case for making this a fatal error; for
a condition like disk full or network unreachable (for a file server), carrying on and giving an error on each file seems unnecessary. But if it is a permission problem, or some such, then it is entirely possible that future files will not have the same problem. */
error (0, errno, "cannot write %s", short_pathname);
goto discard_filename_and_return;
}
1720
if (size > 0)
{
    read_from_server (buf, size);
    if (buf != NULL)
        gunzip_and_write (fd, short_pathname, buf, size);
    else if (write (fd, buf, size) != size)
        error (1, errno, "writing %s", short_pathname);
}
1730
if (close (fd) < 0)
    error (1, errno, "writing %s", short_pathname);

/* This is after we have read the file from the net (a change from previous versions, where the server would send us "M U foo.c" before Update-existing or whatever), but before we finish writing the file (arguably a bug). The timing affects a user who wants status info about how far we have gotten, and also affects whether "U foo.c" appears in addition to various error messages. */
1740
if (updated_filename != NULL)
{
    cvs_output ("U", 0);
    cvs_output (updated_filename, 0);
    cvs_output ("a", 1);
    free (updated_filename);
    updated_filename = 0;
}
1750
patch_failed = 0;
1760
if (data->contents == UPDATE_ENTRIES.UPDATE)
{
    /* If we are fetching a remote file, then we put its contents in a temporary file inside the CVS admin directory, instead of blasting it over the real file in the working directory
    * temp file format is CVS/remote_rev_name
    */
    remote_file = xmalloc (strlen (filename) + strlen (CVSADM_REMOTE_TMP) + strlen (tag) + 10);
    sprintf (remote_file, "%s_%s_%s", CVSADM_REMOTE_TMP, tag, filename);
    if (fetching_remote)
    {
        name_file (temp_filename, remote_file);
    }
    else
    {
        name_file (temp_filename, filename);
    }
}else if (data->contents == UPDATE_ENTRIES.PATCH)
{
1770
    #define DONT_USE_PATCH
    /* Hmm. We support only Rcs-diff, and the server supports only PATCHed (or else it would have sent Rcs-diff instead). Fall back to transmitting entire files. */
    patch_failed = 1;
} else /* Use patch. */

int retcode;
struct stat s;
1780
backup = xmalloc (strlen (filename) + 5);
strcpy (backup, filename);
stat (backup, &s);
(void) unlink_file (backup);
if (isfile (filename))
    error (1, 0, "patch original file %s does not exist", short_pathname);
if (S_ISDIR (temp_filename, &s) < 0)
    error (1, errno, "can't stat patch file %s", temp_filename);
1790
if (s.st_size == 0)
    retcode = 0;
else
    /* This behavior (in which -h takes an argument) is supported by GNU patch 2.1. Apparently POSIX.2 specifies a -h option without an argument. GNU patch 2.1.5 implements this and therefore won't work here. GNU patch versions after 2.1.5 are said to have a kludge which checks if the last 4 args
are 'b SUFFIX ORIGFILE PATCHFILE' and if so omit a
warning (I think -s suppresses it), and then behave
as CVS expects.

Of course this is yet one more reason why in the long
run we want RCS-diff to replace Patched. */
run_setup (PATCH_PROGRAM);
run_arg ("-o");
run_arg ("-a");
run_arg ("-p");
run_arg ("-s");
run_arg (filename);
run_arg (temp_filename):
retcode = run_sec (DEVNULL, RUNTTY, RUNTTY, RUN_NORMAL);
} /* FIXME: should we really be silently ignoring errors? */
(void) unlink_file (temp_filename);
if (retcode == 0)
{
} /* FIXME: should we really be silently ignoring errors? */
(void) unlink_file (backup);
} else
{
  int old_errno = errno;
  char *patch_imp;
  if (istfile (backup))
    rename_file (backup, filename);
  else
    { /* Get rid of the patch reject file. */
      path_imp = xmalloc (strlen (filename) + 10);
      strcpy (path_imp, filename);
      strcat (path_imp, "~");
    } /* FIXME: should we really be silently ignoring errors? */
    (void) unlink_file (path_imp);
    free (path_imp);
    error (retcode == -1 ? 1 : 0, retcode == -1 ? old_errno : 0,
      "could not patch %s", filename);

retcode = -1 ? "will refetch":
patch_failed = 1;
} free (backup);
} /* Use patch. */
else
{
  char *filebuf;
  size_t filebufsize;
  size_t nread;
  char *patchedbuf;
  size_t patchedlen;

  /* Handle UPDATE ENTRIES RCSDIFF. */
  if (istfile (filename))
    error (1, 0, "patch original file %s does not exist", 
      short_pathname);

  filebuf = NULL;
  filebufsize = 0;
  nread = 0;
  getfile (filename, short_pathname, bin ? OPEN_BINARY_READ : "r", 
    &filebuf, &filebufsize, &nread);
  /* At this point the contents of the existing file are in 
   FILEBUF, and the length of the contents is in NREAD.
   The contents of the patch from the network are in BUF,
   and the length of the patch is in SIZE. */

if (! rschangenext (short_pathname, filebuf, nread, buf, size, 
  &patchedbuf, &patchedlen))
  patch_failed = 1;
else
  { /* We have a checksum. Check it before writing
       the file out, so that we don't have to read it
       back in again. */
    unsigned char checksum[16];
    if (! stored_checksum_valid)
      { /*
         struct MD5Context context;
         unsigned char checksum[16];

We have a checksum. Check it before writing
the file out, so that we don't have to read it
back in again. */
    MD5Init (&context);
    MD5Update (&context, (unsigned char *) patchedbuf, patchedlen);
    MD5Final (checksum, &context);
    if (memcmp (checksum, stored_checksum, 16) != 0)
error (0, 0,
   "checksum failure after patch to %s; will refetch",
   short_pathname);

   stored_checksum_valid = 0;
}

if (!patch_failed)
{
  FILE *e;
  e = open_file (temp_filename,
      bin ? OPEN_BINARY_WRITE : "r");
  if (fwrite (patchedbuf, 1, patchedlen, e) != patchedlen)
    error (1, errno, "cannot write %s", temp_filename);
  if (fclose (e) == EOF)
    error (1, errno, "cannot close %s", temp_filename);
  rename_file (temp_filename, filename);
}

free (patchedbuf);
}

free (filebuf);
}

free (temp_filename);

if (stored_checksum_valid && !patch_failed)
{
  FILE *e;
  struct MD5Context context;
  unsigned char buf[1024];
  unsigned len;
  unsigned char checksum[16];

  /*
   * Compute the MD5 checksum.  This will normally only be
   * used when receiving a patch, so we always compute it
   * here on the final file, rather than on the received
   * data.
   * *
   * Note that if the file is a text file, we should read it
   * here using text mode, so its lines will be terminated the same
   * way they were transmitted.
   *
   */
  e = CVS_FOPEN (filename, "w");
  if (e == NULL)
    error (1, errno, "could not open %s", short_pathname);

  MD5Init (&context);
  while ((len = fread (buf, 1, sizeof buf, e)) > 0)
    MD5Update (&context, buf, len);
  if (ferror (e))
    error (1, errno, "could not read %s", short_pathname);
  MD5Final (checksum, &context);

  fclose (e);

  stored_checksum_valid = 0;

  if (memcmp (checksum, stored_checksum, 16) != 0)
  {
    if (data->content) UPDATE_ENTRIES_PATCH
      error (1, 0, "checksum failure on %s",
            short_pathname);

    error (0, 0,
       "checksum failure after patch to %s; will refetch",
       short_pathname);

    patch_failed = 1;
  }
}

if (patch_failed)
{
  /* Save this file to retrieve later.  */
  if
    failed_patches = (char **) xrealloc (char *) failed_patches,

    ((failed_patches_count + 1)
     * sizeof (char *));
  failed_patches[failed_patches_count] = xstrdup (short_pathname);
  ++failed_patches_count;
  stored_checksum_valid = 0;

  free (mode_string);
}
free (buf);
    }
    return;
  }

  int status = change_mode (filename, mode, string, t);
  if (status != 0)
    error (0, status, "cannot change mode of %s", short_pathname);
}

free (mode, string);
free (buf);
}

if (stored_mode_valid)
  change_mode (filename, stored_mode, 1);
stored_mode_valid = 0;

if (stored_modtime_valid)
{
  struct utimbuf t;
  memset (&t, 0, sizeof (t));
  /* There is probably little point in trying to preserved the
   * actime for is it there? What about Checked-in? */
  t.modtime = t.actime = stored_modtime;

  ifdef UTIME_EXPECTS_WRITABLE
  if (iswritable (filename))
    { xchmod (filename, 1);
      change_to_back = 1;
    }
  endif /* UTIME_EXPECTS_WRITABLE */

  if (utime (filename, &t) < 0)
    error (0, errno, "cannot set time on %s", filename);

  ifdef UTIME_EXPECTS_WRITABLE
  if (change_to_back == 1)
    { xchmod (filename, 0);
      change_to_back = 0;
    }
  endif /* UTIME_EXPECTS_WRITABLE */

  stored_modtime_valid = 0;
}

  /* Process the entries line. Do this after we've written the file,
   * since we need the timestamp.
   */
  if (strcmp (command_name, "export") != 0)
  {
    char *local_timestamp;
    char *file_timestamp;

    (void) time (&local_timestamp_time);

    local_timestamp = data->timestamp;
    if (local_timestamp == NULL || ts[0] == '+')
      file_timestamp = "dummy timestamp";
    else
      file_timestamp = NULL;

    /* These special version numbers signify that it is not up to
     * date. Create a dummy timestamp which will never compare
     * equal to the timestamp of the file.
     */
    if (vn[0] == '\0' || vn[0] == '0' || vn[0] == '-')
      local_timestamp = "dummy timestamp";
    else if (local_timestamp == NULL)
      { local_timestamp = file_timestamp;
        mark_timestamp (filename);
      }
  }

  /* Since we didn't get root from the server (as noted above)
   * the root we write out to the entries file is what we think the root is
   * at this point */
  /* Don't touch entries if we are fetching a remote rev */
  if (client_active || fetching_revision)
  { Register (ent, list, filename, vn, local_timestamp,
             options, tag, date, ts[0] == '+' ? file_timestamp : NULL, CVSroot_original.repository);
  } else if (fetching_revision)
  { FILE* remote_special_file = fopen (CVSADMREMOTES, "a+");
    if (remote_special_file != NULL) {
static void
handle_checked_in (args, len)
    char *args;
    int len;
    {
        struct update_entry data dat;
        dat.contents = UPDATE_ENTRIES_CHECKIN;
        dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
        dat.timestamp = NULL;
        call_in_directory (args, update_entries, (char *) &dat);
    }

static void
handle_new_entry (args, len)
    char *args;
    int len;
    {
        struct update_entry data dat;
        dat.contents = UPDATE_ENTRIES_CHECKIN;
        dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
        dat.timestamp = "dummy timestamp from new-entry";
        call_in_directory (args, update_entries, (char *) &dat);
    }

static void
handle_updated (args, len)
    char *args;
    int len;
    {
        struct update_entry data dat;
        dat.contents = UPDATE_ENTRIES_UPDATE;
        dat.existp = UPDATE_ENTRIES_NEW;
        dat.timestamp = NULL;
        call_in_directory (args, update_entries, (char *) &dat);
    }

static void
handle_created Proto((char *, int));

static void
handle_created (args, len)
    char *args;
    int len;
    {
        struct update_entry data dat;
        dat.contents = UPDATE_ENTRIES_UPDATE;
        dat.existp = UPDATE_ENTRIES_NEW;
        dat.timestamp = NULL;
        call_in_directory (args, update_entries, (char *) &dat);
    }

static void
handle_updated existing Proto((char *, int));

static void
handle_updated existing (args, len)
    char *args;
    int len;
    {
        struct update_entry data dat;
        dat.contents = UPDATE_ENTRIES_UPDATE;
        dat.existp = UPDATE_ENTRIES_EXISTING;
        dat.timestamp = NULL;
        call_in_directory (args, update_entries, (char *) &dat);
    }

static void
handle_merged (args, len)
    char *args;
    int len;
    {
        struct update_entry data dat;
        /* Think this could be UPDATE_ENTRIES_EXISTING, but just in case... */
        dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
        dat.timestamp = "result of merge";
        call_in_directory (args, update_entries, (char *) &dat);
    }

return;

fputs (remote_rev, file, "%s/\%d\n", filename, vn, strchr (remote_rev, '/') + 1);
fclose (remote_rev, file);

if (file_timestamp)
    free (file_timestamp);
free (scratch_entries);
free (entries_line);
}
static void
handle_patched (args, len)
{
  int len;

  struct update_entries data;
  dat.entries = UPDATE_ENTRIES_PATCH;
  /
  * Think this could be UPDATE_ENTRIES_EXISTING, but just in case... */
  dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
  dat.contents = NULL;
  open_directory (args, update_entries, (char *) & dat);
}

static void
handle_rsdiff (args, len)
{
  struct update_entries data;
  dat.entries = UPDATE_ENTRIES_RCS_DIFF;
  /
  * Think this could be UPDATE_ENTRIES_EXISTING, but just in case... */
  dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
  dat.contents = NULL;
  open_directory (args, update_entries, (char *) & dat);
}

/* Check this could be UPDATE_ENTRIES_EXISTING, but just in case... */
static int
check_entries (ent, pathname, (char *) & ent);

remove_entry (data, ent_list, short_pathname, filename)
{
  char * data;
  List * ent_list;
  char * short_pathname;
  char * filename;
  
  Scratch_Entry (ent_list, filename);
}

remove_entry_and_file (data, ent_list, short_pathname, filename)
{
  char *data;
  List *ent_list;
  char *short_pathname;
  char *filename;
  
  Scratch_Entry (ent_list, filename);
  /
  * Note that we don't ignore existence error's here. The server
  should be sending Remove-Entry-rather than Removed in cases
  where the file does not exist. And if the user removes the
  file halfway through a cvs command, we should be printing an
  error. */
  if (unlink_file (filename) < 0)
    error (0, errno, "Unable to remove %s", short_pathname);
}

is_root_level (pathname)
{
  char * pathname;

  if (strcmp ((toplevel_repos, CVSRoot_directory) != 0)
    return 0;

  return strchr (pathname, '/') == NULL;
}

set_static (data, ent_list, short_pathname, filename)
{
  FILE *fp;
  fp = open_file (CVSADM,ENTSTAT, "w");
  if (fclose (fp) == EOF)
error (1, errno, "cannot close %s", CVSADM\_ENTSTAT);

static void
directory (args, len)
char *args;
int len;
{
  if (strcmp (command, "export") == 0)
  {
    /* Swallow the repository. */
    read_line (NULL);
    return;
  }

call_in_directory (args, set\_static, (char *)NULL);
}

static void
clear\_static (data, ent\_list, short\_pathname, filename)
char *data;
List *ent\_list;
char *short\_pathname;
char *filename;
{
  if (unlink_file (CVSADM\_ENTSTAT) < 0 &
      existence_error (errno))
    error (1, errno, "cannot remove file %s", CVSADM\_ENTSTAT);
}

static void
directory (pathname, len)
char *pathname;
int len;
{
  if (strcmp (command, "export") == 0)
  {
    /* Swallow the repository. */
    read_line (NULL);
    return;
  }

  if (is\_cvsroot\_level (pathname))
  {
    /*
      * Top level (directory containing CVSROOT). This seems to normally
      * lack a CVS directory, so don't try to create files in it.
      */
    return;
  }

call_in_directory (pathname, clear\_static, (char *)NULL);
}

static void
set\_sticky (data, ent\_list, short\_pathname, filename)
char *data;
List *ent\_list;
char *short\_pathname;
char *filename;
{
  char *tagspec;
  FILE *f;

  read_line (&tagspec);
  f = open\_file (CVSADM\_TAG, "w");
  if (fprintf (f, "%s\n", tagspec) < 0)
    error (1, errno, "writing %s", CVSADM\_TAG);
  if (fclose (f) == EOF)
    error (1, errno, "closing %s", CVSADM\_TAG);
  free (tagspec);
}

static void
directory (pathname, len)
char *pathname;
int len;
{
  if (strcmp (command, "export") == 0)
  {
    /* Swallow the repository. */
    read_line (NULL);
    read_line (NULL);
    return;
  }

  if (is\_cvsroot\_level (pathname))
  {
    /*
      * Top level (directory containing CVSROOT). This seems to normally
      * lack a CVS directory, so don't try to create files in it.
      */
  }
/* Swallow the repository. */
read_line(NULL);

/* Swallow the tag line. */
read_line(NULL);
return;
}
callin_directory (pathname, setsticky, (char *)NULL);
}

static void
clear_sticky (data, ent_list, short pathname, filename)
char *data;
List *ent_list;
char *short pathname;
char *filename;
if (unlink_file (CVSADM_TAG) < 0 && ! existence error (errno))
error (1, errno, "cannot remove %s", CVSADM_TAG);
}

static void
handle_clear_sticky (pathname, len)
char *pathname;
int len;
if (strcmp (command_name, "export") == 0)
{ /* Swallow the repository. */
read_line (NULL);
return;
}

if (is_cvsroot_level (pathname))
{ /*
 * Top level (directory containing CVSROOT). This seems to normally
 * lack a CVS directory, so don't try to create files in it.
 */
return;
}
callin_directory (pathname, clear_sticky, (char *)NULL);
}

static void template_PROTO ((char *, List *, char *, char *));
static void
template (data, ent_list, short pathname, filename)
char *data;
List *ent_list;
char *short pathname;
char *filename;
{ /* FIXME: should be computing second argument from CVSADM TEMPLATE
    and short pathname. */
read_counted_file (CVSADM TEMPLATE, "CVS/Template file");
}

static void handle_template_PROTO ((char *, int));
static void
handle_template (pathname, len)
char *pathname;
int len;
{ callin_directory (pathname, template, NULL);
}

struct save_prog {
char *name;
char *dir;
struct save_prog *next;
};
static struct save_prog *checking_prgs;
static struct save_prog *update_prgs;

/*
* Unlike some responses this doesn't include the repository. So we can't
* just call callin_directory and have the right thing happen; we save up
* the requests and do them at the end.
*/

static void
handle_set_checkin_prog (args, len)
char *args;
int len;
{
char *prog;
struct save_prog *p;
read_line(&prog);
p = (struct save_prog *) xmalloc (sizeof (struct save_prog));
p->next = checking_progs;
p->dir = xtrdup (args);
p->name = prog;
checking_progs = p;
}

static void
do_deferred_progs (args, len)
{
char *args;
int len;
char *prog;
struct save_prog *p;
read_line(&prog);
p = (struct save_prog *) xmalloc (sizeof (struct save_prog));
p->next = update_progs;
p->dir = xtrdup (args);
p->name = prog;
update_progs = p;
}

static void
do_deferred_progs ()
{

}

for (p = checking_progs; p != NULL; )
{
fname = xmalloc (strlen (p->dir) + sizeof CVSADM_CIPROG + 10);
sprintf (fname, "%s%c%s", p->dir, CVSADM_CIPROG);
t = open_file (fname, "w");
if (!fputst (t, "\n", p->name) < 0)
error (1, errno, "writing %s", fname);
if (fclose (t) != EOF)
error (1, errno, "closing %s", fname);
free (p->name);
free (p);
p = q;
free (fname);
free (sname);
for (p = update_progs; p != NULL; )
{
fname = xmalloc (strlen (p->dir) + sizeof CVSADM_UPROG + 10);
sprintf (fname, "%s%c%s", p->dir, CVSADM_UPROG);
t = open_file (fname, "w");
if (!fputst (t, "\n", p->name) < 0)
error (1, errno, "writing %s", fname);
if (fclose (t) != EOF)
error (1, errno, "closing %s", fname);
free (p->name);
free (p->dir);
q = p->next;
free (p);
p = q;
free (fname);
update_progs = NULL;
}

struct save_dir {
    char *dir;
    struct save_dir *next;
};

struct save_dir *prune_candidates;

static void
add_prune_candidate (dir)
{
    struct save_dir *p;

if ((dir[0] == '.' & & dir[1] == '\0') || (prune_candidates != NULL & & strcmp (dir, prune_candidates->dir) == 0))
    return;
  p = (struct save_dir *) xmalloc (sizeof (struct save_dir));
  p->dir = xstrdup (dir);
  p->next = prune_candidates;
  prune_candidates = p;
}

static void process_prune_candidates PROTO((void));

static void
process_prune_candidates (
{
  struct save_dir *p;
  struct save_dir *q;

  if (toplevelwd != NULL)
    if (CVS_CHDIR (toplevelwd) < 0)
      error (1, errno, "could not chdir to %s", toplevelwd);
    for (p = prune_candidates; p != NULL; )
      if (!isemptydir (p->dir, 1))
        char *b;
        if (unlink_file (p->dir) < 0)
          error (6, errno, "cannot remove %s", p->dir);
        b = strchr (p->dir, '/');
        if (b == NULL)
          Subdir_Deregister ((List *) NULL, (char *) NULL, p->dir);
        else
          *b = '\0';
          Subdir_Deregister ((List *) NULL, p->dir, b + 1);
      }
    free (p->dir);
    q = p->next;
    free (p);
    p = q;
  }
  prune_candidates = NULL;
}

/* Send a Repository line. */

static char *last_repos;
static char *last_update_dir;

static void send_repository PROTO((char *, char *, char *));

static void
send_repository (dir, repos, update_dir)
{
  char *dir;
  char *repos;
  char *update_dir;

  char *adm_name;

  /* FIXME: this is probably not the best place to check; I wish I
   * knew where in here's callers to really trap this bug. To
   * reproduce the bug, just do this:
   *  mkdir junk
   *  cd junk
   *  cvs -d some/repos update foo
   *  Poof, CVS seq faults and dies! It's because it's trying to
   *  send a NULL string to the server but dies in sendUpdate_server.
   *  That string was supposed to be the repository, but it doesn't
   *  get set because there's no CVSADM dir, and somehow it's not
   *  getting set from the -d argument either... ?
   */
  if (repos == NULL)
    (/* Lame error. I want a real fix but can't stay up to track
       this down right now. */
    error (1, 0, "no repository");

  if (update_dir == NULL || update_dir[0] == '\0')
    update_dir = *p;
  if (last_repos != NULL & & strcmp (repos, last_repos) == 0 & & last_update_dir != NULL
    /* */
& & strcmp (update_dir, last_update_dir) == 0
/* We've already sent it. */
return;

if (client_errno != 0)
    add_server_comment (update_dir);
/* 80 is large enough for any of CVSADM. */
adm_name = xmalloc (strlen (dir) + 80);
send_to_server ("Directory ", 0);

    /* Send the directory name. I know that this sort of duplicates code elsewhere, but each case seems slightly different. */
    char buf[80];
    char *p = update_dir;
    while (*p != '\0')
        { assert (*p != '\012');
            if (ISDIRSEP (*p))
                { buf[0] = '\0';
                    send_to_server (buf, 1);
                } else
                    { buf[0] = *p;
                        send_to_server (buf, 1);
                    }
            ++p;
        }
    send_to_server ("\012", 1);
    send_to_server (repos, 0);
    send_to_server ("\012", 1);
    if (supported_request ("Static-directory"))
        { adm_name[0] = '\0';
            if (dir[0] != '\0')
                { strcat (adm_name, dir);
                    strcat (adm_name, "\012");
                } strcat (adm_name, CVSADM_ENTSTAT);
            if (isreadable (adm_name))
                { send_to_server ("Static-directory\012", 0);
                }
        }
    if (supported_request ("Sticky"))
        { FILE *f;
            if (dir[0] != '\0')
                fprintf (adm_name, CVSADM_TAG);
            else
                sprintf (adm_name, "%ld/", dir, CVSADM_TAG);
            if (f = CVS_OPEN (adm_name, "r"))
                { if (! f)
                        error (1, errno, "reading \%s", adm_name);
                    } else
                        { char line[80];
                            char *nl = NULL;
                            send_to_server ("Sticky ", 0);
                            while ((gets (line, sizeof (line), f)) != NULL)
                                { send_to_server (line, 0);
                                    nl = strchr (line, '\n');
                                    if (nl != NULL)
                                        break;
                                } if (nl == NULL)
                                    send_to_server ("\012", 1);
                                if (fclose (f) == EOF)
                                    error (0, errno, "closing \%s", adm_name);
                    }
    if (supported_request ("Checkin-prog"))
        { FILE *f;
            if (dir[0] != '\0')
                fprintf (adm_name, CVSADM_CIPROG);
            else
                sprintf (adm_name, "%ls/", dir, CVSADM_CIPROG);
```c
#define CVS_FOPEN (adm_name, "r")
if (f == NULL)
{
    if (l existence error (errno))
        error (l, errno, "reading %s", adm_name);
} else
{
    char line[80];
    char *nl = NULL;

    send_to_server ("Checkin-prog *", 0);
    while ((gets (line, sizeof (line)), f) != NULL)
    {
        send_to_server (line, 0);
        nl = strchr (line, '\n');
        if (nl != NULL)
            break;
    }
}
if (supported_request ("Update-prog"))
{
    FILE *f;
    if ((dir)[0] == '\0')
        strcpy (adm_name, CVSADM_UPROG);
    else
        sprintf (adm_name, "%s/%s", dir, CVSADM_UPROG);

    f = CVS_FOPEN (adm_name, "r");
    if (f == NULL)
    {
        if (l existence error (errno))
            error (l, errno, "reading %s", adm_name);
    } else
    {
        char line[80];
        char *nl = NULL;

        send_to_server ("Update-prog *", 0);
        while ((gets (line, sizeof (line)), f) != NULL)
        {
            send_to_server (line, 0);
            nl = strchr (line, '\n');
            if (nl != NULL)
                break;
        }
    }
}
free (adm_name);
if (lastrepos == NULL)
    free (lastrepos);
if (lastupdate_dir == NULL)
    free (lastupdate_dir);
lastrepos = xstrdup (repos);
lastupdate_dir = xstrdup (update_dir);
}

/* Send a Repository line and set toplevel repos. */

void
send_a_repository (dir, repository, update_dir)
char *dir;
char *repository;
char *update_dir;
{
    if (toplevelrepos == NULL && repository != NULL)
    {
        if (update_dir[0] == '\0'
            || (update_dir[0] == '.' && update_dir[1] == '\0'))
            toplevelrepos = xstrdup (repository);
        else
        {
            /* Get the repository from a CVS/Repository file if update_dir
             * is absolute. This is not correct in general, because
             */
```
* the CVS/Repository file might not be the top-level one.
* This is for cases like "cvs update ./foo/bar" (I'm not sure it matters what toplevel/repos we get, but it does
* matter that we don't hit the "internal error" code below).
*/
if (update_dir[0] == '/')
toplevel/repos = FindRepository (update_dir, update/dir);
else
{
    /*
    * Guess the repository of that directory by looking at
    * subdirectory and removing as many pathname components
    * as are in update/dir. I think that will always (or at
    * least almost always) be 1.
    *
    * So this deals with directories which have been
    * renamed, though it doesn't necessarily deal with
    * directories which have been put inside other
    * directories (and cvs invoked on the containing
    * directory). I'm not sure the latter case needs to
    * work.
    */
    /*
    * This gets toplevel/repos wrong for "cvs update ./foo"
    * but I'm not sure toplevel/repos matters in that case.
    */
    int slashes_in_update_dir;
    int slashes_skipped;
    char *p;
    /*
    * Strip trailing slashes from the name of the update directory.
    * Otherwise, running 'cvs update dir/' provokes the failure
    * "protocol error: illegal directory syntax in dir/" when
    * running in client/server mode.
    */
    strip_trailing_slashes (update_dir);
    slashes_in_update_dir = 0;
    for (p = update_dir; *p != '\0'; ++p)
        if (*p == '/')
            ++slashes_in_update_dir;
    slashes_skipped = 0;
    p = repository + strlen (repository);
    while (1)
    {
        if (p == repository)
            error (1, 0,
                "internal error: not enough slashes in %s",
                repository);
        if (*p == '/')
            ++slashes_skipped;
        if (slashes_skipped < slashes_in_update_dir + 1)
            --p;
        else
            break;
    }
    toplevel/repos = xmalloc (p - repository + 1);
    /* Note that we don't copy the trailing '/'. */
    strncpy (toplevel/repos, repository, p - repository);
    toplevel/repos[p - repository] = '\0';
}
}

*/ The "expanded" modules. */
static int modules_count;
2860 static int modules_allocated;
static char *modules_vector;

static void handle_module_expansion (args, len)
char *args;
int len;
{
    if (modules_vector == NULL)
    {
        modules_allocated = 1; /* Small for testing */
        modules_vector = (char **) xmalloc (modules_allocated * sizeof (modules_vector[0]));
    }
    else if (modules_count >= modules_allocated)
    {
        modules_allocated *= 2;
        modules_vector = (char **) xrealloc ((char *) modules_vector,
        modules_allocated * sizeof (modules_vector[0]));
    }
}
static int  
int i;

module_argv = argv;

module_argv = (char **) xmalloc ((argc + 1) * sizeof (module_argv[0]));
for (i = 0; i < argc; ++i)
module_argv[i] = xstrdup (argv[i]);
module_argv[argc] = NULL;
for (i = 0; i < argc; ++i)
send_repo ([arg[0]];
send_repo (argv[i]);

send_repo (argv[i]);

err = get_server_responses ();
if (lastupdate != NULL)
free (lastupdate);
lastupdate = NULL;
if (lastupdate != NULL)
free (lastupdate);
lastupdate = NULL;
if (err)
send_repo (argv[i]);

send_repo ([arg[0]];

send_repo (argv[i]);

for (i = 0; i < argc; ++i)

if (isfile ([arg[0]]))
send_repo (i, argv, local, build_dirs ? SEND_BUILDDIRS : 0);
send_repo (**, CVSroot\directory, **);
void
client_send_expansions (local, where, build_dirs)
int local;
char *where;
int build_dirs;
void
client_nonexpanded_setup ()
{ }
/* Enforce the notes in cvsclient.texi about how the response is not as free-form as it looks. */
p = strchr (args, ' ');
if (p == NULL)
    goto handle_error;

/* Add server-side cvswrappers line to our wrapper list. */
wrap_add (args, 0);
return;

handle_error:
    error (0, errno, "protocol error: ignoring invalid wrappers %s", args);
}

static void handle_not_carried (args, len)
    char *args;
    int len;
{
    char* server;
    char* root;
    char* repository;
    char* file = xmalloc (len + 1);
    strcpy (file, args);

    read_line (&server);
    read_line (&root);
    read_line (&repository);
    add_remote (file, server, root, repository);

    free (server);
    free (root);
    free (file);
    free (repository);
}

static void handle_create_remote_branch (args, len)
    char *args;
    int len;
{
    char* server;
    char* root;
    char* repository;
    char* revision;
    char* file = xmalloc (len + 1);

    read_line (&server);
    read_line (&root);
    read_line (&repository);
    read_line (&revision);
    strcpy (file, args);

    add_remote_tag (file, server, root, repository, revision);
}

static void handle_m (args, len)
    char *args;
    int len;
{
    /* In the case where stdout and stderr point to the same place, */
    /* fflushing stderr will make output happen in the correct order. */
    /* Often stderr will be line-buffered and this won't be needed, */
    /* but not always (is that true? I think the comment is probably */
    /* based on being confused between default buffering between */
    /* stdout and stderr. But I'm not sure). */
    /* fflush (stderr); */
    fwrite (args, len, sizeof (*args), stdout);
    putc ('\n', stdout);
}

static void handle_mbinary PROTO ((char *, int));
```c
int len;
char *size_string;
size_t size;
size_t totalread;
size_t nread;
char buf[8192];

/* See comment at handle_m about (non)flush of stderr. */

/* Get the size. */
readline (&size_string);
size = atoi (size_string);
free (size_string);

/* OK, now get all the data. The algorithm here is that we read as much as
the network wants to give us in try_read_from_server, and then we output it all, and then
repeat, until we get all the data. */
while (totalread < size)
{
    toread = size - totalread;
    if (toread > sizeof buf)
        toread = sizeof buf;
    nread = try_read_from_server (buf, toread);
    cvs_output_binary (buf, nread);
    totalread += nread;
}

static void handle_e (args, len)
char *args;
int len;
{
    /* In the case where stdout and stderr point to the same place,
    fflushing stdout will make output happen in the correct order. */
    fflush (stdout);
    fwrite (args, len, sizeof (*args), stderr);
    putc ('
', stderr);
}

static void handle_f (args, len)
char *args;
int len;
{
    fflush (stderr);
}

static void handle_mt PROTO ((char *, int));

static void handle_mt (args, len)
char *args;
int len;
{
    char *p;
    char *tag = args;
    char *text;
    /* See comment at handle_m for more details. */
    fflush (stderr);
    p = strchr (args, ' ');
    if (p == NULL)
        text = NULL;
    else
    {
        *p++ = '0';
        text = p;
    }
    switch (tag[0])
    {
        case '+':
            if (strcmp (tag, "+updated") == 0)
                updated_seen = 1;
            break;
        case '-':
            if (strcmp (tag, "-updated") == 0)
                updated_seen = 0;
            break;
        default:
            if (updated_seen)
            {
```
if (strncmp (tag, "fname") == 0)
{
    if (updated_fname != NULL)
    {
        /* Output the previous message now. This can happen
         * if there was no Update-existing or other such
         * response, due to the -n global option. */
        cvs_output (**, 0);
        cvs_output (updated_fname, 0);
        cvs_output ("\n", 1);
        free (updated_fname);
    }
    updated_fname = xstrdup (text);
}
/* Swallow all other tags. Either they are extraneous
 * or they reflect future extensions that we can
 * safely ignore. */
else if (strncmp (tag, "noline") == 0)
    print ("\n");
else if (text != NULL)
    print ("%s", text);
}
#endif /* CLIENT_SUPPORT */
#endif /* CLIENT_SUPPORT */
#endif /* SERVER_SUPPORT */

/* This table must be writable if the server code is included. */

#define RSP_LINE(n, f, t, s) (n, f, t, s)
#define RSP_LINEn (n, f, t, s, n, f, t, s, n, f, t, s)

/* CLIENT_SUPPORT */
#define RSP_LINE(ok, handle, response_type, ok, re, essential)
#define RSP_LINE(error, handle, response_type, error, re, essential)
#define RSP_LINE(valid_requests, handle, valid_requests, response_type, normal, re, essential)
#define RSP_LINE(checked-in, handle, checked_in, response_type, normal, re, essential)
#define RSP_LINE(new-entry, handle, new_entry, response_type, normal, re, optional)
#define RSP_LINE(checksum, handle, checksum, response_type, normal, re, optional)
#define RSP_LINE(copy-file, handle, copy_file, response_type, normal, re, optional)
#define RSP_LINE(updated, handle, updated, response_type, normal, re, essential)
#define RSP_LINE(created, handle, created, response_type, normal, re, optional)
#define RSP_LINE(update-existing, handle, update_existing, response_type, normal, re, optional)
#define RSP_LINE(merged, handle, merged, response_type, normal, re, essential)
#define RSP_LINE(patched, handle, patched, response_type, normal, re, optional)
#define RSP_LINE(diff, handle, rs, diff, response_type, normal, re, optional)
#define RSP_LINE(mode, handle, mode, response_type, normal, re, optional)
#define RSP_LINE(mod-time, handle, mod_time, response_type, normal, re, optional)
#define RSP_LINE(removed, handle, removed, response_type, normal, re, essential)
#define RSP_LINE(remove-entry, handle, remove_entry, response_type, normal, re, optional)
#define RSP_LINE(set-static-directory, handle, set_static_directory, response_type, normal, re, essential)
#define RSP_LINE(clear-static-directory, handle, clear_static_directory, response_type, normal, re, optional)
#define RSP_LINE(set-sticky, handle, set_sticky, response_type, normal, re, optional)
#define RSP_LINE(clear-sticky, handle, clear_sticky, response_type, normal, re, optional)
#define RSP_LINE(template, handle, template, response_type, normal, re, optional)
#define RSP_LINE(set-checkin-prog, handle, set_checkin_prog, response_type, normal, re, optional)
#define RSP_LINE(set-update-prog, handle, set_update_prog, response_type, normal, re, optional)
#define RSP_LINE(notified, handle, notified, response_type, normal, re, optional)
#define RSP_LINE(module-expansion, handle, module_expansion, response_type, normal, re, optional)
#define RSP_LINE(wrapper-rsc, handle, wrapper_rsc, response_type, normal, re, optional)
#define RSP_LINE(not-carried, handle, not_carried, response_type, normal, re, essential)
#define RSP_LINE(create-remote-branch, handle, create_remote_branch, response_type, normal, re, essential)
#define RSP_LINE(mnemonic, handle, mnemonic, response_type, normal, re, essential)
#define RSP_LINE(ts, handle, ts, response_type, normal, re, essential)
#define RSP_LINE(mt, handle, mt, response_type, normal, re, optional)
#define RSP_LINE(mt, handle, mt, response_type, normal, re, optional)

#if 0 /* Possibility should be response_type, error. */
#define RSP_LINE(NULL, NULL, response_type, normal, re, essential)
## Source code

```c
#ifndef RSP_LINE

#include <client.h>

#ifdef CLIENT_SUPPORT or SERVER_SUPPORT */
#endif

CLIENT_SUPPORT */
*/

void send_to_server(char *str, size_t len)
{
    size_t nbytes;
    if (len == 0)
        len = strlen(str);

    buf_output(to_server, str, len);
    /* There is no reason not to send data to the server, so do it whenever we've accumulated enough information in the buffer to make it worth sending. */
    nbytes += len;
    if (nbytes >= 2 * BUFFER_DATA_SIZE)
        {
        int status;
        status = buf_send_output(to_server);
        if (status != 0)
            error(1, status, "error writing to server");
        nbytes = 0;
        }
    
    /* Read up to LEN bytes from the server. Returns actual number of bytes read, which will always be at least one; blocks if there is no data available at all. Gives a fatal error on EOF or error. */
    static size_t
    try_read_from_server(char *buf, size_t len)
    {
        size_t red = 0;
        while (red < len)
            {
            red += try_read_from_server(buf + red, len - red);
            if (red == len)
                break;
            }
    
    /* Get some server responses and process them. Returns nonzero for error, 0 for success. */
    int get_server_responses()
    {
        
```
struct response *rs;
    do
        
        char *cmd;
        int len;
    
        len = readline(&cmd);
        for (rs = responses; rs->name != NULL; ++rs)
            if (strcmp(rs->name, strlcn(rs->name)) == 0)
                
                int cmdlen = strlcn(rs->name);
                
                if (cmd[cmdlen] == '\0')
                    else if (cmd[cmdlen] == ' ')
                        ++cmdlen;
                
                /* The first len characters match, but it's a different
                 * response. e.g. the response is "oklahoma" but we
                 * matched "ok".
                 */
                continue;
                
                (*rs->func)(cmd + cmdlen, len - cmdlen);
                break;
    
            if (rs->name == NULL)
                /* It's OK to print just to the first '\0'. */
                /* We might want to handle control characters and the like
                 * in some other way other than just sending them to stdout.
                 * One common reason for this error is if people use \ext:
                 * with a version of rsh which is doing CRLF translation or
                 * something, and so the client gets "ok:3M" instead of "ok".
                 * Right now that will tend to print part of this error
                 * message over the other part of it. It seems like we could
                 * do better (either in general, by quoting or omitting all
                 * control characters, and/or specifically, by detecting the CRLF
                 * case and printing a specific error message). */
                
                error(0, 0, "warning: unrecognized response '\t' from cvs server",
                        cmd);
                
                free(cmd);
    
    while (rs->type == response_type_normal);
    
        if (updated_fname != NULL)
            
            /* Output the previous message now. This can happen
             * if there was us Update-existing or other such
             * response, due to the -m global option. */
            
            cvs_output("%s
", 0);
            
            cvs_output(updated_fname, 0);
            
            cvs_output("\n\n", 1);
                
                free(updated_fname);
                
                updated_fname = NULL;
    
            if (rs->type == response_type_error)
                return 1;
            if (failure || exit)
                return 1;
            return 0;
    
    /* Get the responses and then close the connection. */
    
    int server_fd = -1;
    
    /* Flag var; we'll set it in start_server() and not one of its
     * callees, such as start_sh()server(). This means that there might
     * be a small window between the starting of the server and the
     * setting of this var, but all the code in that window shouldn't care
     * because it's busy checking return values to see if the server got
     * started successfully anyway.
     */
    
    int server_started = 0;
    
    int get_responses_and_close()
        
        int err = get_server_responses();
        
        int status;
    
    if (last_entries != NULL)
        
        EntriesClose(last_entries);
        
        last_entries = NULL;
        
        defer_close();

    if (client_errno)
process();

/* The calls to bufshutdown are currently only meaningful when
we are using compression. First we shut down TO_SERVER. That
tells the server that its input is finished. It then shuts
down the buffer it is sending to us, at which point our shut
down of FROM_SERVER will complete. */

status = bufshutdown(to_server);
if (status != 0)
    error (0, status, "shutting down buffer to server");

status = bufshutdown(from_server);
if (status != 0)
    error (0, status, "shutting down buffer from server");

ifdef NO_SOCKET_TO_FD
    if (use_socket != 0)
        if (shutdown (server_sock, 2) < 0)
            error (1, 0, "shutting down server socket: %s",
                   SOCK_STRERROR (SOCK_ERRNO));
    endif
else
    endif /* NO_SOCKET_TO_FD */

ifdef (HAVE_KERBEROS) || defined(AUTH_CLIENT_SUPPORT)
    if (server_fd != -1)
        if (shutdown (server_fd, 1) < 0)
            error (1, 0, "shutting down connection to %s: %s",
                   CVSroot_hostname, SOCK_STRERROR (SOCK_ERRNO));
        endif
        /* This text will always be true because we dup the descriptor */
        if (fileno (from_server_fp) != fileno (to_server_fp))
            if (fclose (to_server_fp) != 0)
                error (1, errno, "closing down connection to %s",
                       CVSroot_hostname);
            endif
        endif
    endif /* HAVE_KERBEROS */
else
    endif /* AUTH_CLIENT_SUPPORT */

ifdef SHUTDOWN_SERVER
    SHUTDOWN_SERVER (fileno (to_server_fp));
else /* SHUTDOWN_SERVER */
    endif

ifdef START_RSH_WITH_POPEN_RW
    if (pclose (to_server_fp) == EOF)
        endif /* START_RSH_WITH_POPEN_RW */
else
    endif /* START_RSH_WITH_POPEN_RW */

ifdef / * START_RSH_WITH_POPEN_RW */
    if (fclose (to_server_fp) == EOF)
        error (1, errno, "closing connection to %s",
               CVSroot_hostname);
    endif
endif

if (!bufempty (from_server))
    if (getc (from_server_fp) != EOF)
        error (0, 0, "dying gasps from %s unexpected",
               CVSroot_hostname);
    else if (error (from_server_fp))
        error (0, errno, "reading from %s", CVSroot_hostname);
    endif
endif

fclose (from_server_fp);
/* SHUTDOWN_SERVER */

if (rsh_pid != -1)
    &waitpid (rsh_pid, (int *) 0, 0) = -1)
    error (1, errno, "waiting for process %d", rsh_pid);

server_started = 0;
/* see if we need to sleep before returning */
if (last_register_time)
    time (now);

(void) time (&now);
if (now == last_register_time)
    sleep (1);
/* to avoid time-stamp races */
return errors;

endif NO_EXT_METHOD
static void start_client PROTO((int *, int *));

if defined (AUTH_CLIENT_SUPPORT) || defined (HAVE_KERBEROS)
static struct hostent *init_sockaddr PROTO ((struct sockaddr_in *, char *, unsigned int));
static struct hostent *
init_sockaddr (name, hostname, port)
struct sockaddr_in *name;
char *hostname;
unsigned int port;
{
  struct hostent *hostinfo;
  unsigned short shortport = port;
  memset (name, 0, sizeof (**name));
  name->sin_family = AF_INET;
  name->sin_port = htons (shortport);
  hostinfo = gethostbyname (hostname);
  if (hostinfo == NULL)
    {
      fprintf (stderr, "Unknown host %s.%s", hostname);
      exit (1);
    }
  name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
  return hostinfo;
}
endif /* defined (AUTH_CLIENT_SUPPORT) || defined (HAVE_KERBEROS) */

ifdef AUTH_CLIENT_SUPPORT
static int auth_server_port_number PROTO ((void));
static int
auth_server_port_number ()
{
  struct servent *s = getservbyname ("cvsserver", "tcp");
  if (s)
    return ntohs (s->version);
  else
    return CVS_AUTH_PORT;
}

/* Read a line from socket SOCK. Result does not include the
terminating linefeed. This is only used by the authentication
protocol, which we call before we set up all the buffering stuff.
It is possible it should use the buffers too, which would be faster
(like the server, there isn't really a security issue in terms of
separating authentication from the rest of the code).

Space for the result is malloc'd and should be freed by the caller.

Returns number of bytes read. */
static int
recv_line (sock, resultp)
int sock;
char **resultp;
{
  int c;
  char *result;
  size_t input_index = 0;
  size_t result_size = 80;
  result = (char *) xmalloc (result_size);
  while (1)
char ch;
if (recv (sock, &ch, 1, 0) < 0)
  error (1, 0, "recv() from server %s: %s", CVSroot_hostname, 
  SOCKSTRERROR (SOCK_ERRNO));
  c = ch;
if (c == EOF)
  {
    free (result);
    /* It’s end of file. */
    error (1, 0, "end of file from server");
  }
if (c == '\012')
  break;
result[input_index+1] = c;
while (input_index + 1 >= result_size)
  {  
    result_size = 2;
    result = (char *) xrealloc (result, result_size);
  }
if (resultp)
  *resultp = result;
/* Terminate it just for kicks, but we can deal with embedded NULLs. */
result[input_index] = '\0';

3610  if (resultp == NULL)
  free (result);
  return input_index;
/* Connect to the authenticating server.
If VERIFYONLY is non-zero, then just verify that the password is
correct and then shutdown the connection.
3620  if (VERIFYONLY is 0, then really connect to the server.
If DOREQSAPI is non-zero, then we use GSSAPI authentication rather
than the server password authentication.
If we fail to connect or if access is denied, then die with fatal
error. */
void
connect_to_server (tofdp, fromfdp, verify_only, method)
  int *tofdp, *fromfdp;
  int verify_only;
  int method;
  {
    int sock;
    define SOCK_TO_FD
    int tofd, fromfd;
    sendif
    int port_number;
    struct sockaddr_in client_sai;
    struct hostent *hostinfo;
    char* auth_list_buf;
    char* method_name;
    char* request;
    char start [64];
    char end [64];
    sock = socket (AF_INET, SOCK_STREAM, 0);
    if (sock == -1)
      {
        error (1, 0, "cannot create socket: %s", SOCKSTRERROR (SOCK_ERRNO));
      }
    port_number = auth_server_port_number ();
    hostinfo = init_sockaddr (&client_sai, CVSroot_hostname, port_number);
    if (connect (sock, (struct sockaddr *) &client_sai, sizeof (client_sai)) < 0)
      {
        error (1, 0, "connect to %s: %s", CVSroot_hostname, 
          port_number, SOCKSTRERROR (SOCK_ERRNO));
      }
    recv_line (sock, &auth_list_buf);
    free (auth_list_buf);

3680  if (method == AUTH_GSSAPI) {
    method_name = "GSSAPI";
  } else if (method == AUTH_KERBEROS_V4) {
    method_name = "KERBEROS_V4";
  } else {
    method_name = "PASSWORD";
  }
if (verify_only) {
    request = "VERIFICATION";
} else {
    request = "AUTHENTICATION";
}

sprintf ((start, "BEGIN %s %s REQUEST\012", method, request);

if (send (sock, start, strlen (start), 0) < 0) {
    error (1, 0, "Cannot send: %s", SOCKERRERROR (SOCKERRNO));
}

#else if (method == AUTH_GSSAPI) {
    if (connect_to_gss_server (verify_only, sock, hostinfo))
        goto rejected;

#else if (method == AUTH_KERBEROS_V4) {
    if (connect_to_gss_server (verify_only, sock, hostinfo))
        goto rejected;

#endif

if (send (sock, end, strlen (end), 0) < 0) {
    error (1, 0, "Cannot send: %s", SOCKERRERROR (SOCKERRNO));
}

char *readbuf;

/* Loop, getting responses from the server. */
while (1) {

    readline (sock, &readbuf);

    if (strcmp (readbuf, "I HATE YOU") == 0)
        /* Authorization not granted. */
        goto rejected;

    else if (strcmp (readbuf, "E") == 0) {
        printf (stderr, "AUTHENTICATION: %s\n", readbuf);
        /* Continue with the authentication protocol. */
    }
    else if (strcmp (readbuf, "error ", 6) == 0)
        {
            char *p;
            /* First skip the code. */
            p = readbuf + 6;
            while (*p != NULL)
                ++p;
            /* Skip the space that follows the code. */
            if (*p == '\n')
                ++p;
            /* Now output the text. */
            printf (stderr, "%s\n", p);
            goto rejected;
        }
    else if (strcmp (readbuf, "I LOVE YOU") == 0) {
        free (readbuf);
        break;
    }
    else {
        /* Unrecognized response from server. */
        if (shutdown (sock, 2) < 0)
            error (0, 0, "unrecognized auth response from %s: %s",
                   CVSroot, hostname, readbuf);
    }
}
error (1, 0, "unrecognized auth response from \%s: \%s", 
CVSroot_hostname, read_buf);
    }
    free (read_buf);
    }
}

if (verifyonly)
    if (shutdown (sock, 2) < 0)
        error (0, 0, "shutdown() failed, server \%s: \%s", 
    CVSroot_hostname, 
    SOCK_STRERROR (SOCK_ERRNO));
    return;
else
    if (NO_SOCKET_TO_FD)
    {
        use_socket_style = 1;
        server_sock = sock;
        /* Try to break mistaken callers: */
        fromfp = 0;
        tofp = 0;
        server_fd = sock;
        close_on_exec (server_fd);
        /* Hand them back to the caller. */
        tofd = tofp = fromfp = fromfdp = 0;
    }
    else
    {
    socket_tofdp = 0;
    fromfdp = 0;
    tofd = tofp = fromfp = fromfdp = 0;
    server_fd = sock;
    close_on_exec (server_fd);
    /* Hand them back to the caller. */
    tofd = tofp = fromfp = fromfdp = 0;
    return;
    }

    rejected:
    if (shutdown (sock, 2) < 0)
    {
        error (0, 0, "authorization failed: server \%s rejected access", 
    CVSroot_hostname);
        error (1, 0, "shutdown() failed (server \%s): \%s", 
    CVSroot_hostname, 
    SOCK_STRERROR (SOCK_ERRNO));
    }
    error (1, 0, "authorization failed: server \%s rejected access", 
    CVSroot_hostname);
}

#elif HAVE_KERBEROS
    /* This function has not been changed to deal with NO_SOCKET_TO_FD
     * (i.e., systems on which sockets cannot be converted to file
     * descriptors). The first person to try building a kerberos client
     * on such a system (OS/2, Windows 95, and maybe others) will have to
     * make take care of this. */
    
    void
    start_tcppserver (tofdp, fromfdp)
    
    {
        int *tofdp, *fromfdp;
        
        int s;
        const char *portenv;
        int port;
        struct hostent *hp;
        struct sockaddr_in sin;
        char *hname;

        s = socket (AF_INET, SOCK_STREAM, 0);
        if (s < 0)
            error (1, 0, "cannot create socket: \%s", 
        SOCK_STRERROR (SOCK_ERRNO));
        
        /* Get CVS_CLIENT_PORT or look up cvs/tcp with CVS_PORT as default */
        portenv = getenv ("CVS_CLIENT_PORT");
        if (portenv != NULL)
        {
            port = atoi (portenv);
            if (port <= 0)
            {
                error (0, 0, "CVS_CLIENT_PORT must be a positive number! If you");
                error (0, 0, "are trying to force a connection via rsh, please");
                error (0, 0, "put \"server:\" at the beginning of your CVSROOT");
                error (1, 0, "variable.");
            }
        }
        if (trace)
            fprintf(stderr, "Using TCP port \%d to contact server.\%s", port);
else
{
    struct servent *sp;
    sp = getservbyname ("cvs", "top");
    if (sp == NULL)
        port = CVS_PORT;
    else
        port = ntohs ((sp->sport));
}

hp = init_sockaddr (&sin, CVSroot.hostname, port);
hp->hname = xmalloc (strlen (hp->hname) + 1);
strcpy (hp->hname, hp->hname);

if (connect (a, (struct sockaddr *) &sin, sizeof (sin)) < 0)
    error (1, 0, "connect to %s:%d failed: %s", CVSroot.hostname, port, SOCKSTRERROR (SOCKERRNO));

#define HAVE_KERBEROS
{
    const char *realm;
    struct sockaddr_in ladder;
    int laddrlen;
    KTEXT ticket;
    MSGDAT msgdata;
    CREDENTIALS cred;
    int status;

    realm = krb.realmofhost (hname);
    laddrlen = sizeof (ladder);
    if (getsockname (a, (struct sockaddr *) &ladder, &laddrlen) < 0)
        error (1, 0, "getsockname failed: %s", SOCKSTRERROR (SOCKERRNO));

    /* We don't care about the checksum, and pass it as zero. */
    status = krb.gendauth (KOPT_DO_MUTUAL, &ticket, "rcmd", hname, realm, (unsigned long) 0, &msgdata, &cred, sched, &ladder, &sin, "KEXTL.0");

    if (status != KSUCCESS)
        error (1, 0, "kerberos authentication failed: %s", krb.geterrtext (status));
    memcpy (kblock, cred.session, sizeof (C_Binding));
}
#undef HAVE_KERBEROS

#define HAVE_GSSAPI
3930 /* Receive a given number of bytes. */
static void
recv (void *sock, buf, need)
{
    int sock;
    char *buf;
    int need;

    while (need > 0)
    {
        int got;

        got = recv (sock, buf, need, 0);
        if (got < 0)
            error (1, 0, "recv() from server %s: %s", CVSroot.hostname, SOCKSTRERROR (SOCKERRNO));
        buf += got;
        need -= got;
    }

    /* Connect to the server using GSSAPI authentication. */

static int
connect (server (only)verify, sock, hostinfo)
{
```c
{ char *str;
  char buf[1024];
  gss_buffer_in *tokin, tokout;
  OM_uint32 statmin, statmaj;
  gss_name_in server_name;

  str = "BEGIN GSSAPI AUTHENTICATION REQUEST\012";

  if (send (sock, str, strlen (str), 0) < 0)
    error (1, 0, "cannot send: \%s", GSS_STRERROR (SOCKETERRNO));

  sprintf (buf, "cvs@%s", hostname);
  tokin.length = strlen (buf);
  tokin.value = buf;
  gss_import_name (&statmin, &tokin, GSS Unsure\ HOSTBASED\ SERVICE, &server_name);

  tokin ptr = GSS Unsure\ BUFFER;
  gcontext = GSS Unsure\ CONTEXT;

  do {
    statmaj = gss_init_sec_context (&statmin, GSS C\ NO\ CREDENTIAL, &gcontext, server_name, GSS C NULL\ OID, (GSS C MUTUAL\ FLAG | GSS C\ REPLAY\ FLAG), 0, NULL, tokin, str, NULL, &tokout, NULL, NULL);

    if (statmaj != GSS C\ COMPLETE \&\& statmaj != GSS C\ CONTINUE\ NEEDED)
      { OM_uint32 message_context;
        message_context = 5;
        gss_display_status (&statmin, statmaj, GSS C\ C\ CODE, GSS C NULL\ OID, &message_context, &tokout);
        error (1, 0, "GSSAPI authentication failed: \%s", (char *) tokout.value);
      }

    if (tokout.length == 0)
      { tokin.length = 0;
        }
    else {
      char cbuf[2];
      int need;

      cbuf[0] = (tokout.length >> 8) & 0xff;
      cbuf[1] = tokout.length & 0xff;

      if (send (sock, cbuf, 2, 0) < 0)
        error (1, 0, "cannot send: \%s", GSS_STRERROR (SOCKETERRNO));

      if (send (sock, tokout.value, tokout.length, 0) < 0)
        error (1, 0, "cannot send: \%s", GSS_STRERROR (SOCKETERRNO));

      recvbytes (sock, cbuf, 2);
      need = ((cbuf[0] & 0xff) << 8) | (cbuf[1] & 0xff);
      assert (need <= sizeof buf);
      recvbytes (sock, buf, need);

      tokin.length = need;
    }

    tokin.value = buf;
    tokin.ptr = &tokin;
  } while (statmaj == GSS C\ CONTINUE\ NEEDED);

  return 1; /* success */
}

sendf /* HAVE GSSAPI */

int connect_server (int verify_only, int sock, struct hostent *hostname) {
  char repository = CVSServer\ directory;
  char username = CVSServer\ username;
  char password = NULL;

  /* Get the password, probably from \/.cvs\_password */
  password = get_cvs\_password ();

  /* Send the data the server needs. */
  if (send (sock, repository, strlen (repository), 0) < 0)
    error (1, 0, "cannot send: \%s", GSS_STRERROR (SOCKETERRNO));

  if (send (sock, "\012", 1, 0) < 0)
    error (1, 0, "cannot send: \%s", GSS_STRERROR (SOCKETERRNO));

  if (send (sock, username, strlen (username), 0) < 0)
    error (1, 0, "cannot send: \%s", GSS_STRERROR (SOCKETERRNO));
```
if (send (sock, \"GET\", 1, 0) < 0)  
        error (1, 0, \"Cannot send: \%s\", SOCK_strerror (SOCK_ERRNO));  
if (send (sock, password, strlen (password), 0) < 0)  
        error (1, 0, \"Cannot send: \%s\", SOCK_strerror (SOCK_ERRNO));  
if (send (sock, \"\012\", 1, 0) < 0)  
        error (1, 0, \"Cannot send: \%s\", SOCK_strerror (SOCK_ERRNO));  

/* Paranoia. */  
memset (password, 0, strlen (password));  
return 1;  
/* success */

static int  
semi_variable proc (node, closure)  
    Node =node;  
    void closure;  
    {  
semi_variable server (*atl, 0);  
semi_variable server (node->key, 0);  
semi_variable server (*atl, 1);  
semi_variable server (node->data, 0);  
semi_variable server (*\012", 1);  
return 0;  
    }  
/* Contact the server. */  
void  
start_server ()  
    {  
int tofd, fromfd;  
char clog = getenv (\"CVS_CLIENT_LOG\");  

/* Note that generally speaking we do *not* fall back to a different  
way of connecting if the first one does not work. This is slow  
(*really* slow on a 14.4kbps link); the clean way to have a CVS  
which supports several ways of connecting is with access methods. */
 switch (CVSrootmethod)  
    {  
ifdef AUTHCLIENT_SUPPORT  
    case server method:  
        /* */  
        /* Toss the return value. It will die with error if anything  
goes wrong anyway. */  
    }
connect_to_server (&tofd, &fromfd, 0, AUTH_PASSWORD);
break;
#endif

if HAVE_KERBEROS
  case kerberos_method:
    connect_to_server (&tofd, &fromfd, 0, AUTH_KERBEROS_V4);
  /* This causes the server to send the greeting twice */
  /* start_to_server (&tofd, &fromfd); */
  break;
#endif

if HAVE_GSSAPI
  connect_to_server (&tofd, &fromfd, 0, AUTH_GSSAPI);
#endif
  case ext_method:
  if defined(NO_EXT_METHOD)
    error (0, 0, "extension method not supported by this port of CVS");
    error (1, 0, "try :server: instead");
    break;
  else
    start_rsh_server (&tofd, &fromfd);
#endif
  break;
#endif
  case server_method:
  if defined(START_SERVER) & defined(NO_SOCKET_TO_FD)
    /* This is a system on which we can only write to a socket using send/rece. Therefore its START_SERVER needs to return a socket. */
    use_socket_style = 1;
    server_sock = tofd;
  #endif
  false
  /* FIXME: It should be possible to implement this portably,
   * like pserver, which would get rid of the duplicated code
   * in {unix, windows-NT...}/startserver.c. */
  error (1, 0, ": server: access method is not supported by this port of CVS");
#endif
  break;
#endif
  default:
    error (1, 0, ":server: internal error: unknown access method");
#endif
  break;

  /* Hi, I'm Darlene and I'll be your server tonight... */
  server_started = 1;
#endif
  NO_SOCKET_TO_FD
  if (use_socket_style)
    {
      to_server = socket_bufferinitialize (server_sock, 0,
          (BUFSIZEMEMPROC) NULL);
      from_server = socket_bufferinitialize (server_sock, 1,
          (BUFSIZEMEMPROC) NULL);
    }
else
  endif
  /* NO_SOCKET_TO_FD */
  {
    /* todo: some OS's don't need these calls... */
    close_to_execute (tofd);
    close_to_execute (fromfd);
  }
#endif
  /* SCO 3 and AIX have a nasty bug in the I/O libraries which precludes
duping the same file descriptor twice, so dup it if it is the
same. */
  if (tofd == fromfd)
    {
      fromfd = dup (tofd);
      if (fromfd < 0)
        error (1, errno, "cannot dup net connection");
    }
#endif
  /* These will use binary mode on systems which have it. */
  to_server_fp = fopen (tofd, FOPEN_BINARY_WRITE);
  if (to_server_fp == NULL)
    error (1, errno, "cannot fopen %d for write", tofd);
  to_server = stdin_bufferinitialize (to_server_fp, 0,
      (BUFSIZEMEMPROC) NULL);
  from_server_fp = fopen (fromfd, FOPEN_BINARY_READ);
  /* The shell use binary mode on systems which have it. */
if (from_server != NULL) {
    error (l, er, "cannot free \%d for read", fromfd);
    from_server = stdio_buffer_initialize (from_server, l, (BUFMEMERRPROC) NULL);
}

int len = strlen (log);
char *buf = xmalloc (len + 5);
char *fp;
FILE *fp;
strcpy (buf, log);
p = buf + len;

/* Open logfiles in binary mode so that they reflect exactly what was transmitted and received (that is more important than that they be maximally convenient to view). */

/* Note that if we create several connections in a single CVS client (currently used by update.c), then the last set of logfiles will overwrite the others. There is currently no way around this. */
/* strcopy (p, "in"); */
fp = open_file (buf, "wb");
if (fp == NULL) {
    error (0, errno, "opening to-server logfile %s", buf);
    exit (0);
} else {
    from_server = log_buffer_initialize (from_server, fp, 0, (BUFMEMERRPROC) NULL);
}

free (buf);

/* Clear static variables. */
if (toplevel_repos != NULL) {
    free (toplevel_repos);
toplevel_repos = NULL;
}
if (last_update_dir != NULL) {
    free (last_update_dir);
    last_update_dir = NULL;
    stored_checksum_valid = 0;
    stored_mode_valid = 0;
}

if (strcmp (command_name, "init") != 0) {
    send_to_server ("Root", 0);
    send_to_server (CVSroot_directory, 0);
    send_to_server ("init", 1);
}

struct response *rs;
send_to_server ("Valid-responses", 0);

for (rs = responses; rs->name != NULL; ++rs) {
    send_to_server (rs->name, 0);
    send_to_server ("Valid-responses", 0);
}

send_to_server ("Valid-requests", 0);

if (get_server_responses ()
    error_exit ();

    /* Now handle global options. */
    /* -H, -f, -d, -e should be handled OK locally. */
    /* -h we ignore (treating it as a server installation issue). */
    /* PIXME: should be an error message. */
-v we print local version info; FIXME: Add a protocol request to get
the version from the server so we can print that too.
*/

int have_global = supported_request("Global_option");

if (noexec)
    if (have_global)
        send_to_server("Global_option -n\012", 0);
    else
        error(1, 0,
        "This server does not support the global -n option.");

if (quiet)
    if (have_global)
        send_to_server("Global_option -q\012", 0);
    else
        error(1, 0,
        "This server does not support the global -q option.");

if (really_quiet)
    if (have_global)
        send_to_server("Global_option -Q\012", 0);
    else
        error(1, 0,
        "This server does not support the global -Q option.");

if (cvswrite)
    if (have_global)
        send_to_server("Global_option -r\012", 0);
    else
        error(1, 0,
        "This server does not support the global -r option.");

if (trace)
    if (have_global)
        send_to_server("Global_option -t\012", 0);
    else
        error(1, 0,
        "This server does not support the global -t option.");

if (logoff)
    if (have_global)
        send_to_server("Global_option -l\012", 0);
    else
        error(1, 0,
        "This server does not support the global -l option.");

/* Find out about server-side cvswrappers. An extra network
turnaround for cvs import seems to be unavoidable, unless we
want to add some kind of client-side place to configure which
filenames imply binary. For cvs add, we could avoid the
problem by keeping a copy of the wrappers in CVSAADM (the main
reason to bother would be so we could make add work without
contacting the server, I suspect). */

if ((strcmp(command_name, "import") == 0)
    || (strcmp(command_name, "add") == 0))
    if (supported_request("wrapper-sendme-rcsOptions"))
        int err;
        send_to_server("wrapper-sendme-rcsOptions\012", 0);
        err = get_server_responses();
        if (err != 0)
            error(err, 0, "error reading from server");
if (cvsencrypt) {
    #ifdef ENCRYPTION
    /* Turn on encryption before turning on compression. We do not want to try to compress the encrypted stream. Instead, we want to encrypt the compressed stream. If we can't turn on encryption, bomb out; don't let the user think the data is being encrypted when it is not. */
    #ifdef HAVE_KERBEROS
    if (CVSRoot_method == kserver_method)
        { if (!supported_request("Kerberos-encrypt"))
                error (1, 0, "This server does not support encryption");
            if (to_server)
                from_server = krb_encrypt_buffer_initialize (from_server, 1,
                                                        sched, kbloc,
                                                        (BUFMEMERRPROC) NULL);
            to_server = krb_encrypt_buffer_initialize (to_server, 0, sched,
                                                      kbloc,
                                                      (BUFMEMERRPROC) NULL);
        }
    else
    #endif /* HAVE_KERBEROS */
    #ifdef HAVE_GSSAPI
    if (CVSRoot_method == gserver_method)
        { if (!supported_request("Gssapi-encrypt"))
                error (1, 0, "This server does not support encryption");
            if (to_server)
                from_server = cvs_gssapi_wrap_buffer_initialize (from_server, 1,
                                                        (BUFMEMERRPROC) NULL);
            to_server = cvs_gssapi_wrap_buffer_initialize (to_server, 0, gcontext,
                                                        (BUFMEMERRPROC) NULL);
            cvs_gssapi_encrypt = 1;
        }
    else
    #endif /* HAVE_GSSAPI */
    #else /* ! ENCRYPTION */
    #endif /* ! ENCRYPTION */
}

if (gzip_level)
    { if (supported_request("Gzip-stream"))
        { char gzip_level[buf][5];
          send_to_server (gzip-stream *, 0);
          sprintf (gzip_level[buf], "#", gzip_level);
          send_to_server (gzip_level[buf], 0);
          send_to_server ("012", 1);
          /* All further communication with the server will be compressed. */
          to_server = compress_buffer_initialize (to_server, 0, gzip_level,
                                                      (BUFMEMERRPROC) NULL);
          from_server = compress_buffer_initialize (from_server, 1,
                                                     gzip_level,
                                                     (BUFMEMERRPROC) NULL);
        }
    #endif /* ! ENCRYPTION */
    #else /* ! ENCRYPTION */
    #endif /* ! ENCRYPTION */
    #endif /* ! CLIENT_GZIP_PROCESS */
else if (supported_request("gzip-file-contents"))
    { char gzip_level[buf][5];
      send_to_server (gzip-file-contents *, 0);
      sprintf (gzip_level[buf], "#", gzip_level);
      send_to_server (gzip_level[buf], 0);
      send_to_server ("012", 1);
      gzip_level = gzip_level;
    }
else
    { fprintf (stderr, "This client does not support gzip-file-contents\n"); /* Setting gzip_level to 0 prevents us from giving the error twice if update has to contact the server again to fetch unpatchable files. */
      gzip_level = 0;
    }
Right now, we have two different definitions for this function, depending on whether we start the rsh server using popenRW or not. This isn’t ideal, and the best thing would probably be to change the OS/port to be more like the regular Unix client (i.e., by implementing piped child). ... but I’m doing something else at the moment, and wish to make only one change at a time. -Karl

This is actually a crock – it’s OS/port-specific, for no one else uses it. If I get time, I want to make piped child and all the other stuff in os2/run.c work right. In the meantime, this gets us up and running, and that’s most important. */

static void
start_rsh_server(tofdp, fromfdp)
{ int *tofdp, *fromfdp;

/* If you’re working through firewalls, you can set the */
CVS_RSH environment variable to a script which uses rsh to
invoke another rsh on a proxy machine. */
char *cvv_rsh = getenvar (“CVS_RSH”);
char *cvv_server = getenvar (“CVS_SERVER”);
int i = 0;
/* This needs to fit “rsh”, “-h”, “-l”, “USER”, “host”,
“cmd (w/ args)”, and NULL. We leave some room to grow. */
char rsh_argv[10];

if (tvv_rsh)
/* People sometimes suggest or assume that this should default
to “remsh” on systems like HPUX in which that is the
system-supplied name for the rsh program. However, that
causes various problems (keep in mind that systems such as
HPUX might have non-system-supplied versions of “rsh”, like
a Kerberized one, which one might want to use). If we
based the name on what is found in the PATH of the person
who runs configure, that would make it harder to
consistently produce the same result in the face of
different people producing binary distributions. If we
based it on "remsh" always being the default for HPUX
(e.g. based on uname), that might be slightly better but
would require us to keep track of what the defaults are for
each system type, and probably would cope poorly if the
existence of remsh or rsh varies from OS version to OS
version. Therefore, it seems best to have the default
remain "rsh", and tell HPUX users to specify remsh, for
example in CVS_RSH or other such mechanisms to be devised,
if that is what they want (the manual already tells them
that). */

4590
cvs_rsh = "rsh";
if (lcvw_server)
cvs_server = "cvs";

4600
/* The command line starts out with rsh. */
rsh_argv[i++] = cvs_rsh;

4610
/* The command line starts out with rsh. */
rsh_argv[i++] = cvs_rsh;

/* The command line starts out with rsh. */
rsh_argv[i++] = cvs_server;

/* Mark the end of the arg list. */
rsh_argv[i++] = (char *) NULL;

4620

/* Do the deed. */
rsh_pid = popenRW (rsh_argv, pipes);
if (rsh_pid < 0)
    error (1, errstr, "cannot start server via rsh");

/* Give caller the file descriptors. */
tofdp = pipes[0];
fromfdp = pipes[1];

4630

else /* START_RSH_WITH_POPEN_RW */

4640

static void
start_rsh_server (tofdp, fromfdp)
int *tofdp;
int *fromfdp;
{
/* If you're working through firewalls, you can set the
CVS_RSH environment variable to a script which uses rsh to
invoke another rsh on a proxy machine. */
char *cvs_rsh = getenv ("CVS_RSH");
char *cvs_server = getenv ("CVS_SERVER");
char *command;

if (lcvw_rsh)
cvs_rsh = "rsh";
if (lcvw_server)
cvs_server = "cvs";

4650
/* Pass the command to rsh as a single string. This shouldn't
affect most rsh servers at all, and will pacify some buggy
versions of rsh that grab switches out of the middle of the
command (they're calling the GNU getopt routines incorrectly). */
command = xmalloc (strlen (cvs_server) + strlen (CVSroot) +
                 50);

/* If you are running a very old (Nov 3, 1994, before 1.5)
* version of the server, you need to make sure that your .bashrc
* on the server machine does not set CVSROOT to something
* containing a colon (or better yet, upgrade the server). */
sprintf (command, "%s server", cvs_server);

4660
char *argv[10];
char *p = argv;
/* If the login names differ between client and server * pass it on to rsh. */
if (CVSRoot_username != NULL) {
    *p++ = "-l";
    *p++ = CVSRoot_username;
}

/* The login names differ between client and server */
if (rsh_pid == NULL) {
    *p++ = command;
    *p++ = NULL;
    if (trace)
        { int i;
          fprintf (stderr, "Starting server: ");
          for (i = 0; argv[i]; i++)
            fprintf (stderr, " %s", argv[i]);
          fprintf (stderr, " -> Starting server: ");
          for (i = 0; argv[i]; i++)
            putc (argv[i][0], stderr);
        }
    rsh_pid = piped_child (argv, argv, fromfdp);
    if (rsh_pid < 0)
        error (1, errno, "cannot start server via rsh");
    free (command);
}

sendif /* START_RSH_WITH_OPEN_RW */
sendif /* NO_EXT_METHOD */

/ * Send an argument STRING. */
void
send_arg (string)
{ char *string;
  char *buf[];
  char *p = string;
  send_to_server (*string, 0);
  while (*p)
    { if (*p == '\n')
      { send_to_server (*\002 arguments, 0);
        } else
      { buf[0] = *p;
        send_to_server (buf, 1);
        ++p;
        } send_to_server (*\002, 1);
      }

static void send_modified PROTO ((char *, char *, Vers, TS *));

/* VERS>OPTIONS specifies whether the file is binary or not. NOTE: BEFORE * using any other fields of the struct vers, we would need to fix * vers->process->import->file to set them up. */
static void
send_modified (file, short pathname, vers)
{ char *file;
  char *short_pathname;
  Vers, TS *vers;
  /* File was modified, send st. */
  struct stat stb;
  int fd;
  char buf;
  char *mode, string;
  size_t bufsize;
  int bin;
  if (trace)
      (void) fprintf (stderr, "Sending file \"%s\" to server\n");
  / * Don't think we can assume fstat exists. */
  if (CVS_STAT (file, &stb) < 0)
      error (1, errno, "reading \"%s\", short_pathname);
mode_string = mode_to_string (sb.st_mode);

/* Beware: on systems using CRLF line termination conventions,
the read and write functions will convert CRLF to LF, so the
number of characters read is not the same as sb.st_size. Text
files should always be transmitted using the LF convention, so
we don't want to disable this conversion. */
bufsize = sb.st_size;
buf = xmalloc (bufsize);

/* Is the file marked as containing binary data by the "-kb" flag?
If so, make sure to open it in binary mode: */
if (vers & & vers->options)
    bin = ! (strcmp (vers->options, "-kb"));
else
    bin = 0;

ifdef BROKEN_READWRITE_CONVERSION
if (bin)
    {
        /* If only stdin, not open/write/etc., do text/binary
conversion, use convert_file which can compensate
(FIXME: we could just use stdio instead which would
avoid the whole problem). */
        char title[1024];
        steppy(title, file);
        strcpy(title, "CVSBFCTMP");
        convert_file (file, O_RDONLY, title, O_RDONLY | O_CREAT | O_TRUNC | OPEN_BINARY);
        fd = CVSS_OPEN (title, O_RDONLY | OPEN_BINARY);
        if (fd < 0)
            error (1, errno, "reading %s", short_pathname);
    }
else
    fd = CVSS_OPEN (file, O_RDONLY | OPEN_BINARY);
endif

else
    fd = CVSS_OPEN (file, O_RDONLY | OPEN_BINARY | (bin ? OPEN_BINARY : 0));
endif

if (fd < 0)
    error (1, errno, "reading %s", short_pathname);

if (file->strip_level & & sb.st_size > 100)
    {
        size_t newsize = 0;
        read_and_gzip (fd, short_pathname, (unsigned char **)buf,
                      &bufsize, &newsize,
                      file->strip_level);
        if (close (fd) < 0)
            error (0, errno, "warning: can't close %s", short_pathname);
            {
                char tmp[80];
                send_to_server ("Modified ", 0);
                send_to_server (file, 0);
                send_to_server ("\%012zU", 3);
                send_to_server (mode_string, 0);
                send_to_server ("\%012zU", 2);
                sprintf (tmp, "%lu", (unsigned long) newsize);
                send_to_server (tmp, 0);
                send_to_server (buf, newsize);
            }
        else
            {
                int newsize;

                char *bufp = buf;
                int len;
                /* FIXME: This is gross. It assumes that we might read
less than st_size bytes (true on NT), but not more.
Instead of this we should just be reading a block of
data (e.g. 8192 bytes), writing it to the network, and
so on until EOF. */
                while ((len = read (fd, bufp, (buf + sb.st_size) - bufp)) > 0)
                    bufp += len;
                if (len < 0)
                    error (1, errno, "reading %s", short_pathname);
                newsize = buf - buf;
            }
        if (close (fd) < 0)
            error (0, errno, "warning: can't close %s", short_pathname);
{  char tmp[80];
    send_to_server("Modified ", 0);
    send_to_server(file, 0);
    send_to_server("/", 1);
    send_to_server(modstring, 0);
    send_to_server("/", 1);
    sprintf(tmp, "%lu\n", newsize);
    send_to_server(tmp, 0);
}
#endif BROKEN_READWRITE__CONVERSION
if (bin) {
    char title[1024];
    strcpy(title, file);
    strcat(title, " .CVSBFCTMP");
    if (CVSUNLINK(title) < 0)
        error(0, errno, "warning: can't remove temp file %s", title);
} sendbuf

/* Note that this only ends with a newline if the file ended with *
 * one. */
/* The address of an instance of this structure is passed to *
 * send_fileproc, send_filesdoneproc, and send_fileentproc, as the *
 * callerdat parameter. */

struct send_data {
    /* Each of the following flags are zero for clear or nonzero for set. */
    int builddirs;
    int force;
    int noc contents;
};

static int send_fileproc PROTO ((void *callerdat, struct file_info *finfo));

/* Deal with one file. */
static int
send_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{  struct send_data *args = (struct send_data *) callerdat;
    Vers_TS *vers;
    struct file_info xfinfo;
    /* File name to actually use. Might differ in case from *
     * file_info->file. */
    char *filename;
    send_repository ("", finfo->repository, finfo->update_list);
    xfinfo = *finfo;
    xfinfo.repository = NULL;
    xfinfo.rcs = NULL;
    vers = Version_TS (&xfinfo, NULL, NULL, NULL, 0, 0);
    if (vers->entdata != NULL)
        filename = vers->entdata->user;
    else
        filename = finfo->file;

    if (vers->vn_user != NULL)
    { /* The Entries request. */
        send_to_server("Entry */", 0);
        send_to_server(filename, 0);
        send_to_server("*/", 0);
        send_to_server(vers->vn_user, 0);
        send_to_server("*/", 0);
        if (vers->ts_conflict != NULL)
        { /* The Entries request. */
            if (vers->ts_user != NULL &&
                strcmp(vers->ts_conflict, vers->ts_user) == 0)
                send_to_server("*/", 0);
            else
            { 
        send_to_server("Unlinked", 0);
    }
    send_to_server("*/", 0);
    send_to_server(vers->entdata != NULL ? vers->entdata->options

7
if (vers->request (*"Kopt"))
{
    char *opt;
    opt = wrap_rcs_option (filename, 1);
    send_to_server (opt, 0);
    free (opt);
}
else
{
    error (0, 0,
          "warning: ignoring -k options due to server limitations");
}

if (vers->ts_user == NULL)
{
    /*
    * Do we want to print "file was lost" like normal CVS?
    * Would it always be appropriate?
    */
    /* File no longer exists. Don't do anything, missing files just happen. */
}
else if (vers->ts_user == NULL
    && vers->ts_user->force
    && stcmp (vers->ts_user, vers->ts_rcs) != 0)
{
if (args->nocontents
    && supported_request (*"Is-modified")
{
    send_to_server (*"Is-modified ", 0);
    send_to_server (filename, 0);
    send_to_server (*"012", 1);
}
else
    send_modified (filename, finfo->fullname, vers);
}
else
{
    send_to_server (*"Unchanged ", 0);
    send_to_server (filename, 0);
    send_to_server (*"012", 1);
}

/* if this directory has an ignore list, add this file to it */
if (ignorelist)
{
    Node *p;
    p = getnode (1);
p->type = FILES;
p->key = xstrdup (info->file);
( void ) addnode ( ignlist, p);
}

if ( handling_remotes ) {
    /* Send the remote revisions needed for completion of this command */
    FILE* remote_rev_file = fopen ( CVSADM_REMOTES, "r" );
    char* line;
    int linelen,
    int line_allocated;

    if ( remote_rev_file != NULL ) {
        FILE* new_remote_rev_file; 
        rename ( CVSADM_REMOTES, CVSADM_REMOTES_BACKUP);
        new_remote_rev_file = fopen ( CVSADM_REMOTES, "r" );
        if ( new_remote_rev_file != NULL ) {
            while ( ( linelen = getline ( &line, &line_allocated, remote_rev_file ) ) > 0 ) {
                char* file = NULL;
                char* rev = NULL;
                char* data = NULL;

                file = line;
                rev = strchr ( file, '/' );
                if ( rev == NULL )
                    continue;

                *rev = '\0';
                rev++;

                data = strchr ( rev, '/' );
                if ( data == NULL )
                    continue;

                *data = '\0';
                data++;

                *strchr ( data, '\n' ) = '\0';

                if ( strcmp ( file, filename ) == 0 ) {
                    /* File matches */
                    sprintf ( fullname, "%s/%s", CVSADM, data );
                    send_remote_rev ( file, rev, fullname, vers);
                    free ( fullname);
                } else {
                    sprintf ( new_remote_rev_file, "%s\n", line );
                }

                fclose ( new_remote_rev_file);
                unlink ( CVSADM_REMOTES_BACKUP);

            }

            fclose ( remote_rev_file);

        } freevers (&vers);
        return 0;
    }

    int send_remote_rev ( char* file,
        char* rev,
        char* datafile,
        Vers** vers)
    {
        /* Send a remote revision to the server */
        struct stat sb;
        int fd;
        char* buf;
        char* mode_string;
        size_t bufsize;
        int bin;

        if ( trace )
            ( void ) fprintf ( stderr, " -> Sending remote revision of \"%s\" to server\n", file);

        /* Don't think we can assume stat exists. */
        if ( CVS_STAT ( datafile, &sb ) < 0 )
            error ( 1, errno, "reading %s", datafile);

        mode_string = mode_to_string ( sb.at.mode);

        /* Beware: on systems using CRLF line termination conventions,
        the read and write functions will convert CRLF to LF, so the
        number of characters read is not the same as sb.at.size. Text
        files should always be transmitted using the LF convention, so
        we don't want to disable this conversion. */
        bufsize = sb.at.size;
        buf = xmalloc ( bufsize );

        /* Is the file marked as containing binary data by the "-kb" flag? */

    }
If so, make sure to open it in binary mode: */

if (vers & vers->options)
    bin = !(strcmp (vers->options, "-KB"));
else
    bin = 0;

#endif BROKEN_READWRITE_CONVERSION

if (bin)
    /* If only stdin, not open/write/etc., do text/binary
     conversion, use convert_file which can compensate
     (FIXME: we could just use stdio instead which would
     avoid the whole problem). */
    char title[1024]; strcpy(title, datafile); strcat(title, ".CVSBFCTMP");
    convert_file (datafile, RDONLY, title, QnWONLY | QnCREAT | QnTRUNC | OPEN_BINARY);
    fd = CVS_OPEN (title, O_RDWR | OPEN_BINARY);
    if (fd < 0)
        error (1, errno, "reading %s", short_pathname);
} else
    fd = CVS_OPEN (datafile, O_RDONLY | OPEN_BINARY);
else
    fd = CVS_OPEN (datafile, O_RDONLY | (bin ? OPEN_BINARY : 0));
#endif

if (fd < 0)
    error (1, errno, "reading %s", datafile);

if (fgets (getline, level & sb.state_save > 100)
    size & newline = 0;
    read_and_gzip (fd, datafile, (unsigned char **)buf, &bufsize, &newsize, file_get_level);

    if (close (fd) < 0)
        error (0, errno, "warning: can't close %s", datafile);

    {
        char tmp[80];
        send_to_server ("Remote-revision ", 0);
        send_to_server (file, 0);
        send_to_server ("\012", 1);
        send_to_server (rev, 0);
        send_to_server ("\012", 1);
        send_to_server (mode_string, 0);
        send_to_server ("\012", 2);
        sprintf (tmp, "\012\012", (unsigned long) newsize);
        send_to_server (tmp, 0);
        send_to_server (buf, newsize);
    }
else
    {
        int newsize;

        {
            char *obuf = buf;
            int len;

            /* FIXME: This is gross. It assumes that we might read
             less than st.size bytes (true on NT), but not more.
             Instead of this we should just be reading a block of
             data (e.g. 8192 bytes), writing it to the network, and
             so on until EOF. */
            while (((len = read (fd, bufp, (buf + sb.state_size) - bufp)) > 0)
                  bufp += len;

                if (len < 0)
                    error (1, errno, "reading %s", datafile);

                newsize = bufp - buf;
            }

            if (close (fd) < 0)
                error (0, errno, "warning: can't close %s", datafile);

            { 
                char tmp[80];
                send_to_server ("Remote-revision ", 0);
                send_to_server (file, 0);
                send_to_server ("\012", 1);
                send_to_server (rev, 0);
                send_to_server ("\012", 1);
                send_to_server (mode_string, 0);
            }
static void sendproc (void *, int, char *, char *, List *);

5250 static void sendfilesdoneproc (callerdat, err, repository, update_, dir, entries);

5270 static int sendfiledoneproc (callerdat, err, repository, update_, dir, entries);

5280 static long sendfilesdoneproc (callerdat, err, repository, update_, dir, entries);

5290 static Dtype sendfilesdoneproc (callerdat, err, repository, update_, dir, entries);

5300 static long sendfilesdoneproc (callerdat, err, repository, update_, dir, entries);

*/
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/ 
*/
/* print the warm fuzzy message */
if (quiet)
    error (0, 0, "Ignoring \a", update_dir);
return (R_SKIP_ALL);
}

/* If the directory does not exist yet (e.g. "cvs update -d foo"),
* no need to send any files from it. If the directory does not
* have a CVS directory, then we pretend that it does not exist.
* Otherwise, we will fail when trying to open the Entries file.
* This case will happen when checking out a module defined as
* "-a .".
*/
cvsadm_name = xmalloc (strlen (dir) + sizeof (CVSADM) + 10);
sprintf (cvsadm_name, "%s/\a", dir, CVSADM);
dir_exists = isdir (cvsadm_name);
free (cvsadm_name);

/* initialize the ignore list for this directory */
ignlist = getlist ();

/* If there is an empty directory (e.g. we are doing 'cvs add' on a
* newly-created directory), the server still needs to know about it.
*/
if (!quiet)
{
    /* Get the repository from a CVS/Repository file whenever possible.
    * The repository variable is wrong if the names in the local
    * directory don’t match the names in the repository.
    */
    char *repos = NameRepository (dir, update_dir);
    sendrepository (dir, repos, update_dir);
    free (repos);
}
else
{
    /* It doesn’t make sense to send a non-existent directory,
     * because there is no way to get the correct value for
     * the repository (I suppose maybe via the expand-modules
     * request). In the case where the “obvious” choice for
     * repository is correct, the server can figure out whether
     * to recreate the directory; in the case where it is wrong
     * (that is, does not match what modules give us), we might as
     * well just fail to recreate it.
     *
     * Checking for noexec is a kludge for "cvs -n add dir". */
    /* Don’t send a non-existent directory unless we are building
     * new directories (build_dirs is true). Otherwise, CVS may
     * see a D line in an Entries file, and recreate a directory
     * which the user removed by hand. */
    if (args->build_dirs && !noexec)
        sendrepository (dir, repository, update_dir);
}
return (dir_exists ? R_PROCESS : R_SKIP_ALL);
}

/*
 * Send each option in a string to the server, one by one.
 * This assumes that the options are separated by spaces, for example
 * STRING might be "-foo -C5 -y".
 */
void send_option_string (string)
{
    char *copy;
    char *p;
    copy = xstrdup (string);
    p = copy;
    while (1)
    {
        char *s;
        char l;
        for (s = p; *s != ' ' && *s != '\0'; s++)
        {
            l = *s;
            *s = '\0';
            if (l != p)
                sendarg (p);
            if (l == '\0')
                break;
            p = s + 1;
        }
    }
}
void send_file_names(argc, argv, flags)
{ int i;
  int level;
  int max_level;

  /* The fact that we do this here as well as start-recursion is a bit
   of a performance hit. Perhaps worth cleaning up someday. */
  if (flags & SEND_EXPAND_WILD)
    expand_wild(argc, argv, &argc, &argv);

  max_level = 0;
  for (i = 0; i < argc; ++i)
  {
    level = pathname_levels(argv[i]);
    if (level > max_level)
      max_level = level;
  }

  if (max_level > 0)
  {
    if (supported_request("Max-dotdot"))
    {
      char buf[10];
      sprintf(buf, "%d", max_level);
      send_to_server("Max-dotdot ", 0);
      send_to_server(buf, 0);
      send_to_server("\012", 1);
    }
    else
    {
      /* "leading .." is not strictly correct, as this also includes
       * cases like "foo/..../". But trying to explain that in the
       * error message would probably just confuse users. */
      error(1, 0, "leading .. not supported by old (pre-Max-dotdot) servers");
    }
  }

  for (i = 0; i < argc; ++i)
  {
    char buf[1];
    char *p = argv[i];
    char *line = NULL;

    #ifdef FILENAMES_CASE_INSENSITIVE
    /* We want to send the file name as it appears
     * in CVS/Entries. We put this inside an ifdef
     * to avoid doing all these system calls in
     * cases where fncmp is just strcmp anyway. */
    /* For now just do this for files in the local
     * directory. Would be nice to handle the
     * non-local case too, though. */
    /* The isdir check could more gracefully be replaced
     * with a way of having Entries_Open report back the
     * error to us and letting us ignore existence error.
     * Or some such. */
    if (p == last_component(p) & & isdir (CVSADM))
    {
      List *entries;
      Node *node;

      /* If we were doing non-local directory,
       * we would save_wd, CVS_CHDIR
       * like in update.c:emptydir. */
      /* Note that if we are adding a directory,
       * the following will read the entry
       * that we just wrote there, that is, we
       * will get the case specified on the
       * command line, not the case of the
       * directory in the filesystem. This
       * is correct behavior. */
      entries = Entries_Open (0, NULL);
      node = findnode (entries, p);
      if (node != NULL)
      {
        line = xstrdup (node->key);
        p = line;
      }
    }
  }
5490 } } EntryProc(entries); }

sendif /* FILENAMES/ASE/NSENSITIVE */

send_to_server (*argv *, 0);

while (*p)
{
  if (*p != '\n')
  }

else if (ISDIRSEP (*p))
  { buf[0] = '/';
  send_to_server (buf, 1);
  }

else
  { buf[0] = *p;
  send_to_server (buf, 1);
  ++p;
  }

send_to_server (*argv *, 1);

if (line != NULL)
  free (line);

5502 if (flags & SEND_EXPAND_WILD)
  {
    int i;

    for (i = 0; i < argc; ++i)

      free (argv[i]);

    free (argv);
  }

5530 /* Send Repository, Modified and Entry. argc and argv contain only
the files to operate on (or empty for everything), not options.
local is nonzero if we should not recurse (-l option). flags &
SEND_BUILD_DIRS is nonzero if nonexistent directories should be
sent. flags & SEND_FORCE is nonzero if we should send unmodified
files to the server as though they were modified. flags &
SEND_NO_CONTENTS means that this command only needs to know
whether a file is modified, not the contents. Also sends Argument
lines for argc and argv, so should be called after options are sent. */

void send_files(argc, argv, local, aflag, flags)
int argc;
char **argv;
int local;
int aflag;
unsigned int flags;

{ struct send_data args;
int err;

5550 /*
  * aflag controls whether the tag/date is copied into the version.
  * But we don't actually use it, so I don't think it matters which we pass
  * for aflag here.
  */

  argc.build_dirs = flags & SEND_BUILD_DIRS;
  argc.force = flags & SEND_FORCE;
  argc.no_contents = flags & SEND_NO_CONTENTS;
  err = start_recursion
(send_fileproc, send_filesdoneproc,
5560 send_direct_proc, (DIRLEAVEPROC)NULL, (void *) &args,
argc, argv, local, WLOCAL, aflag, 0, (char *)NULL, 0);

if (err)
  error_exit ();

if (toplevel_repo == NULL)
  /*
  * This happens if we are not processing any files,
  * or for checkouts in directories without any existing stuff
  * checked out. The following assignment is correct for the
  * latter case, I don't think toplevel_repo matters for the
  * former.
  */

  toplevel_repo = strdup (CVSroot_directory);

  send_repository (**, toplevel_repo, ",");

}

void client_import_setup (repository)
char *repository;
{  
  if (toplevel_repo == NULL) /* should always be true */
    send_repo(repository ("*", repository, ")
  
}  
/*  
* Process the argument import file.  
*/  
int
client_process_import_file (message, vfile, vtag, targv, repository,  
repository_server, all_file_binary)

char *message;
char vfile;
char vtag;
int target;
char *targv[];
char repository;  
int all_file_binary;
{
  char *update_dir;
  char *fullname;
  
VersTS vers;
  assert (toplevel_repo != NULL);
  
  if (strncmp (repository, toplevel_repo, strlen (toplevel_repo)) != 0)  
    error (1, 0,  
      "internal error: pathname '%s' doesn't specify file in '%s'",  
repository, toplevel_repo);
  
  if (strncmp (repository, toplevel_repo) == 0)
    {
      update_dir = "*",
      fullname = xstrdup (vfile);
    }  
  else
    {
      update_dir = repository + strlen (toplevel_repo) + 1;
      fullname = xmalloc (strlen (vfile) + strlen (update_dir) + 10);  
      strcat (fullname, "/");
      strcat (fullname, update_dir);
  
  vers.options = xmalloc (4);  
  vers.options += strlen ("-kb") + 1 /*
  strcpy (vers.options, "-kb");
  
      send_repo(repository ("*", repository, update_dir);
  if (all_file_binary)
    {
      vers.options = xmalloc (4);  
      vers.options += strlen ("-kb") + 1 /*
      strcpy (vers.options, "-kb");
    }
  
  else
    {
      vers.options = wrap_options (vfile, 1);
    }
  
  if (vers.options != NULL)
    {
      if (supported_request ("Kopt"))
        {
          send_to_server ("Kopt", 0);
          send_to_server (vers.options, 0);
        }
      else
        error (0, 0,  
          "warning: ignoring -k options due to server limitations*");
      
      send_modified (vfile, fullname, allvers);
      if (vers.options != NULL)
        free (vers.options);
      free (fullname);
      return 0;
    }
  
void
client_import_done ()
{
  
  if (toplevel_repo == NULL)  
    /*
     * This happens if we are not processing any files,  
     * or for checkouts in directories without any existing stuff
     * checked out. The following assignment is correct for the
     * latter case; I don't think toplevel_repo matters for the  
     * former.
     */
    
    /* FIXME: "can't happen" now that we call client_modat the beginning. +*/
    toplevel_repo = xstrdup (CVSroot_directory);
    send_repo(repository ("*", toplevel_repo, ");
  
}
static void
notify_file (data, ent_list, short_pathname, filename)
char *data;
List *ent_list;
char *short_pathname;
char *filename;
{
FILE *fp;
FILE *newf;
sizet line_len = 8192;

char *line = xmalloc (line_len);
char *cp;
int nread;
int nwritten;

fp = open_file (CVSADM_NOTIFY, "r");
if (getline (&line, &line_len, fp) < 0)
{
  if (feof (fp))
    error (0, 0, "cannot read %s: end of file", CVSADM_NOTIFY);
  else
    error (0, errno, "cannot read %s", CVSADM_NOTIFY);
  goto error_exit;
}

if (cp == NULL)
{
  error (0, 0, "malformed %s file", CVSADM_NOTIFY);
  goto error_exit;
}

if (strcmp (filename, line + 1) != 0)
{
  error (0, 0, "protocol error: notified %s, expected %s", filename,
         line + 1);
}

if (getline (&line, &line_len, fp) < 0)
{
  if (feof (fp))
  {
    free (line);
    if (fclose (fp) < 0)
      error (0, errno, "cannot close %s", CVSADM_NOTIFY);
    if (CVS_UNLINK (CVSADM_NOTIFYTMP) < 0)
      error (0, errno, "cannot remove %s", CVSADM_NOTIFY);
    return;
  }
  else
  {
    error (0, errno, "cannot read %s", CVSADM_NOTIFY);
    goto error_exit;
  }

newf = open_file (CVSADM_NOTIFYTMP, "w");
if (fputs (line, newf) < 0)
{
  error (0, errno, "cannot write %s", CVSADM_NOTIFYTMP);
  goto error2;
}

while ((nread = fread (line, 1, line_len, fp)) > 0)
{
  p = line;
  while ((nwritten = fwrite (p, 1, nread, newf)) < 0)
  {
    nread -= nwritten;
    p += nwritten;
  }
  if (ferror (newf))
  {
    error (0, errno, "cannot write %s", CVSADM_NOTIFYTMP);
    goto error2;
  }
  if (ferror (fp))
  {
    error (0, errno, "cannot read %s", CVSADM_NOTIFY);
    goto error2;
  }
  if (fclose (newf) < 0)
  {
    error (0, errno, "cannot close %s", CVSADM_NOTIFYTMP);
    goto error_exit;
  }
}
free (line);
if (fclose (fp) < 0)
{
  error (0, errno, "cannot close %s", CVSADM_NOTIFY);
}
return;

/* In this case, we want rename_file() to ignore noexec. */
int saved_noexec = noexec;
noexec = 0;
rename_file(CVSADM_NOTIFYTMP, CVSADM_NOTIFY);
noexec = saved_noexec;
}

5770  return;
5771  error2:  (void) fclose (newf);
5772  error_exit:  free (line);
5773  (void) fclose (fp);
}

5780  static void  handle_notified (args, len)
5781  {
5782    call_in_directory (args, notified_file, NULL);
5783  }

5790  void  client_notify (repository, update_dir, filename, notif_type, val)
5791  {
5792    void  send_to_server (string, options);
5793    char  buf[2];
5794    send_to_server (repository, (*, repository, update_dir));
5795    send_to_server (Notify *, 0);
5796    send_to_server (filename, 0);
5797    send_to_server ("\012", 1);
5798    send_to_server (buf[0] = notif_type;
5799    send_to_server (buf[1] = '0';
5800    send_to_server (buf[1], 1);
5801    send_to_server ("\t", 1);
5802    send_to_server (val, 0);
5803  }

5810  /* * Send an option with an argument, dealing correctly with newlines in
* the argument. If ARG is NULL, forget the whole thing.
* */
5811  void  option_with_arg (option, arg)
5812  {
5813    char  *option;
5814    char  *arg;
5815    if (arg == NULL)
5816      return;
5817    send_to_server ("Argument ", 0);
5818    send_to_server (option, 0);
5819    send_to_server ("\012", 1);
5820    send_to_server (arg, 0);
5821    send_to_server ("\t", 1);
5822    send_to_server (val, 0);
5823  }

5830  /* * Send a date to the server. The input DATE is in RCS format. The
* time will be GMT.
* We then convert that to the format required in the protocol
* (including the "-D" option) and send it. According to
* cvscvsclient.texi, RFC 822/1123 format is preferred, but for now we
* use the format that we always have, for
* conservatism/laziness/paranoia. As far as I know all servers
* support the RFC 822/1123 format, so probably there would be no
* particular danger in switching. */

5840  void  client_senddate (date)
5841  {
5842    const char  *date;
5843    int  year, month, day, hour, minute, second;
5844    char  buf[100];
5845    if (sscanf (date, SDATEFORM, &year, &month, &day, &hour, &minute, &second) != 6)
5846      error (1, 0, "client_senddate: sscanf failed on date");
5847    }
sprint (buf, "%Y/%m/%d %H:%M:%S", month, day, year, 
hour, minute, second); 
} }

void send_init_command ()
{
  /* This is here because we need the CVSroot directory variable. */
  send_server ("init", 5);
  send_server (CVSroot, 0); 
  send_server ("012", 0); 
}

typedef struct 
{
  char* repository; /* Location of the file in the repository */ 
  char* server; /* Name of the server */ 
  char* root; /* Location of the repository on the server */ 
  int done; 
  char* file; 
  char* revision; 
  int handling; 
} remote_node; 

typedef struct 
{
  int (*func) (); 
  int argc; 
  char** argv; 
  int first_file_arg; 
  int done; 
} 

5870 process_remotes_closure; 

static List* outstanding_remotes = NULL; 
int first_file_arg; 
int handling_remotes; 
int fetch_remotes; 

int process_remotes (func (Node*, n, void* c));

enum 
{
  REMOTE_HANDLING_FETCH, 
  REMOTE_HANDLING_RERUN, 
  REMOTE_HANDLING_FETCH_WAITING, 
  REMOTE_HANDLING_FETCHING, 
  REMOTE_HANDLING_ADD_REMOTE_BRANCH 
}; 

void add_remote (char* file, char* server, char* root, char* repository) 
{
  remote_node* remote = malloc (sizeof (remote_node)); 
  Node* node = getnode (); 
  node -> data = (char*) remote; 
  if (outstanding_remotes == NULL) {
    outstanding_remotes = getlist ();
  } 
  if (remote != NULL) {
    char* revision = strchr (file, ' ') + 1; 
    remote -> server = xstrdup (server); 
    remote -> done = 0; 
    remote -> file = xstrdup (file); 
    remote -> revision = xstrdup (revision); 
    remote -> file [revision - file - 1] = '\0'; 
    remote -> repository = xstrdup (repository); 
    /* There are two things we may need to do with a remote: rerun the command 
    * on a different server, or fetch the contents of the remote and rerun the 
    * command on the original server. Fetching is used by commands which need 
    * access to some revision off a different server, i.e. diff and update. */
  } 
  if (fetch_remotes) {
    remote -> handling = REMOTE_HANDLING_FETCH; 
  } else {
    remote -> handling = REMOTE_HANDLING_RERUN; 
  } 
  addnode (outstanding_remotes, node); 
}

5930 void add_remote_tag (char* file, char* root, char* server, char* repository, char* revision) 
{
  char* host = xstrdup (revision) + 1; 
  char* path; 
  char* rev; 
  remote_node* remote = malloc (sizeof (remote_node)); 
  Node* node = getnode ();
}
host = rev + 1;
path = strdup (host, ':');
if (path == NULL) {
  error (1, 1, "Invalid remote tag");
} else {
  *path = '0';
  path++;
}

rev = strchr (path, ':');

if (rev == NULL) {
  error (1, 1, "Invalid remote tag");
} else {
  *rev = '0';
  rev++;
}

node = data = (char*) remote;

if (outstanding_remotes == NULL) {
  outstanding_remotes = getlist();
}

if (remote != NULL) {
  remote = strdups (path);
  remote = server = strdups (host);
  remote = done = 0;
  remote = file = strdups (file);
  remote = revision = strdups (rev);
  remote = repository = repository;
  remote = handling = REMOTE_HANDLING_ADD_REMOTE_BRANCH;

  addnode (outstanding_remotes, node);
}

void add_node_to_fetching (remote* node = old_remote)
{
  remote = node = malloc (sizeof remote);
  Node* node = getnode (node);
  node = data = (char*) remote;

  if (outstanding_remotes == NULL) {
    outstanding_remotes = getlist();
  }

  if (remote != NULL) {
    remote = root = strdups (old_remote -> root);
    remote = server = strdups (old_remote -> server);
    remote = done = 0;
  }

  remote = file = strdups (old_remote -> file);
  remote = revision = strdups (old_remote -> revision);
  remote = repository = strdups (old_remote -> repository);
  remote = handling = REMOTE_HANDLING_FETCHING;

  /* always add at front because we want them to be seen first, before their corresponding fetch entries */
  addnode_at_front (outstanding_remotes, node);
}

int client_process_remotes (int *func (), int argc, char** argv)
{
  /* Note that completed entries are not removed from the list. This is because there is no way to remove items from the list while iterating */
  process_remotes = closure closure = { func, argc, argv, 0, 1 };
  handling = 1;

  /* We dont' want any subsequently executed commands to affect this */
  closure.first_filearg = first_filearg;

  do {
    /* reset the fetch_remotes flag because the function we invoke to process remotes will set it as appropriate */
    fetch_remotes = 0;
    /* assume we are done */
    closure.done = 1;
  }

  /* Always start with the initial arguments */
  closure.argc = argc;
  closure.argv = argv;
  closure.func = func;

  /* make the temporary file */
  unlink (CVSADM\REP\REMOTE);

  /* go through the list and figure out what to do with it */
walklist (outstanding_remotes, process_remotes, func, &closure);

/** If there are things not done, call the function (which
 may have been changed ) */
if (!closure.done)
    {(*closure.func) (closure.argc, closure.argv);
} while (!closure.done);

handling_remotes = 0;
unlink (CVSADM_REMOTE);

6040  return 0;  /* we are done */
}

int remote_loop (int argc, char** argv)
{
    return 0;
}

int process_remotes_func (Node* n, void* c)
{
    process_remotes_closure* closure = c;
    remote_node* remote = (remote_node*) (n -> data);

    /* For now, each pass through the list only does one node, but that doesn't have to be
 the case in general */
    if (!closure -> done)
        return 0;

6060  /* Only process a node if it's still pending */

    if (!remote -> done) {
        switch (remote -> handling) {
            case REMOTE_HANDLING_RERUN:
                {  /* If we are rerunning the command on a different server, setup the new arguments and root */
                    int err = setup_root (remote);
                    if (err == 0) {
                        err = setup_args (closure, remote);
                    }

                    /* inform the caller that a command needs to be run */
                    closure -> done = 0;

                    /* Mark the node as done whether an error occurred or not - to avoid
 retrying. If we need to go to a different server to get the file,
 a new entry will appear in the remotes list anyway */
                    remote -> done = 1;
                    break;
                }
                
            case REMOTE_HANDLING_FETCH:
                {  /* If we need to fetch a file before we can rerun the command, then we create a new entry
 in the remotes list, change the state of this entry, and move along. The next
 time through the list we will fetch the file and then we can retry this operation */
                remote -> handling = REMOTE_HANDLING_FETCH_WAITING;
                add_remo tes_for_fetching (remote);

                closure -> func = remote_loop;
                closure -> done = 0;
                break;
            }

            case REMOTE_HANDLING_FETCHING:
                {  /* This is a file which needs to be fetched in order to complete some other operation in the
 list. We get it using update to stdout, but we send the output into a file in the CVS directory.
 After we are done, we find the corresponding "fetch" entry and let it know it can proceed */
                    int err = setup_root (remote);
                    if (err == 0) {
                        err = setup_args_fetch (closure, remote);
                        closure -> done = 0;
                        remote -> done = 1;
                    }
                    break;
                }

            case REMOTE_HANDLING_FETCH_WAITING:
                {  /* When we get to this node, the dependents have already been fetched, so we can just rerun
 the command. The command itself is responsible for sending the remote revisions to the server */
                    int err = setup_root (remote);
                    if (err == 0) {
                        err = setup_args (closure, remote);

                    /* inform the caller that a command needs to be run */
                    closure -> done = 0;

                    /* Mark the node as done whether an error occurred or not - to avoid
 retrying. If we need to go to a different server to get the file,
 a new entry will appear in the remotes list anyway */
                    remote -> done = 1;
                    break;
                }

        }
    }

} /* remote_loop */

client.c o 177
case REMOTE_HANDLING\_ADD\_REMOTE\_BRANCH: {
    /* When we are adding a remote branch, we need to create a file on the destination server
     * which have one revision with exactly the same contents as the working files
     * and make sure that that revision has the appropriate number and tag. To do that,
     * the add command takes a new argument which allows the client to specify the remote
     * branchpoint */
    int err = setup\_root (remote);
    if (err == 0) {
        err = setup\_args\_remote\_branch (closure, remote);
    }
    closure \=> done = 0;
    remote \=> done = 1;
    break;
}
return 0;
*/

/* Change the CVS root in the global variables (grumble grumble) to
match the CVS root needed for this remote file */
int setup\_root (remote\_node= remote)
{
    FILE* fpin;
    char* new\_root = xmalloc (1 +
        strlen (method\_names [CVSroot\_method]) + 1 +
        ((CVSroot\_username != NULL) ? strlen (CVSroot\_username) : 0) + 1 +
        strlen (remote \=> server) + 1 +
        strlen (remote \=> root) + 10);
    sprintf (new\_root, "\%s\%s\%s\%s\%s",
        method\_names [CVSroot\_method],
        CVSroot\_username != NULL ? CVSroot\_username : *,
        (CVSroot\_username != NULL) ? CVSroot\_username : *,
        remote \=> server,
        remote \=> root);

    parse\_or\_root (new\_root);
    free (new\_root);
    fpin = fopen (CVSADM\_REP\_REMOTE, *r*);
    if (fpin == NULL) {
        fpin = fopen (CVSADM\_REP\_REMOTE, *w*);
        if (fpin != NULL) {
            fwrite (fpin, 1, 1, fpin);
            fclose (fpin);
        } else {
            fclose (fpin);
        }
        return 0;
    }

    int setup\_args (process\_remotes\_closure= closure, remote\_node= remote)
    {
        int i;

        int new\_argc = closure \=> first\_file\_arg;
        char** new\_argv = (char**) xmalloc (new\_argc * sizeof (char*));
        for (i = 0; i < new\_argc; i++) {
            if (i < closure \=> first\_file\_arg) {
                new\_argv [i] = xstrdup (closure \=> argv [i]);
            } else {
                new\_argv [i] = xstrdup (remote \=> file);
            }
        }

        closure \=> argc = new\_argc;
        closure \=> argv = new\_argv;

        return 0;
    }

    int remote\_fetch\_update\_wrapper (int argc, char** argv);
int remote\_branch\_add\_wrapper (int argc, char** argv);

    /* We always fetch a file with cvs update -p -r rev filename */
    int setup\_args\_fetch (process\_remotes\_closure= closure, remote\_node= remote)
    {
        int i;
        char* args [5] = { "update\", "-r", xstrdup (remote \=> revision), xstrdup (remote \=> file), NULL };

        closure \=> func = &remote\_fetch\_update\_wrapper;
        closure \=> argc = 4;
        closure \=> argv = (char**) xmalloc (sizeof (char*) * (closure \=> argc + 1));
        for (i = 0; i <= closure \=> argc; i++) {
            closure \=> argv [i] = args [i];
        }

        return 0;
    }
int setup_remote_branch (process_remotes_closure * closure, remote_node * remote)
{
    int i;
    char * args[5] = { "add", "-r", xstrdup (remote->revision), xstrdup (remote->file), NULL };
    closure->func = &remote_branch_add_wrapper;
    closure->argc = 4;
    closure->argv = (char**) xmalloc (sizeof (char*) * (closure->argc + 1));
    for (i = 0; i < closure->argc; i++) {
        closure->argv[i] = args[i];
    }
    return 0;
}

static int fetching_remotes = 0;
int fetch_remote (struct file_info* finfo)
{
    return 1;
}

int remote_fetch_update_wrapper (int argc, char** argv)
{
    int result;
    printf ("remote_fetch_update_wrapper %s %s %s %s
", argv[0], argv[1], argv[2], argv[3]);
    fetching_remote = 1;
    result = update (argc, argv);
    fetching_remote = 0;
    return result;
}

int remote_branch_add_wrapper (int argc, char** argv)
{
    int result;
    printf ("remote_branch_add_wrapper %s %s %s %s
", argv[0], argv[1], argv[2], argv[3]);
    adding_remote = 1;
    result = add (argc, argv);
    adding_remote = 0;
    return result;
}

#endif /* CLIENT_SUPPORT */
A.9 client.h

/* Interface between the client and the rest of CVS. */

/* Stuff shared with the server. */
extern char *mode_string PROTO((mode));
extern int change_mode PROTO((char *, char *, int));
extern int gssapi_level;
extern int file_gssapi_level;
extern int filter_gssapi PROTO((int, int, pid_t *));
extern int filter_through_gssapi PROTO((int, int, pid_t *));

#endif defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT)

/* Whether the connection should be encrypted. */
extern int cvsextract;
/* Whether the connection should use per-packet authentication. */
extern int cvsauthenticate;

#endif HAVE_KERBEROS

/* We can't declare the arguments without including krb.h, and I don't
   want to do that in every file. */
extern struct buffer *krb_menu;
buffer_initialize();
#endif /* HAVE_KERBEROS */

#endif HAVE_GSSAPI

/* Set this to turn on GSSAPI encryption. */
extern int cvs_gssapi_encrypt;
#endif /* HAVE_GSSAPI */
#endif /* ENCRYPTION */

#endif HAVE_GSSAPI

/* We can't declare the arguments without including gssapi.h, and I
don't want to do that in every file. */
extern struct buffer *cvs_gssapi_wrap;
buffer_initialize();
#endif /* HAVE_GSSAPI */
#endif defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT) */

#endif CLIENT_SUPPORT

/* Flag variable for seeing whether the server has been started yet.
   As of this writing, only edit.c:notify/heck() uses it.
*/
extern int server_started;
/* Is the -P option to checkout or update specified? */
extern int client_runcmdline;

#endif AUTH_CLIENT_SUPPORT

extern int use_authenticating_server;
void connect_to_server PROTO((int *tofdp, int *fromfdp, int verifyonly,
   int method));
#endif CVS_AUTH_PORT
#define CVS_AUTH_PORT 6011
#endif /* CVS_AUTH_PORT */
#endif /* AUTH_CLIENT_SUPPORT */

#endif defined (AUTH_SERVER_SUPPORT) || defined (SERVER_SUPPORT) &&
defined (HAVE_KERBEROS) || defined (SERVER_SUPPORT) &&
defined (HAVE_KERBEROS) || defined (SERVER_SUPPORT) &&
defined (HAVE_KERBEROS)

extern void server_authenticate_connection PROTO ((void));
#endif

/* Talking to the server. */
void send_to_server PROTO((char *str, size_t len));
void read_from_server PROTO((char *buf, size_t len));

/* Internal functions that handle client communication to server, etc. */
int supported_request PROTO((char *));
void option_with_arg PROTO((char *option, char *arg));

/* Get the responses and then close the connection. */
extern int get_response_and_close PROTO ((void));

/* Start up the connection to the server on the other end. */
void start_server PROTO ((void));
# Client Support

### Global Definitions

```c
#define SEND_EXPAND_WILD 1
#define SEND_BUILD_DIRS 1
#define SEND_FORCE 2
#define SEND_NO_CONTENTS 4
```

### Proto Declarations

```c
void send_file_names_PROTO((int argc, char **argv, unsigned int flags));
```

```c
void send_file_PROTO((int argc, char **argv, unsigned int flags));
```

```c
void send_files_PROTO((int argc, char **argv, int local, int aflag, unsigned int flags));
```

```c
void send_arg_PROTO((char *string));
```

```c
void send_option_STRING_PROTO((char *string));
```

```c
extern void send_a_repository_PROTO((char *, char *, char *));
```

```c
typedef /* CLIENT SUPPORT */

enum {
    response_type_normal,
    response_type_ok,
    response_type_error
} type;
```

### Response Structure

```c
struct response {
    /* Name of the response. */
    char *name;

define CLIENT_SUPPORT
    /* Function to carry out the response. ARG is the text of the
    * command after name and, if present, a single space, have been
    * stripped off. The function can scribble into ARG if it wants.
    * Note that although LEN is given, ARG is also guaranteed to be
    * '"0' terminated.
    */
    void (*func)(char *args, int len);

    /* ok and error are special; they indicate we are at the end of the
    * responses, and error indicates we should exit with nonzero
    * exitstatus.
    */
    enum {
        response_type_normal, response_type_ok, response_type_error
    } type;

defendif

    /* Used by the server to indicate whether response is supported by
    * the client, as set by the Valid-responses request. */
    enum {
        rs_essential,
        rs_optional,
        rs_not_supported
    } status;
};
```

### External Definitions

```c
extern struct response responses[];
```
#ifdef CLIENT_SUPPORT
extern void client_senddate PROTO((const char *date));
extern void client_expand_modules PROTO((int argc, char **argv, int local));
extern void client_send_expansions PROTO((int local, char **where, int build_dir));
extern void client_nonexpanded_setup PROTO((void));
extern int client_process_remotes (int (*func) (), int argc, char **argv);
extern int adding_remote;
struct file_info; // avoid circular includes
extern int fetch_remote (struct file_info);
extern void send_init_command PROTO ((void));
extern char **failed_patches;
extern int failed_patches_count;
extern int tag_is_remote;
extern char *toplevel_wd;
extern void client_import_setup PROTO((char *repository));
extern int client_process_import_file
PROTO((char *message, char *vfile, char *vtag,
int targc, char **targv[], char *repository, int all_files_binary));
extern void client_import_done PROTO((void));
extern void client_notify PROTO((char *, char *, char *, int, char *));
sendif /* CLIENT_SUPPORT */
#endif
A.10 commit.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Commit Files
 * "commit" commits the present version to the RCS repository, AFTER
 * having done a test on conflicts.
 * The call is: cvs commit [options] files...
 */

#include <assert.h>
#include "cvs.h"
#include "edit.h"
#include "getline.h"

static Dtype check_direntproc PROTO ((void *callerdat, char *dir,
    char *repos, char *update_dir,
    List *entries));
static int checkfileproc PROTO ((void *callerdat, struct fileinfo *finfo);
static int checkfilesdoneproc PROTO ((void *callerdat, int err,
    char *repos, char *update_dir,
    List *entries));
static int checkaddfile PROTO ((char *file, char *repository, char *tag,
    char *options, RCSNode **rcsnode));
static Dtype commit_direntproc PROTO ((void *callerdat, char *dir,
    char *repos, char *update_dir,
    List *entries));
static int commitfilesdoneproc PROTO ((void *callerdat, char *dir,
    int err, char *update_dir,
    List *entries));
static int commit_addfileproc PROTO ((void *callerdat, struct fileinfo *finfo);
static int commitfilesdoneproc PROTO ((void *callerdat, int err,
    char *repository, char *update_dir,
    List *entries));
static int finaladdPROC ((struct fileinfo *finfo, char *revision, char *tag,
    char *options));
static int findmaxrev PROTO ((Node *p, void *closure));
static int lock_RCSPROC ((char *user, RCSNode *nodes, char *rev,
    char *repository));
static int precommit_list_proc PROTO ((Node *p, void *closure));
static int precommit_proc PROTO ((char *repository, char *filter));
static int remove_filePROC ((struct fileinfo *info, char *tag,
    char *message));
static void fix_RCSnodesPROC ((char *rcs, char *user));
static void fixaddfilePROC ((char *file, char *repository));
static void fixbranchPROC ((char *file, char *repository));
static void unlockrcsPROC ((RCSNode *rcs));
static void unlockprocPROC ((Node *p));
static void masterlist_unlockprocPROC ((Node *p));
static char *locate_RCSPROC ((char *file, char *repository));

struct commit_info
{
    Dtype status; /* as returned from classify_file() */
    char *rev; /* a numeric rev, if we know it */
    char *tag; /* any sticky tag, or -r option */
    char *options; /* Any sticky -k option */
};
struct master_lists
{
    List *ulist; /* list for UpdateLogfile */
    List *clist; /* list with commit_info structs */
};

static int force = 0;
static int got_message;
static int run_module = 0;
static int aflag = 0;
static char *saved_tag;
static char *write_dirtag;
static int write_dirtagbranch;
static char *logfiles;
static List *mulist;
static List *saved_list;
static char *saved_message;
static time_t last_register_time;
static const char *const commit_usage[] = {
/* Do not run the module program (if any). */
/* Process directories recursively. */
/* Local directory only (not recursive). */
/* Force the file to be committed; disables recursion. */
/* Read the log message from file. */
/* msg 

"\t-R \tProcess directories recursively."
"\t-m \tLog message."
"\t-F \tRead the log message from file."
"\t-f \tForce the file to be committed; disables recursion."
"\t-l \tLocal directory only (not recursive)."
"\t-n \tDo not run the module program (if any)."
"Examining %s"
exit (0, 0, "Examining %s", update_dir);

static Dtyle find_direct_proc PROTO ((void *callerdat, char *dir, char *repository, char *update_dir, List *entries));

static Dtyle find_direct_proc (callerdat, dir, repository, update_dir, entries)
void *callerdat;
char *dir;
char *repository;
char *update_dir;
List *entries;
{
  struct find_data *find_data = (struct find_data *)callerdat;
  /* This check seems to slowly be creeping throughout CVS (update
   is that it (or some variant thereof) should go in all the
   dirent procs. Unless someone has some better idea... */
  if (isdir (dir))
    return (R_SKIP_ALL);
  /* Initialize the ignore list for this directory */
  find_data->ignorelist = getlist ();
  /* Print the same warm fuzzy as in check direntproc, since that
   code will never be run during client/server operation and we
   want the messages to match. */
  if (!quiet)
    error (0, 0, "Examining %s", update_dir);
  return R_PROCESS;
}

/* As a static until we get around to fixing ignore files to pass
it along as an argument. */
static struct find_data *find_data_static;

static void find_ignproc PROTO ((char *, char *));

static void find_ignproc (file, dir)
char *file;
char *dir;
{
struct question *p;

p = (struct question *) xmalloc (sizeof (struct question));
p->dir = xstrdup (dir);
p->repos = xstrdup (find_data->repository);
p->file = xstrdup (file);
p->next = find_data->questionables;
find_data->questionables = p;
}

static int find_filedoneproc PROTO ((void *callerdat, int err, char *repository, char *update_dir, List *entries));

static int
find_filedoneproc (callerdat, err, repository, update_dir, entries)
void *callerdat;
int err;
char *repository;
char *update_dir;
List *entries;
{
    struct find_data *find_data = (struct find_data *) callerdat;
    find_data->repository = repository;

    /* if this directory has an ignore list, process it then free it */
    if (find_data->ignlist)
    {
        find_data_static = find_data;
        ignore_files (find_data->ignlist, entries, update_dir, find_ignoreproc);
        dellist (&find_data->ignlist);
    }

    find_data->repository = NULL;

    return err;
}

static int find_fileproc PROTO ((void *callerdat, struct file_info *finfo));

/* Machinery to find out what is modified, added, and removed. It is
possible this should be broken out into a new classify function;
merging it with classify file is almost sure to be a mess, though,
because classify file has all kinds of repository processing. */

static int
find_fileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;
{
    Version *vers;
    enum classify_type status;
    Node *node;
    struct find_data *args = (struct find_data *) callerdat;
    struct logfile_info *data;
    struct file_info xinfo;

    /* if this directory has an ignore list, add this file to it */
    if (args->ignlist)
    {
        Node *p;

        p = getnode ();
        p->type = FILES;
        p->key = xstrdup (finfo->file);
        if (addnode (args->ignlist, p) != 0)
            freenode (p);
    }

    xinfo = *finfo;
    xinfo.repository = NULL;
    xinfo.repos = NULL;

    vers = Version (*versinfo, NULL, saved_tag, NULL, 0, 0);
    if (vers->ts_user == NULL
        && vers->vn_user != NULL
        && vers->vn_user[0] == 'A')
    {
        /* FIXME: If vn_user is starts with 'A' but ts_user is non-NULL,
        what classify file does is print "%s should be removed and is still there".
        I'm not sure what it does then. We probably should do the same. */
        status = T_FILEMOVED;
    }
    else if (vers->vn_user == NULL)
    {
        if (vers->ts_user == NULL)
            error (0, 0, "nothing known about 'Ts'", finfo->fullname);
        else
            error (0, 0, "use 'Ts add' to create an entry for 'Ts'",
                   program_name, finfo->fullname);

        return 1;
    }
else if (vers->tv_user != NULL
  && vers->tv_user == NULL
  && vers->tv_user[0] == '0')
    /* FIXME: If tv_user is "0" but tv_user is NULL, what classify
     * does is print "new-born file has disappeared" and removes the entry.
     * We probably should do the same. */
    status = T_AWWDED;
else if (vers->tv_user != NULL
  && vers->tv_user == NULL
  && (args->force || strcmp (vers->tv_user, vers->tv_user) != 0))
    /* If we are forcing commits, pretend that the file is
     * modified. */
    status = T_MODIFIED;
else
  {
    /* This covers unmodified files, as well as a variety of other
     * cases. FIXME: we probably should be printing a message and
     * returning 1 for many of those cases (but I'm not sure
     * exactly which ones). */
    return 0;
  }
node = getnode();
node->key = xstrdup (info->fullname);

  data = (struct logfile_info *) xmalloc (sizeof (struct logfile_info));
data->type = status;
data->tag = xstrdup (vers->tag);
data->revOld = data->revNew = NULL;
node->type = UPDATE;
node->delproc = update_delproc;
node->data = (char *) data;
(void) addnode (args->ulist, node);
++args->argc;
freevers = (kvers);
return 0;
}

static int copy_ulist PROTO ((Node *, void *));

static int

  copy_ulist (node, data)
  
  Node *node;
  void *data;
  
  {  
    struct find_data *args = (struct find_data *)data;
    struct addnode (args->arg0, args->arg1++) = node->key;
    return 0;
  }

sendif /* CLIENT_SUPPORT */

int

  commit (argc, argv)
  
  int argc;
  char **argv;
  
  {  
    int c;
    int err = 0;
    int local = 0;
    
    if ((argc -= -1)
        usage (commit_usage);

#define CVS_BADROOT
  /*
   * For log purposes, do not allow "root" to commit files. If you look
   * like root, but are really logged in as a non-root user, it's OK.
   */
  */
  /* FIXME: Shouldn't this check be much more closely related to the
     * readonly user stuff (CVSROOT/readonly, &c). That is, why should
     * root be able to "cvs root", "cvs import", &c, but not "cvs ci"? */
  if (geteuid () == (uid_t) 0)
    {  
      struct passwd *pw;
      
      if ((pw = (struct passwd *) getpwnam (getcaller ())) == NULL)
        error (1, 0, "you are unknown to this system");
      else if (pw->pw_uid == (uid_t) 0)
        error (1, 0, "cannot commit files as 'root'");

    
  sendif /* CVS_BADROOT */

  optind = 0;
  while ((c = getopt (argc, argv, "+nlRm:fF:r:")) != -1)
  {  
    switch (c)
    {  

360  case 'n':
    run_module_prog = 0;
    break;
  case 'm':
    ifdef FORCE_USE_EDITOR
      use_editor = 1;
    else
      use_editor = 0;
    endif
    if (saved_message)
      free(saved_message);
      saved_message = NULL;
    endif
    saved_message = xstrdup(optarg);
    break;
  case 'r':
    if (saved_tag)
      free(saved_tag);
      saved_tag = xstrdup(optarg);
    break;
  case 'l':
    local = 1;
    break;
  case 'k':
    local = 0;
    break;
  case 'f':
    force_ci = 1;
    local = 1;
    /* also disable recursion */
    break;
  ifdef FORCE_USE_EDITOR
    use_editor = 1;
  else
    use_editor = 0;
  endif
  logfile = optarg;
  break;
  case 'F':
    default:
      usage (commit_usage);
      break;
    }
 )argc -= optind;
  argv += optind;

  /* numeric specified revision means we ignore sticky tags... */
  if (saved_tag && isdigit(*saved_tag))
    aflag = 1;
    /* strip trailing dots */
    while (*saved_tag[strlen(saved_tag)] == '.')
      saved_tag[strlen(saved_tag)] = 0;
    }
  /* some checks related to the "F logfile" option */
  if (logfile)
    {
      int n, logfd;
      struct stat statbuf;
      if (saved_message)
        error (1, 0, "cannot specify both a message and a log file");
        /* FIXME: Why is this binary? Needs more investigation. */
        if ((logfd = CVS_OPEN (logfile, O_RDONLY | OPEN_BINARY)) < 0)
          error (1, errno, "cannot open log file ", logfile);
        if (fstat(logfd, &statbuf) < 0)
          error (1, errno, "cannot find size of log file ", logfile);
        saved_message = xmalloc (statbuf.st_size + 1);
        /* FIXME: Should keep reading until EOF, rather than assuming
the first read gets the whole thing. */
        if ((n = read (logfd, saved_message, statbuf.st_size + 1)) < 0)
          error (1, errno, "cannot read log message from ", logfile);
      (void) close (logfd);
      saved_message[n] = '\0';
    }
  ifdef CLIENT_SUPPORT
    if (client_active)
      {
        struct finddata find_args;
        }
Ign_setup();

find_arg.ulist = getlist();
find_arg.argv = 0;
find_arg.questionables = NULL;
find_arg.ignorelist = NULL;
find_arg.repository = NULL;

/* It is possible that only a numeric tag should set this.
   I haven't really thought about it much.
   Anyway, I suspect that setting it unnecessarily only causes
   a little unneeded network traffic. */
find_arg.force = forceargv | saved_tag != NULL;

err = start_recursion(find_sound, find_end, endproc,
                      find_sound, DIRLEAVEPROC) NULL,
                      (void *)find_args.argv, argv, local, 
                      WLOCAL, 0, 0,
                      (char *)NULL, 0);

if (err)
   error (1, 0, "correct above errors first!");

if (find_arg.argv == 0)
   /* Nothing to commit. Exit now without contacting the
      server (note that this means that we won't print "?
      foo" for files which merit it, because we don't know
      what is in the CVSROOT/cvsignore file). */
   return 0;

/* Now we keep track of which files we actually are going to
   operate on, and only work with those files in the future.
   This saves time: we don't want to search the file system
   of the working directory twice. */
find_arg.argv = (char **)malloc ((find_arg.argv + sizeof (char *)));
find_arg.argv[0] = 0;
walklist(find_arg.ulist, copy_olist, &find_arg);

/* Do this before calling do_editor; don't ask for a log
   message if we can't talk to the server. But do it after we
   have made the checks that we can locally (to more quickly
   catch syntax errors, the case where no files are modified,
   added or removed, etc.).

   On the other hand, calling start_server before do_editor
   means that we chew up server resources the whole time that
   the user has the editor open (hours or days if the user
   forgets about it), which seems dubious. */
start_server();

/*
 * We do this once, not once for each directory as in normal CVS.
 * The protocol is designed this way. This is a feature.
 */
if (use_editor)
do_editor (*argv, &saved_message, (char *)NULL, find_arg.ulist);

/* Run the user-defined script to verify/check information in
   the log message
*/
do_verify(saved_message, (char *)NULL);

/* We always send some sort of message, even if empty. */
/* FIXME: is that true? There seems to be some code in do_editor
   which can leave the message NULL. */
/* optionwitharg (*argv, saved_message);

/* OK, now process all the questionable files we have been saving
up. */
{
   struct question *q;
   struct question *p;

   p = find_arg.questionables;
   while (p != NULL)
      {
         if (ignore_inhibit_server || supported_request ("questionable"))
            {
               cvs_output ("?? \", 2);
               if (p->dir[0] != '\0')
                    {
                       cvs_output (p->dir, 0);
                       cvs_output ("\", 1);
                       }
               cvs_output (p->file, 0);
               cvs_output ("\n", 1);
            } else
         {
               send_to_server ("Directory", 0);
               send_to_server (p->dir[0] == '\0' ? "" : p->dir, 0);
            }
      }
send_to_server("\012", 1);
send_to_server(p->repos, 0);
send_to_server("\012", 1);

send_to_server("Questionable ", 0);
send_to_server(p->file, 0);
send_to_server("\012", 1);
}
free (p->dir);
free (p->repos);
free (p->file);
q = p->next;
free (p);
p = q;
}

if (local)
send_arg("-1");
if (force)
    send_arg("-f");
if (sysmodule)arg
    send_arg("-m");
option_with_arg ("-s", saved_tag);

/* Sending only the names of the files which were modified, added, or removed means that the server will only do an up-to-date check on those files. This is different from local CVS and previous versions of client/server CVS, but it probably is a Good Thing, or at least Not Such A Bad Thing. */
sendfiles (find_args.argv, find_args.argv, local, 0);

send_to_server("\012", 0);
err = get_responses_and_close (1);
if (err != 0 && use_editor && saved_message != NULL)
{
    /* If there was an error, don't make the user's carefully constructed prose. This is something of a kludge; a better solution is probably more along the lines of #150 in TODO
     * doing a second up-to-date check before accepting the log message has also been suggested, but that seems kind of iffy because the real up-to-date check could still fail, another error could occur, &c. Also, a second check would slow things down. */

    char *fname;
    FILE *fp;
    fname = cvs_tempname (1);
    fp = CVS_FOPEN (fname, "w+"),
    if (fp == NULL)
    {
        error (1, 0, "cannot create temporary file %s", fname);
        if (fwrite (saved_message, 1, strlen (saved_message), fp)
            != strlen (saved_message))
            error (1, errno, "cannot write temporary file %s", fname);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close temporary file %s", fname);
        error (0, 0, "saving log message in %s", fname);
    }

    return err;
}

sendif

if (saved_tag != NULL)
tag_check_valid (saved_tag, argc, argv, local, aflag, "msg");

/* XXX - this is not the perfect check for this */
if (argc <= 0)
    write_dirtag = saved_tag;

wrap_setup ();
lock_tree_for_write (argc, argv, aflag);

/*
 * Set up the master update list and hard link list
 */
mulist = getlist (1);
ifdef PRESERVE_PERMissions_SUPPORT
if (preserve_perms)
{
    hardlist = getlist ();
    /*
    We need to save the working directory so that
    * check/lepmc proc can construct a full pathname for each file.
    */
    workingdir = xgetwd ();
}
endif

/*
Run the recursion processor to verify the files are all up-to-date
*/
eff = start_recursion (check_fileproc, check_filesdoneproc, 
check_directoryproc, (DIRLEAVEPROC) NULL, NULL, argv, 
argv, local, WLOCAL, aflag, 0, (char *) NULL, 1);
if (err)
{
    LockCleanup ();
    error (I, 0, "correct above errors first!");
}

/*
Run the recursion processor to commit the files
*/
write_dirnonbranch = 0;
if (noexec == 0)
    err = start_recursion (commit_fileproc, commit_filesdoneproc, 
commit_directoryproc, commit_dirleaveproc, NULL, 
argv, argv, local, WLOCAL, aflag, 0, 
(char *) NULL, 1);

/*
Unlock all the dirs and clean up
*/
LockCleanup ();
dellist (&mulist);

if (last_register_time)
{
    time_t now;
    (void) time (now);
    if (now == last_register_time)
    {
        sleep (1);  /* to avoid time-stamp races */
    }
}

return (err);

/* This routine determines the status of a given file and retrieves
the version information that is associated with that file. */

static
Ctype
classify_file_internal (finfo, vers);
struct fileinfo *finfo;

.{
int save_noexec, save_quiet, save_really_quiet;
Ctype status;

/* FIXME: Do we need to save quiet as well as really_quiet? Last
 time I glanced at ClassifyFile I only saw it looking at really_quiet
 not quiet. */
save_noexec = noexec;
save_quiet = quiet;
save_really_quiet = really_quiet;
noexec = quiet = really_quiet = 1;

/* handle specified numeric revision specially */
if (saved_tag & & isdigit (+saved_tag))
{
    /* If the tag is for the trunk, make sure we're at the head */
if (numdots (saved_tag) < 2)
{
    status = ClassifyFile (finfo, (char *) NULL, (char *) NULL, 
(char *) NULL, 1, aflag, vers, 0);
    if (status == T_MODIFIED || status == T_ADDED)
    {
        CType xstatus;
        freevers (*vers);
    }
}
xstatus = Classify_file (info, saved_tag, (char *) NULL, 
(char *) NULL, 1, aflag, vers, 0);
if (xstatus == T_REMOVE_ENTRY)
    status = T_MODIFIED;
else if (status == T_MODIFIED & & xstatus == T_CONFLICT)
    status = T_MODIFIED;
else
    status = xstatus;
}
else
{
    char *xtag, *cp;
    /*
     * The revision is off the main trunk; make sure we're
     * up-to-date with the head of the specified branch.
     */
    xtag = xstrdup (saved_tag);
    if (strlen (xtag) > 1)
    { /* 
        cp = strncr (xtag, '.');
        */
        cp = strchr (xtag, '.');
        xcp = '\0';
    }
    status = Classify_file (info, xtag, (char *) NULL, 
(char *) NULL, 1, aflag, vers, 0);
    if (!status ||
        (status == T_REMOVE_ENTRY || status == T_CONFLICT)
        && (cp = strchr (xtag, '.') != NULL)
    { /* pluck one more dot off the revision */
        cp = '\0';
    }
    freesn, ts (vers);
    status = Classify_file (info, xtag, (char *) NULL, 
(char *) NULL, 1, aflag, vers, 0);
    status = T_MODIFIED;
} /* now, muck with vers to make the tag correct */
free ((*vers)--tag);
(*vers)->tag = xstrdup (saved_tag);
free (xtag);
else
    status = Classify_file (info, saved_tag, (char *) NULL, (char *) NULL, 
1, 0, vers, 0);
noexec = save_noexec;
quiet = save_quiet;
really_quiet = save_really_quiet;

return status;

} /* Check to see if a file is ok to commit and make sure all files are
    * up-to-date */
/* ARGUSED */
static int
check_fileproc (callerdat, info)
    void *callerdat;
void *

struct file_info *info,
{ 
    CType status;
    char *xdir;
    Node *p;
    List *ulist, *cilist;
    VersTS *vers;
    struct commit_info *ci;
    struct logfile_info *li;

    status = classify_file_internal (info, &vers);
    /*
     * If the force-commit option is enabled, and the file in question
     * appears to be up-to-date, just make it look modified so that
     * it will be committed.
     */
    if (force && status == T_UP_TO_DATE)
        status = T_MODIFIED;

switch (status)
    {
    case T_CHECKOUT:
        ifdef SERVER_SUPPORT
            case T_PATCH:
            endif
            case T_NEEDS_MERGE:
            case T_CONFLICT:
            case T_REMOVE_ENTRY:
            }
error (0, 0, "up-to-date check failed for \"%s\", info->fullname);
  freevers_s (&vers);
  return (1);
}
case T焕发MODIFIED:
case T焕发ADDED:
case T焕发REMOVED:
  /*
   * some quick sanity checks; if no numeric -r option specified:
   * - can't have a sticky date
   * - can't have a sticky tag that is not a branch
   * Also,
   */
  if (status ==焕发REMOVED || vers->tag &&
      !RCS焕然branch (info->rcs, vers->tag)) {
    error (0, 0,
           "sticky tag \"%s\" for file \"%s\" is not a branch",
           vers->tag, info->fullname);
    freevers_s (&vers);
    return (1);
  }
}
if (status ==焕发MODIFIED || force && vers->ts_conflict) {
  char +filestamp;
  int retcode;
  /* We found a "conflict" marker.
     * If the timestamp on the file is the same as the
     * timestamp stored in the Entries file, we block the commit.
     */
  ifdef SERVER_SUPPORT
    if (server_active == vers->ts_conflict[0] == '\n') {
      filestamp = time_stamp (info->file);
      retcode = strcmp (vers->ts_conflict, filestamp);
      free (filestamp);
    } else {
      /* We found a "conflict" marker.
         * If the timestamp on the file is the same as the
         * timestamp stored in the Entries file, we block the commit.
         */
  endif
  if (retcode == 0)
    error (0, 0,
           "file \"%s\" had a conflict and has not been modified",
           info->fullname);
    freevers_s (&vers);
    return (1);
  }
}
if (info->has_markers (info)) {
  /* Make this a warning, not an error, because we have
     no way of knowing whether the "conflict indicators"
     are really from a conflict or whether they are part
     of the document itself (cvstexinfo and sanity.sh in
     \CVS\ itself, for example, tend to want to have strings
     like \">>>\" at the start of a line). Making people
     kludge this the way they need to kludge keyword
     expansion seems undesirable. And it is worse than
     keyword expansion, because there is no -ko
     analogue. */
  warning: file \"%s\" seems to still contain conflict indicators",
           info->fullname);
}
}
if (status == T_REMOVED && vers->tag && isdigit *(vers->tag))
{
    /* Remove also tries to forbid this, but we should check
     * here. I'm only worried about somewhat obscure cases
     * (hacking the Entries file, using an old version of
     * CVS for the remove and a new one for the commit), but
     * there might be other cases. */
    error (0, 0,
        "cannot remove file \"%s\" which has a numeric sticky tag of \"%s\",
        finfo->fullname, vers->tag);

    freevers (&vers);
    return (1);
}
if (status == T_ADDED)
{
    if (vers->tag == NULL)
    {
        char *rcs;
        rcs = xmalloc (strlen (finfo->repository)
            + strlen (finfo->file)
            + sizeof RCSEXTR
            + 5);

        /* Don't look in the attic; if it exists there we
         * will move it back out in checkaddfile. */
        sprintf(rcs, \"%s/\%s\", finfo->repository, finfo->file,
            RCSEXTR);

        if (isreadable (rcs))
        {
            error (0, 0,
                "cannot add file \"%s\" when RCS file \"%s\" already exists",
                finfo->fullname, rcs);
            freevers (&vers);
            free (rcs);
            return (1);
        }
    }

    if (vers->tag && isdigit *(vers->tag) &&

        numdots (vers->tag) > 1)
    {
        error (0, 0,
            "cannot add file \"%s\" with revision \"%s\"; must be on trunk",
            finfo->fullname, vers->tag);

        freevers (&vers);
        return (1);
    }
}

/* done with consistency checks; now, to get on with the commit */
if (finfo->update_al[0] == \"D\")
    xdir = \"\",
else
    xdir = finfo->update_al[0];
if (((p = findnode (uclist, xdir)) != NULL)
    { ulist = ((struct master_lists *) p->data)->ulist;
        cilist = ((struct master_lists *) p->data)->cilist;
    }
else
    {
        struct master_lists *ml;

        ulist = getlist ();
        cilist = getlist ();
        p = getnode ();
        p->key = xstrdup (xdir);
        p->type = UPDATE;
        ml = (struct master_lists *)
            xmalloc (sizeof (struct master_lists));

        ml->ulist = ulist;
        ml->cilist = cilist;
        p->data = (char *) ml;
        p->delproc = masterlist_delproc;
        (void) addnode (mlist, p);
    }
/* first do ulist, then cilist */
p = getnode ();
p->key = xstrdup (finfo->file);
p->type = UPDATE;
p->delproc = update_delproc;
li = ((struct logfile_info *)
    xmalloc (sizeof (struct logfile_info)));
li->type = status;
li->tag = xstrdup (vers->tag);
li->revold = xstrdup (vers->revision);
li->revnew = NULL,
p->data = (char *) li;
(void) addnode (ulist, p);

p = getnode ();
p->key = xstrdup (info->file);
p->type = UPDATE;
p->delproc = ci->delproc;
ci = (struct commitinfo *) xmalloc (sizeof (struct commitinfo));

if (vers->tag)
  ci->rev = xstrdup (vers->tag);
else
  ci->rev = RCS whatbranch (info->rcs, vers->tag);

if (isdigit (vers->tag))
  ci->status = status;
else
  ci->status = ci;
(void) addnode (cilist, p);

1010  ifdef PRESERVE PERMISSIONS SUPPORT
{  
   /* Add this file to hardlist, indexed on its inode. When we're doing a remove operation? */
   char *fullpath;
   Node *linkp;
   struct hardlinkinfo *hlinfo;

   /* Get the full pathname of the current file. */
   fullpath = xmalloc (strlen(working_dir) + strlen(info->fullname) + 2);
   sprintf (fullpath, "%s/%s", working_dir, info->fullname);

   /* To permit following links in subdirectories, files are keyed on finfo->fullname, not on info->name. */
   hlinfo = (struct hardlinkinfo *) xmalloc (sizeof (struct hardlinkinfo));
   hlinfo->status = status;

   linkp->data = (char *) hlinfo;
}

#endif

break;

case T_UNKNOWN:
  error (0, 0, "unknown status %d", status);
  freevers (&vers);
  return (1);

1050  case T_UPTODATE:
    break;
    default:
      error (0, 0, "CVS internal error: unknown status %d", status);
    break;

freevers (&vers);
return (0);

1060
/* By default, return the code that tells do_recursion to examine all directories */

/* ARGSUSED */
static Dtype
check direntproc (callerdat, dir, repos, update_dir, entries)
  void *callerdat;
  char *dir;
char *repos;
char *update_dir;
List *entries;
{
  if (lisdir (dir))
    return (H_SKIP_ALL);

  if (quiet)
    error (0, 0, "Examining %s", update_dir);
1080     return (R_PROCESS);
  }

*/
* Walklist proc to run pre-commit checks
*/
static int
precommit_list_proc (p, closure)
    Node *p;
    void *closure;
1090 {
    struct logfileinfo *li;
    li = (struct logfileinfo *) p->data;
    if (li->type == T_ADDED
         || li->type == T_MODIFIED
         || li->type == T_REMOVED)
        run_arg (p->key);
    return (0);
}

/* Callback proc for pre-commit checking */
static int
precommit_proc (repository, filter)
    char *repository;
    char *filter;
1110 {
    /* see if the filter is there, only if it's a full path */
    if (is_absolute (filter))
        {
            char *s, *cp;
            s = xstrdup (filter);
            for (cp = s; *cp; cp++)
                if (isspace (*cp))
                    {
                        *cp = '\0';
                        break;
                    }
    if (!isfile (s))
        {
            error (0, errno, "cannot find pre-commit filter "%s", s);
            free (s);
            return (1); /* so it fails! */
        }
    free (s);
    run_setup (filter);
    run_arg (repository);
    (void) walklist (saved_ulist, precommit_list_proc, NULL);
    return (run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL, RUN_REALLY));
  }

/* Run the pre-commit checks for the dir */
1140 /* ARGSUSED */
static int
check_precommitproc (callerdat, err, repos, update_dir, entries)
    void *callerdat;
    int err;
    char *repos;
    char *update_dir,
    List *entries;
1150 {
    int n;
    Node *p;

    /* find the update list for this dir */
    p = findnode (mulist, update_dir);
    if (p != NULL)
        saved_ulist = ((struct masterlists *) p->data)->ulist;
    else
        saved_ulist = (List *) NULL;

    /* skip the checks if there's nothing to do */
    if (saved_ulist == NULL || saved_ulist->list->next == saved_ulist->list)
        return (err);

    /* run any pre-commit checks */
    if ((n = ParseInfo (CVSROOTADM.COMMITINFO, repos, precommit_proc, 1)) > 0)
        {
            error (0, 0, "Pre-commit check failed");
            err += n;
        }
Do the work of committing a file

/* ARGSUSED */

static int maxrev;
static char *sbranch;

commitleproc (callerdat, finfo)

{
  Node *p;
  int err = 0;
  List *ulist, *cilist;
  struct commit_info *ci;

  /* Keep track of whether write_dirtag is a branch tag.
   * Note that if it is a branch tag in some files and a nonbranch tag
   * in others, treat it as a nonbranch tag. It is possible that case
   * should elicit a warning or an error. */
  if (write_dirtag != NULL && finfo->rcs != NULL)
    {
      char rev = RCS_getversion (finfo->rcs, write_dirtag, NULL, 1, NULL);
      if (rev != NULL && RCS_nodeisbranch (finfo->rcs, write_dirtag))
        write_dirtagnonnull = 1;
      if (rev != NULL)
        free (rev);
    }

  if (finfo->update_dir[0] == '.')
    p = findnode (mulist, '*.*');
  else
    p = findnode (mulist, finfo->update_dir);

  /* if p is null, there were file type command line args which were
   * all up-to-date so nothing really needs to be done */
  if (p == NULL)
    return (0);
  ulist = ((struct master_lists *) p->data)->ulist;
  cilist = ((struct master_lists *) p->data)->cilist;

  /* At this point, we should have the commit message unless we were called
   * with files as args from the command line. In that latter case, we
   * need to get the commit message ourselves */
  if (flag_message)
    {
      got_message = 1;
      if (use_editor)
        do_editor (finfo->update_dir, &saved_message,
                   finfo->repository, ulist);
      do_verify (saved_message, finfo->repository);
    }

  p = findnode (cilist, finfo->file);
  if (p == NULL)
    return (0);
  ci = (struct commit_info *) p->data;
  if (ci->status == T_MODIFIED)
    {
      if (finfo->rcs != NULL)
        error (1, 0, "internal error: no parsed RCS file*");
      if (lock RCS (finfo->file, finfo->rcs, ci->rev, finfo->repository) != 0)
        {
          unlockrcs (finfo->rcs);
          err = 1;
          goto out;
        }
      else if (ci->status == T_ADDED)
        {
          if (checkaddfile (finfo->file, finfo->repository, ci->tag, ci->options, &finfo->rcs) != 0)
            {
              fixaddfile (finfo->file, finfo->repository);
              err = 1;
              goto out;
            }
        }
    }
1260
/* adding files with a tag, now means adding them on a branch. Since the branch test was done in checkfileproc for modified files, we need to stub it in again here. */
if (ci->tag)
{
    if (finfo->racs == NULL)
    {
        error (1, 0, "internal error: no parsed RCS file");
        ci->rrev = RCSwhatbranch (finfo->racs, ci->tag);
        err = Checkin ("h", finfo, finfo->racs->path, ci->rrev,
                        ci->tag, ci->options, saved_message);
        if (err != 0)
        {
            unlockrcs (finfo->racs);
            fixbranch (finfo->racs, sbranch);
        }
        (void) time (klast_register_time);
        ci->status = T_UPTODATE;
    }
}

/* Add the file for real */
if (ci->status == T_ADDED)
{
    char *xrev = (char *) NULL;
    if (ci->rrev == NULL)
    {
        /* find the max major rev number in this directory */
        maxrev = 0;
        (void) walklist (finfo->entries, findmaxrev, NULL);
        if (maxrev == 0)
            maxrev = 1;
        xrev = xmalloc (20);
        (void) sprintf (xrev, "%d", maxrev);
    }
    *XXX - an added file with symbolic -r should add tag as well */
    err = finaladd (finfo, ci->rrev ? ci->rrev : xrev, ci->tag, ci->options);
    if (xrev)
        free (xrev);
    else if (ci->status == T_MODIFIED)
    {
        err = Checkin ("m", finfo,
                        finfo->racs->path, ci->rrev, ci->tag,
                        ci->options, saved_message);
        (void) time (klast_register_time);
        if (err != 0)
        {
            unlockrcs (finfo->racs);
            fixbranch (finfo->racs, sbranch);
        }
    }
    else if (ci->status == T_REMOVED)
    {
        err = removefile (finfo, ci->tag, saved_message);

        ifdef SERVER_SUPPORT
        if (server_active) {
            servergetentryonly ();
            serverupdated (finfo, NULL,
                           (mode_t) -1,
                           (unsigned char *) NULL,
                           (struct buffer *) NULL);
        }
        endif
    }
1340  /* Clearly this is right for T_MODIFIED. I haven’t thought so much about T_ADDED or T_REMOVED. */
    notify_do ("c", finfo->racs, getcaller (), NULL, NULL, NULL, finfo->repository);
    out:
    if (err != 0)
    { /* on failure, remove the file from ulist */
        p = findnode (ulist, finfo->racs);
        if (p != NULL)
        { /* remove from ulist */
if (p)
   delnode (p);
else
   
   /* On success, retrieve the new version number of the file and
   copy it into the log information (see logmsg.c
   (logfile写出WRITE) for more details). We should only update
   the version number for files that have been added or
   modified but not removed. Why? classify_file internal
   will return the version number of a file even after it has
   been removed from the archive, which is not the behavior we
   want for our commitlog messages; we want the old version
   number and then "NONE." */
if (ci->status != T_REMOVED)
   {p = findnode (ulist, finfo->file);
   if (p)
      {
      Vers_TS *vers;
      struct logfile_info *li;
      (void) classify_file_internal (finfo, &vers);
      li = (struct logfile_info *) p->data;
      li->rev_new = xstrdup (vers->vn_rev);
      freevers (vers);
      }
   }
   return (err);
}

/ * Log the commit and clean up the update list
/ *
/ * ARGSUSED */
static int
commitfiledoneproc (callerdat, err, repository, update=dir, entries)
void *callerdat;
int err;
char *repository;
char *update=dir;
List *entries;
{Node *p;
 List *ulist;
   p = findnode (mulist, update=dir);
   if (p == NULL)
      return (err);
   ulist = ((struct master_lists *) p->data)->ulist;
   got_message = 0;
   Update_logfile (repository, saved_message, (FILE *) 0, ulist);
   /* Build the administrative files if necessary. */
   char *p;
if (strcmp ((CVSroot=directory, repository, 
stenlen (CVSroot=directory)) != 0)
   error (0, 0, 
"internal error: repository (%s) doesn't begin with root (%s)!",
repository, CVSroot=directory);
p = repository + stlen (CVSroot=directory);
if (p == */
++p;
if (strcmp ("CVSROOT", p) == 0
   /* Check for subdirectories because people may want to create
   subdirectories and list files therein in checkoutlist. */
   || strcmp ("CVSROOT", p, stlen ("CVSROOT") == 0)
   }
   /* "Database" might a little bit grandiose and/or vague,
   but "checked-out copies of administrative files, unless
   in the case of modules and you are using ndbm in which
   case modules (p, dir, db)" is verbose and excessively
   focused on how the database is implemented. */
   /* mkmodule requires the absolute name of the CVSROOT directory.
   Remove anything after the 'CVSROOT' component -- this is
   necessary when committing in a subdirectory of CVSROOT. */
   char *administr = xstrdup (repository);
int cvsrootlen = stlen ("CVSROOT")
assert (administr[p = repository + cvsrootlen] == '\0')
1440    | admin_dir[p - repository + cvsvrootlen] == '/'
     admin_dir[p - repository + cvsvrootlen] = '\0';
cvs_output (program_name, 0);
cvs_output ('.*', 1);
cvs_output (command_name, 0);
cvs_output ('*: Rebuilding administrative file database\n', 0);
     mkmodule (admin_dir);
     free (admin_dir);
    }

1450 if (err == 0 && run_module (prog)  
     FILE *fp;
     if ((fp = CVS_FOPEN (CVSADM_CIPROG, '*r')) != NULL)  
     {
     char *line;
     int line_length;
     size_t line_chars_allocated;
1460     char *repos;
     line = NULL;
     line_chars_allocated = 0;
     line_length = getline (&line, &line_chars_allocated, fp);
     if (line_length > 0)  
     {
           /* Remove any trailing newline. */
     if (line[line_length - 1] == '\n')
     line[line_length - 1] = '\0';
1470     repos = Name_Repository ((char *) NULL, update_dir);
     run_setup (line);
     run_ig (repos);
     cvs_output (program_name, 0);
     cvs_output ('.*', 1);
     cvs_output (command_name, 0);
     cvs_output ('*: Executing "', 0);
     run_print (stdout);
     cvs_output ('"\n', 0);
     (void) run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL);
1480     free (repos);
     }
     else
     {
     if (ferror (fp))
     error (0, errno, "warning: error reading %s",
                    CVSADM_CIPROG);
     }
     if (line != NULL)
     free (line);
1490 if (fclose (fp) < 0)
     error (0, errno, "warning: cannot close %s", CVSADM_CIPROG);
     else
     {
     if (! existence error (errno))
     error (0, errno, "warning: cannot open %s", CVSADM_CIPROG);
     }
    }
1500 return (err);
    }

    /* Get the log message for a dir */
    /* ARGSUSED */
    static Dtype
    commitcrentproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
1510 char *dir;
     char *repos;
     char *update_dir;
     List *entries;
     {
     Node *p;
     List *ulist;
     char *realrepos;
     if (!isdir (dir))
    return (H_SKIP_ALL);
    /* find the update list for this dir */
     p = findnode (mulist, update_dir);
     if (p != NULL)
     ulist = ((struct master_lists *) p->data)->ulist;
     else
     ulist = (List *) NULL;
     return (ulist)
     }
static int findmaxrev (p, closure)
    Node *p;
    void *closure;
{
    char *cp;
    int thisrev;
    Entnode *entdata;
    
    entdata = (Entnode *) p->data;
    if (entdata->type != ENT_FILE)
        return (0);
    cp = strchr (entdata->version, '.');
    if (cp != NULL)
        *cp = '\0';
    thisrev = atoi (entdata->version);
    if (cp != NULL)
        *cp = '.';
    if (thisrev > maxrev)
        maxrev = thisrev;
    return (0);
}

static int remove_file (finfo, tag, message)
{ struct file_info *finfo;
    char stag;
    char *message;
    
    mode_t omask;
    int retcode;
    char *tmp,
    int branch;
    int lockflag;
    char *corev;
    char *sprev;
    char *oldpath;
    
    corev = NULL;
    rev = NULL;
    
    /* skip the files as an optimization */
    if (ulist == NULL | ulist->next == ulist)
        return (R_SKIP_FILES);
    /* get commit message */
    realrepos = NameRepository (dir, update, dir);
    got_message = 1;
    if (useeditor) 
doeditor (update, dir, &savedmessage, realrepos, ulist);
doverify (savedmessage, realrepos);
free (realrepos);
    return (R_PROCESS);

    / *
    * Process the post-commit proc if necessary
    */
    /* ARGSUSED */
    static int
    commitdirleaveproc (callerdat, dir, err, update, dir, entries)
    void *callerdat;
    char *dir;
    int err;
    char *update, dir;
    List *entries;
    {
    /* update the per-directory tag info */
    /* FIXME? Why? The “commit examples” node of cvs.texinfo briefly
     mentions commit - r being sticky, but apparently in the context of
     this being a confusing feature! */
    if (err == 0 & & write_dirtag != NULL)
        
        { WriteTag (NULL, write_dirtag, NULL, write_dirtagnonbranch,
            update, dir, NameRepository (dir, update, dir));
        }
    return (err);
    
    /* find the maximum major rev number in an entries file */
    static int
    findmaxrev (p, closure)
    Node *p;
    void *closure;
    
    char *cp;
    int thisrev;
    Entnode *entdata;
    
    entdata = (Entnode *) p->data;
    if (entdata->type != ENT_FILE)
        return (0);
    cp = strchr (entdata->version, '.');
    if (cp != NULL)
        *cp = '\0';
    thisrev = atoi (entdata->version);
    if (cp != NULL)
        *cp = '.';
    if (thisrev > maxrev)
        maxrev = thisrev;
    return (0);
}

    /* Actually remove a file by moving it to the attic
     XXX - if removing a .v file that is a relative symbolic link to
     another .v file, we probably should add a ".." component to the
     link to keep it relative after we move it into the attic. */
    static int
    remove_file (finfo, tag, message)
    struct file_info *finfo;
    char stag;
    char *message;
    
    mode_t omask;
    int retcode;
    char *tmp,
    int branch;
    int lockflag;
    char *corev;
    char *sprev;
    char *oldpath;
    
    corev = NULL;
    rev = NULL;
    
    /* skip the files as an optimization */
    if (ulist == NULL | ulist->next == ulist)
        return (R_SKIP_FILES);
    /* get commit message */
    realrepos = NameRepository (dir, update, dir);
    got_message = 1;
    if (useeditor) 
doeditor (update, dir, &savedmessage, realrepos, ulist);
doverify (savedmessage, realrepos);
free (realrepos);
    return (R_PROCESS);

    / *
    * Process the post-commit proc if necessary
    */
    /* ARGSUSED */
    static int
    commitdirleaveproc (callerdat, dir, err, update, dir, entries)
    void *callerdat;
    char *dir;
    int err;
    char *update, dir;
    List *entries;
    {
    /* update the per-directory tag info */
    /* FIXME? Why? The “commit examples” node of cvs.texinfo briefly
     mentions commit - r being sticky, but apparently in the context of
     this being a confusing feature! */
    if (err == 0 & & write_dirtag != NULL)
        
        { WriteTag (NULL, write_dirtag, NULL, write_dirtagnonbranch,
            update, dir, NameRepository (dir, update, dir));
        }
    return (err);
    
    /* find the maximum major rev number in an entries file */
    static int
    findmaxrev (p, closure)
    Node *p;
    void *closure;
    
    char *cp;
    int thisrev;
    Entnode *entdata;
    
    entdata = (Entnode *) p->data;
    if (entdata->type != ENT_FILE)
        return (0);
    cp = strchr (entdata->version, '.');
    if (cp != NULL)
        *cp = '\0';
    thisrev = atoi (entdata->version);
    if (cp != NULL)
        *cp = '.';
    if (thisrev > maxrev)
        maxrev = thisrev;
    return (0);
}
prev->rev = NULL;
retcode = 0;
if (info->rcs == NULL)
    error (1, 0, "internal error: no parsed RCS file");
branch = 0;
if (tag && ((branch = RCS_isbranch (info->rcs, tag)) ))
{
    /* a symbolic tag is specified; just remove the tag from the file */
    if (retcode = RCS_deltag (info->rcs, tag)) != 0)
    {
        if (quiet)
            error (0, retcode == -1 ? errno : 0,
                "Failed to remove tag \"%s\" from \"%s\", tag, info->fullname");
        return (1);
    }
    RCS_rewrite (info->rcs, NULL, NULL);
    Scratch_Entry (info->entries, info->file);
    return (0);
}
/* we are removing the file from either the head or a branch */
/* commit a new, dead revision. */
/* Print message indicating that file is going to be removed. */
cvs_output ("Removing \"%s\".", tag);
cvs_output (info->fullname, 0);
cvs_output ("\"%s\".", 0);

rev = NULL;
lockflag = 1;
if (branch)
{
    char *branchname;
    rev = RCS_whatbranch (info->rcs, tag);
    if (rev == NULL)
    {
        error (0, 0, "Cannot find branch \"%s\".", tag);
        return (1);
    }
    branchname = RCS_getbranch (info->rcs, rev, 1);
    if (branchname == NULL)
    {
        /* no revision exists on this branch. use the previous revision but do not lock. */
        corev = RCS_gettag (info->rcs, tag, 1, (int *) NULL);
        prev->rev = xstrdup(rev);
        lockflag = 0;
    }
    else
    {
        corev = xstrdup (rev);
        prev->rev = xstrdup(branchname);
        free (branchname);
    }
}

} else /* Not a branch */
{
    /* Get current head revision of file. */
    prev->rev = RCS_head (info->rcs);
}
/* if removing without a tag or a branch, then make sure the default branch is the trunk. */
if (tag && ! branch)
{
    if (RCS_getbranch (info->rcs, NULL) != 0)
    {
        error (0, 0, "Cannot change branch to default for \"%s\", info->fullname");
        return (1);
    }
    RCS_rewrite (info->rcs, NULL, NULL);
}
#endif

#define SERVER_SUPPORT
if (server_active) {
    /* If this is the server, there will be a file sitting in the
       temp directory which is the kludgy way in which server.c
       tells time_stamp that the file is no longer around. Remove
       it so we can create temp files with that name (ignore errors). */
    unlink_file (info->file);
}
#endif

/* check something out. Generally this is the head. If we have a
particular rev, then name it. */
retcode = RCS_checkout (info->rcs, info->file, rev ? corev : NULL,
                     (char *) NULL, (char *) NULL, RUN_TTY,
                     (RCS_CHECKOUTPROC) NULL, (void *) NULL);
if (retcode != 0)
{
    error (0, 0, "failed to check out '%s'", info->fullname);
    return (1);
}

/* Except when we are creating a branch, lock the revision so that
we can check in the new revision. */
if (lockflag)
{
    if (RCS_lock (info->rcs, rev ? corev : NULL, 1) == -1)
        RCS_write (info->rcs, NULL, NULL);
}
if (corev != NULL)
    free (corev);

retcode = RCS_checkin (info->rcs, info->file, message, rev,
                      RCS_FLAGS_DEAD | RCS_FLAGS_QUIET);
if (retcode != 0)
{
    if (quiet)
        error (0, retcode == -1 ? errno : 0,
              "failed to commit dead revision for '%s'", info->fullname);
    return (1);
}

if (rev != NULL)
    free (rev);
old_path = info->rcs->path;
if (!branch)
{
    /* this was the head; really move it into the Attic */
tmp = xmalloc(strlen(info->repository) +
              sizeof('/') +
              sizeof(CVSATTIC) +
              sizeof('/') +
              strlen(info->file) +
              sizeof(RCSEXTR) + 1);
(void) sprintf (tmp, "%s/%s/%s%s",
              info->repository, CVSATTIC,
              info->file, RCSEXTR);
(void) sprintf (tmp, "%s/%s/%s%s",
              info->repository, CVSATTIC,
              info->file, RCSEXTR);
    free (tmp);
    return (1);
}

/* The old value of info->rcs->path is in old_path, and is
freed below. */
info->rcs->path = tmp;

/* Print message that file was removed. */
cvs_output (old_path, 0);
cvs_output ("cvs: ", 0);
cvs_output (info->file, 0);
cvs_output ("rename previous revision: "; 0);
cvs_output ("name: "; 0);
cvs_output ("value: "; 0);
free (prevrev);
if (old_path != info->rcs->path)
    free (old_path);
Scratch_Entry (info->entries, info->file);
    return (0);
}

/* Do the actual checkin for added files */
static int
finaladd (info, rev, tag, options)
struct file_info *info;
char *rev;
char *tag;
char *options;
{
int ret;
char *rcs;

rcs = locate_rcs (finfo->file, info->repository);
ret = Checkin ("!", finfo, rcs, rev, tag, options, saved_message);
if (ret == 0) {
    char *tmp = xmalloc (strlen (finfo->file) + sizeof (CVSADM) + sizeof (CVSEXT_LOG) + 10);
    (void) sprintf (tmp, "%a/fайл", CVSADM, finfo->file, CVSEXT_LOG);
    (void) unlink_file (tmp);
    free (tmp);
} else
    fixaddfile (finfo->file, info->repository);
(void) time (klast_register_time);
free (rcs);
return (ret);
}

/*
 * Unlock an rcs file
 */
static void unlockrcs (rcs)
    RCSNode *rcs;
RCSNode *rcs;
{ int retcode;

if ((retcode = RCS_unlock (rcs, NULL, 0)) != 0)
    error (retcode == -1 ? 1 : 0, retcode == -1 ? errno : 0,
        "could not unlock %s", rcs->path);
else
    RCS_rewrite (rcs, NULL, NULL);
}
/*
 * remove a partially added file. if we can parse it, leave it alone.
 */

static void
fixaddfile (file, repository)
char *file;
char *repository;
{ RCSNode *rcsfile;
char *rcs;
int save_really_quiet;
rcs = locate_rcs (file, repository);
save_really_quiet = really_quiet;
really_quiet = 1;
if ((rcfile = RCS_parse_rcsfile (rcs)) == NULL)
    (void) unlink_file (rcs);
else
    freercsnode (rcfile);
really_quiet = save_really_quiet;
free (rcs);
}

static void
fixbranch (rcs, branch)
RCSNode *rcs;
char *branch;
{ int retcode;

if (branch != NULL)
    if ((retcode = RCS_setbranch (rcs, branch)) != 0)
        error (retcode == -1 ? 1 : 0, retcode == -1 ? errno : 0,
            "cannot restore branch to %s for %s", branch, rcs->path);
    RCS_rewrite (rcs, NULL, NULL);
}

/*
 * do the initial part of a file add for the named file. if adding
 * with a tag, put the file in the Attic and point the symbolic tag
 * at the committed revision.
 */

static int
checkaddfile (file, repository, tag, options, rcsnode)
char *file;
char *repository;
char *tag;
char *options;
RCSNode **rcsnode;
{
char *rcs;
char *fname;
mode_t omask;
int retcode = 0;
int newfile = 0;
RCSNode *rcsnode = NULL;
int retval;
if (tag){
rcs = xmalloc (strlen (repository) + strlen (file) + sizeof (RCSEXT) + sizeof (CVSATTIC) + 10);
(void) sprintf (rcs, "%s/%s", repository, file, RCSEXT);
if (!isreadable (rcs))
{
(void) sprintf(rcs, "%s/%s", repository, CVSATTIC);
omask = umask (cvsumask);
if (CVS_MKDIR (rcs, 0777) != 0 && errno!=EEXIST)
error (1, errno, "cannot make directory '"%s'" , rcs);
(void) umask (omask);
(void) sprintf [rcs, "%s/%s/%s", repository, CVSATTIC, file, RCSEXT];
}
else
rcs = locate (rcs, file, repository);
if (isreadable (rcs))
{/∗ file has existed in the past. Prepare to resurrect. ∗/
char *rev;
if ((rcsfile = **rcsnode) == NULL)
{
error (0, 0, "could not find parsed rcsfile '"%s'", file);
retval = 1;
goto out;
}
if (tag == NULL)
{
char *oldfile;
/* we are adding on the trunk, so move the file out of the Attic. ∗/
oldfile = xstrdup (rcs);
sprintf (rcs, "%s/%s", repository, file, RCSEXT);
}
if (strcmp (oldfile, rcs) == 0)
{
error (0, 0, "internal error: confused about attic for '"%s'", oldfile);
out1:
free (oldfile);
retval = 1;
goto out;
}
if (CVS_RENAME (oldfile, rcs) != 0)
{
error (0, errno, "failed to move '"%s' out of the attic", oldfile);
goto out1;
}
if (isreadable (oldfile) || isreadable (rcs))
{
error (0, 0, "\ninternal error: '"%s' didn't move out of the attic", oldfile);
goto out1;
}
free (oldfile);
free (rcsfile->path);
rcsfile->path = xstrdup (rcs);
}
rev = RCS_getversion (rcsfile, tag, NULL, 1, (int *) NULL);
/* and lock it */
if (lock_RCS (file, rcsfile, rev, repository))
{
error (0, 0, "cannot lock '"%s'", rcs);
if (rev != NULL)
free (rev);
retval = 1;
goto out;
}
if (rev != NULL)
    free (rev);
} else {
    /* this is the first time we have ever seen this file; create 
     * an RCS file. */
    char *desc;
    size_t desclen;
    size_t desclen;
    char *opt;
    desc = NULL;
    desclalloc = 0;
    desclen = 0;
    fname = xmalloc (strlen (fname) + sizeof (CVSADM) + sizeof (CVSEXTPREFIX) + 10);
    (void) sprintf (fname, "%s/%s", CVSADM, file, CVSEXTPREFIX);
    /* If the file does not exist, no big deal. In particular, the 
     * server does not (yet at least) create CVSEXTPREFIX files. */
    if (isfile (fname))
        /* FIXME: Should be including update for in the appropriate 
         * place here. */
        get_file (fname, name, "r", &desc, &desclalloc, &desclen);
        free (fname);
    /* From reading the RCS 5.7 source, "rcs -i" adds a newline to the 
     * end of the log message if the message is nonempty. 
     * Do it. RCS also deletes certain whitespace, in cleanlogmsg, 
     * which we don't try to do here. */
    if (desclen > 0)
        {
        expand_string (&desc, &desclalloc, desclen + 1);
        desc[desclen++] = \"\012\";
        }
    /* Set RCS keyword expansion options. */
    if (options && options[0] == -1 && options[1] == 'k')
        opt = options + 2;
    else
        opt = NULL;
    /* This message is an artifact of the time when this 
     * was implemented via "rcs -i". It should be revised at 
     * some point (does the "initial revision" in the message from 
     * RCScheckm indicate that this is a new file? Or does the 
     * "RCS file" message serve some function?). */
    cvs_output ("RCS file: %s/%s\0", NULL);
    cvs_output (rcs, 0);
    cvs_output (\"\0\", 0);
    if (addrcs_file (NULL, rcs, file, NULL, opt,
        NULL, NULL, NULL, NULL, descl, desclen, NULL, NULL) != 0)
        {
        retval = 1;
        goto out;
        }
    rcsfile = RCS_parsersrcfile (rcs);
    newfile = 1;
    if (desc != NULL)
        free (desc);
    }
    /* when adding a file for the first time, and using a tag, we need 
     * to create a dead revision on the trunk. */
    if (!tag && !newfile)
    {
    char *tmp;
    FILE *fp;
    /* move the new file out of the way. */
    fname = xmalloc (strlen (fname) + sizeof (CVSADM) + sizeof (CVSPREFIX) + 10);
    (void) sprintf (fname, "%s/%s", CVSADM, CVSPREFIX, file);
    rename_file (file, fname);
    /* Create empty FILE. Can't use copy_file with a DEVNULL 
     * argument - copy_file now ignores device files. */
    fp = fopen (file, "w");
    if (fp == NULL)
        error (1, errno, "cannot open %s for writing", file);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", file);
    tmp = xmalloc (strlen (fname) + strlen (tag) + 80);
    /* commit a dead revision. */
(void) sprintf (tmp, "file %s was initially added on branch %s. ",
file, tag);
retcode = RCS_checkin (rcsfile, NULL, tmp, NULL,
RCS_FLAGS_DEAD | RCS_FLAGS_QUIET);
free (tmp);
if (retcode != 0)
{
  error (retcode == -1 ? retcode == -1 ? errno : 0,
         "could not create initial dead revision %s", rcs);
  retval = 1;
goto out;
}
/* put the new file back where it was */
rename (fname, file);
free (fname);
/* double-check that the file was written correctly */
freercsnode (&rcsfile);
rcsfile = RCS_parse (file, repository);
if (rcsfile == NULL)
{
  error (0, 0, "could not read %s", rcs);
  retval = 1;
goto out;
}
if (rcsnode != NULL)
{
  assert (rcsnode == NULL);
  *rcsnode = rcsfile;
}
/* and lock it once again. */
if (lock_RCS (file, rcsfile, NULL, repository))
{
  error (0, 0, "cannot lock '%s'.", rcs);
  retval = 1;
goto out;
}
if (tag != NULL)
{
  /* when adding with a tag, we need to stub a branch, if it
doesn't already exist. */
  if (rcsfile == NULL)
  {
    if (rcsnode != NULL && *rcsnode != NULL)
      rcsfile = *rcsnode;
    else
      {
        rcsfile = RCS_parse (file, repository);
        if (rcsfile == NULL)
        {
          error (0, 0, "could not read %s", rcs);
          retval = 1;
goto out;
        }
      }
  }
  if (!RCS_nodeisbranch (rcsfile, tag))
  {
    /* branch does not exist. Stub it. */
    char *head;
    char *magicrev;
    head = RCS_getversion (rcsfile, NULL, NULL, 0, (int *) NULL);
    magicrev = RCS_magicrev (rcsfile, head);
    retcode = RCS_settag (rcsfile, tag, magicrev);
    RCS_rewrite (rcsfile, NULL, NULL);
    free (head);
    free (magicrev);
    if (retcode != 0)
    {
      error (retcode == -1 ? retcode == -1 ? errno : 0,
             "could not stub branch %s for %s", tag, rcs);
      retval = 1;
goto out;
    }
  }
  else
  {
    /* lock the branch. (stubbed branches need not be locked.) */
    if (lock_RCS (file, rcsfile, NULL, repository))
    {
error (0, 0, "cannot lock ‘%s’.", rcs);
  retval = 1;
  goto out;
}
}
if (rcsnodemode && *rcsnodemode != rcsfile)
  {freesrcnnode(rcsnodemode);
   *rcsnodemode = rcsfile;
  }
fileattrnewfile (file);
/* I don't think fixrcsmodes is needed any more. In the
addrcs file case, the algorithms used by addrcsfile and
fixrcsmodes are the same, so there is no need to go through
it all twice. In the other cases, I think we want to just
preserve the mode that the file had before we started. That is
a behavior change, but I would think a desirable one. */
fixrcsmodes (rcs, file);
  retval = 0;
out: free (rcs);
return retval;
}
/* Attempt to place a lock on the RCS file; returns 0 if it could and 1 if it
couldn't. If the RCS file currently has a branch as the head, we must
move the head back to the trunk before locking the file, and be sure to
put the branch back as the head if there are any errors. */
static int
lockRCS (user, rcs, rev, repository)
char *user;
RCSNode *rcs;
char *rev;
char *repository;
{
  char *branch = NULL;
  int err = 0;

  /* For a specified, numeric revision of the form “1” or “1.1”, (or when
   * no revision is specified “”), definitely move the branch to the trunk
   * before locking the RCS file.
   *
  */
  if (rev == NULL || (rev && isdigit (*rev) && numdots (rev) < 2))
    {
      branch = xstrdup (rcs->branch);
      if (branch != NULL)
        {
          if (RCSsetbranch (rcs, NULL) != 0)
            {
              error (0, 0, "cannot change branch to default for ‘%s’.",
                  rcs->path);
              if (branch)
                free (branch);
              return (1);
            }
          err = RCSlock (rcs, NULL, 1);
        }
      else
        { (void) RCS_lock (rcs, rev, 1);
        }
    }
/* We used to call RCSrewrite here, and that might seem
appropriate in order to write out the locked revision
information. However, such a call would actually serve no
purpose. CVS locks will prevent any interference from other
CVS processes. The comment above rcsinternal_lockfile
explains that it is already unsafe to use RCS and CVS
simultaneously. It follows that writing out the locked
revision information here would add no additional security.
If we ever do care about it, the proper fix is to create the
RCS lock file before calling this function, and maintain it
until the checkin is complete.

The call to RCSlock is still required at present, since in
some cases RCS checkin will determine which revision to check in by looking for a lock. FIXME: This is rather roundabout, and a more straightforward approach would probably be easier to understand. */

if (err == 0)
{
    if (sbranch != NULL)
    {
        free (sbranch);
        sbranch = branch;
        return (0);
    }

    /* try to restore the branch if we can on error */
    if (branch != NULL)
    {
        fixbranch (rcs, branch);
    }
    free (branch);
    return (1);
}

/* Called when "adding" files to the RCS repository. It doesn't seem to be possible to get RCS to use the right mode, so we change it after the fact. TODO: now that RCS has been librarified, we have the power to change this. */

static void
fixrcsmodes (rcs, user)
char * rcs;
char * user;

{ struct stat sb;
    mode_t rcs_mode;

    #ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* Do ye nothing to the modes on a symbolic link. */
    if (preserve_perms && islink (user))
        return;
    #endif

    /* Now we compute the new mode.
     * TODO: decide whether this whole thing can/should be skipped when 'preserve_perms' is set. Almost certainly so. -twp
     *
     * The algorithm that we use is:
     *
     * Write permission is always off (this is what RCS and CVS have always done).
     *
     * If $S_IRUSR$ is on (user read), then the read permission of the RCS file will be on. It would seem that if this is off, then other users can't do "cvs update" and such, so perhaps this should be hardcoded to being on (it is a strange case, though the case in which a user file doesn't have user read permission on).
     *
     * If $S_IWUSR$ is on (user execute), then set execute permission on the RCS file. This allows other users who check out the file to get the right setting for whether a shell script (for example) has the executable bit set.
     *
     * The result of that calculation is modified by CVSUMASK. The reason, of course, that the read and execute settings take the user bit and copy it to all three bits (user, group, other), is that it should be CVSUMASK, not the umask of individual users, which is the sole determinant of modes in the repository. */
    rcs_mode = 0;
    if (sb_mode & $S_IRUSR$)
        rcs_mode |= $S_IRUSR | $S_IRGRP | $S_IROTH;
    if (sb_mode & $S_IWUSR$)
        rcs_mode |= $S_IWUSR | $S_IWGRP | $S_IWOTH;
    rcs_mode &= ~cvsumask;

    if (chmod (rcs, rcs_mode) < 0)
        error (0, errno, "warning: cannot change node of %s", rcs);
}

/* free an UPDATE node's data */

void
update_delproc (p)
Node *p;

struct logfile_info *li;
li = (struct logfile_info *) p->data;
if (li->tag)
  free (li->tag);
if (li->rev_old)
  free (li->rev_old);
if (li->rev_new)
  free (li->rev_new);
free (li);

/* Free the commit_info structure in p. */
static void
ci_delproc (p)
  Node *p;

  ci = (struct commit_info *) p->data;
  if (ci->rev)
    free (ci->rev);
  if (ci->tag)
    free (ci->tag);
  if (ci->options)
    free (ci->options);
  free (ci);

/* Free the commit_info structure in p. */
static void
masterlist_delproc (p)
  Node *p;

  ml = (struct master_lists *) p->data;
dellist (&ml->ulist);
dellist (&ml->clist);
free (ml);

/* Find an RCS file in the repository. Most parts of CVS will want to rely
instead on RCS_parse which performs a similar operation and is called by
recurse.c which then puts the result in useful places like the rcs field
of struct file_info.
REPOSITORY is the repository (including the directory) and FILE is
the filename within that directory (without RCSEXT). Returns a
newly-malloc'd array containing the absolute pathname of the RCS
file that was found. */
static char *
locate_rcs (file, repository)
  char *file;
  char *repository;

  rcs = xmalloc (strlen (repository) + strlen (file) + sizeof (RCSEXT) + 10);
  (void) sprintf (rcs, "%s/%s", repository, file, RCSEXT);
  if (!isreadable (rcs))
    {
      (void) sprintf (rcs, "%s/%s", repository, CVSATTIC, file, RCSEXT);
      if (!isreadable (rcs))
        (void) sprintf (rcs, "%s/%s", repository, file, RCSEXT);
    }
  return rcs;
A.11 create_adm.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * 
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * 
 * Create Administration.
 * 
 * Creates a CVS administration directory based on the argument repository; the
 * "Entries" file is prefilled from the "initrecord" argument.
 * */

#include "cvs.h"

#include <cvs.h>

update dir includes dir as its last component.

Return value is 0 for success, or 1 if we printed a warning message.

Note that many errors are still fatal; particularly for unlikely errors
20

a fatal error is probably better than a warning which might be missed
or after which CVS might do something non-useful. If WARN is zero, then
don't print warnings; all errors are fatal then. */

int
Create_Admin (dir, update dir, repository, tag, date, nonbranch, warn)
char *dir;
char *update dir;
char *repository;
char *tag;
char *date;
int nonbranch;
int warn;
{
    FILE *fout;
    char *cp;
    char *reposcopy;
    char *tmp;
    
    #ifdef SERVER_SUPPORT
    40
    if (trace)
    {
        fprintf (stderr, ":> Create_Admin (%s, %s, %s, %s, %s, %d, %d)
";
            (server active) ? 'S' : ' ',
            dir, update dir, repository, tag ? tag : "",
            date ? date : "", nonbranch, warn);
    }
    #endif
    
    if (noexec)
    50
    return 0;
    
    tmp = xmalloc (strlen (dir) + 100);
    if (dir != NULL)
    {
        (void) sprintf (tmp, "%s/%s", dir, CVSADM);
        else
        {
            (void) strcpy (tmp, CVSADM);
            if (isfile (tmp))
                error (1, 0, "there is a version in %s already", update dir);
        }
    60
    else
        error (1, errno, "cannot make directory%s in %s",
            CVSADM, update dir);
    
    if (warn)
    70
    {
        /* The reason that this is a warning, rather than silently
           just skipping creating the directory, is that we don't want
           CVS's behavior to vary subtly based on factors (like directory
           permissions) which are not made clear to the user. With
           the warning at least we let them know what is going on. */
        error (0, errno, "warning: cannot make directory %s in %s",
            CVSPATH, repository, tag); 
        free (tmp);
        return 1;
    }
    else
        error (1, errno, "cannot make directory %s in %s",
            CVSPATH, repository, tag);
    }

    /* record the current cvsroot for later use */
    CreateRoot (dir, CVSSRCDIR);
    if (dir != NULL)
    

(void) sprintf (tmp, "%s/%s", dir, CVSADM_REP);
else
    (void) strcpy (tmp, CVSADM_REP);
fout = CVS_FOPEN (tmp, "r+"),
if (fout == NULL)
    {
        if (update_dir[0] == '\' )
            error (1, errno, "cannot open %s", tmp);
        else
            error (1, errno, "cannot open %s/%s", update_dir, CVSADM_REP);
    }
reposcopy = xstrdup (repository);
Sanitize_Repository_Name (reposcopy);
/* The top level of the repository is a special case – we need to
write it with an extra dot at the end. This trailing "." stuff
ruins me the wrong way – on the other hand, I don't want to
spend the time making sure all of the code can handle it if we
don't do it. */
if (strcmp (reposcopy, CVSroot_directory) == 0)
    {
        reposcopy = xrealloc (reposcopy, strlen (reposcopy) + 3);
        strcat (reposcopy, ".");
    }
    cp = reposcopy;
#endif RELATIVE_REPO
/*
 * If the Repository file is to hold a relative path, try to strip off
 * the leading CVSroot argument.
 */
if (CVSroot_directory != NULL)
    {
        path = xmalloc (strlen (CVSroot_directory) + 10);
        (void) sprintf (path, "%s", CVSroot_directory);
        if (strcmp (cp, path, strlen (path)) == 0)
            cp += strlen (path);
        free (path);
    }
#endif
if (strcmp (fout, "%s/%s", cp) < 0)
    {
        if (update_dir[0] == '\')
            error (1, errno, "write to %s failed", tmp);
        else
            error (1, errno, "write to %s/%s failed", update_dir, CVSADM_REP);
    }
if (fclose (fout) == EOF)
    {
        if (update_dir[0] == '
')
            error (1, errno, "cannot close %s", tmp);
        else
            error (1, errno, "cannot close %s/%s", update_dir, CVSADM_REP);
    }
/* now, do the Entries file */
if (dir != NULL)
    (void) sprintf (tmp, "%s/%s", dir, CVSADM_ENT);
else
    (void) strcpy (tmp, CVSADM_ENT);   fout = CVS_FOPEN (tmp, "w+"),
if (fout == NULL)
    {
        if (update_dir[0] == '\')
            error (1, errno, "cannot open %s", tmp);
        else
            error (1, errno, "cannot open %s/%s", update_dir, CVSADM_ENT);
    }
if (fclose (fout) == EOF)
    {
        if (update_dir[0] == '
')
            error (1, errno, "cannot close %s", tmp);
        else
            error (1, errno, "cannot close %s/%s", update_dir, CVSADM_ENT);
    }
/* Create a new CVS/Tag file */
WriteTag (dir, tag, date, nonbranch, update_dir, repository);
#endif SERVER_SUPPORT
if (server.active)
    {
        servertemplate (update_dir, repository);
    }
if (trace)
{ fprintf(stderr, "%c< Create_Admin\n",  
(server_active) ? '"' : '"');  
}  
#endif  
free (reposcopy);  
free (tmp);  
return 0;  
}
A.12 cvs.h

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS kit.
 */

/* basic information used in all source files */

#include "config.h" /* this is stuff found via autoconf */
#include "options.h" /* these are some larger questions which
                        can't easily be automatically checked for */

/* Changed from if __STDC__ to ifdef __STDC__ because of Sun's acc compiler */
#ifdef __STDC__
#define PTR void *
#else
#define PTR char *
#endif

/* Add prototype support. */
#ifdef PROTO
#define PROTO(ARGS) ARGS
#endif

#ifndef STDC_HEADERS
#include <stdio.h>
#else
#include <stdlib.h>
#endif

#include <unistd.h>

#include <string.h>

#ifdef HAVE_ERRNO_H
#include <errno.h>
#else
ifndef errno
extern int errno;
#endif /* HAVE_ERRNO_H */
#endif

#include "system.h"
#include "hash.h"
#ifndef SERVER_SUPPORT || defined(CLIENT_SUPPORT)
#include "client.h"
#endif

#ifdef MY_NDBM

#include "myndbm.h"
#endif
#include <ndbm.h>
#endif /* MY_NDBM */
#include "regex.h"
#include "getopt.h"
#include "wait.h"
#include "ent.h"
#include "vms.h"
#include ".h"
#include "myndbm.h"

#define CVSADM "CVS"
#define CVSADM_ENT "CVS/Entries."
#define CVSADM_ENTBAK "CVS/Entries.Backup"
#define CVSADM_ENTLOG "CVS/Entries.Log"
#define CVSADM_ENTSTAT "CVS/Entries.Static"
#define CVSADM_REPO "CVS/Repository."
#define CVSADM_REPO_REMOTE "CVS/Repository.remote"
#define CVSADM_REMOTE_TMP "CVS/Remote"
#define CVSADM_REPO_ROOT "CVS/Root."
#define CVSADM_REPO_CIPROG "CVS/Checkin.prog"
#define CVSADM_REPO_PROG "CVS/Update.prog"
#define CVSADM_REPO_TAG "CVS/Tag."
#define CVSADM_REPO_NOTIFY "CVS/Notify."
#define CVSADM_REPO_NOTIFY_TMP "CVS/Notify.tmp"
#define CVSADM_REPO_BASE "CVS/Base"
#define CVSADM_REPO_BASEREV "CVS/Baserev."
#define CVSADM_REPO_BASEREV_TMP "CVS/Baserev.tmp"
#define CVSADM_REPOTEMPLATE "CVS/Template."
#define CVSADM_REPO_REMOTES比赛 "CVS/Repository.remote"
#define CVSADM_REPO_REMOTES_USERS "CVS/Repository.user"
#define CVSADM_REPO_REMOTES_REMOTE "CVS/Repository.remote"
#define CVSADM_REPO_REMOTES_REMOTE_TMP "CVS/Remote"
#define CVSADM_REPO_REMOTES_ROOT "CVS/Root."
#define CVSADM_REPO_REMOTES_CIPROG "CVS/Checkin.prog"
#define CVSADM_REPO_REMOTES_PROG "CVS/Update.prog"
#define CVSADM_REPO_REMOTES_TAG "CVS/Tag."
#define CVSADM_REPO_REMOTES_NOTIFY "CVS/Notify."
#define CVSADM_REPO_REMOTES_NOTIFY_TMP "CVS/Notify.tmp"
#endif
*/
#include "system.h".  Note that the ONLY reason for
this is if various system calls (getwd, getcwd, readlink) require/want
us to use it.  All other parts of CVS allocate pathname buffers
dynamically, and we want to keep it that way.  */
#endif
#define PATH_MAX
#define MAXPATHLEN
#define PATH_MAX 1024+2
#endif
*/
Definitions for the CVS Administrative directory and the files it contains.
Here are #define's to make changing the names a simple task.  */

/* A directory in which we store base versions of files we currently are
   editing with "cvs edit". */
#define CVSADM_REPO_BASE "CVS/Base"
#define CVSADM_REPO_BASEREV "CVS/Baserev."
#define CVSADM_REPO_BASEREV_TMP "CVS/Baserev.tmp"
/* File which contains the template for use in log messages. */
#define CVSADM_REPOTEMPLATE "CVS/Template."
#endif /* USE_VMS_FILENAMES */
#define CVSADM_REPO_CACHED_REMOTE_SEPARATOR "->"
*/
/* This is the special directory which we use to store various extra
   per-directory information in the repository.  It must be the same as
   CVSADM to avoid creating a new reserved directory name which users cannot
   use, but is a separate #define because if anyone changes it (which I don't
   recommend), one needs to deal with old, unconverted, repositories.

See fileattr.h for details about file attributes, the only thing stored
in CVSREP currently. */
#define CVSREP "CVS"
#define CVSROOTADM "$CVSROOT"
#define CVSROOTADM_MODULES "modules"
#define CVSROOTADM_LOGINFILE "logininfo"
#define CVSROOTADM_RCSINFO "rcsinfo"
#define CVSROOTADM_COMMITINFO "commitinfo"
#define CVSROOTADM_TAGINFO "taginfo"
#define CVSROOTADM_EDITINFO "editinfo"
#define CVSROOTADM_VERIFYMSG "verifymsg"
#define CVSROOTADM_HISTORY "history"
#define CVSROOTADM_VALTAGS "valtags"
#define CVSROOTADM_NOTIFY "notify"
#define CVSROOTADM_USERS "users"
#define CVSROOTADM_READERS "readers"
#define CVSROOTADM_WRITERS "writers"
#define CVSROOTADM_PASSWD "passwd"
#define CVSROOTADM_CONFIG "config"

#define CVSNULLREPOS "Emptydir" /* an empty directory */

/* Other CVS file names */

/* Files go in the attic if the head main branch revision is dead; otherwise they go in the regular repository directories. The whole concept of having an attic is sort of a relic from before death support but on the other hand, it probably does help the speed of some operations (such as main branch checkouts and updates). */
#define CVSSATTIC "Attic"

#define CVSLOCK "$cvs.lock"
#define CVSRFL "$cvs.rfl"
#define CVSWFL "$cvs.wfl"
#define CVSRFLPAT "$cvs.rfl.*" /* wildcard expr to match read locks */
#define CVSNULLREPOS "Emptydir"
#define CVSEXITLOG "$x" 
#define CVSPREFIX "$cvsignore"
#define CVSIGNORE "$cvsignores"
#define CVSWRAPPER "$cvswrappers"
#define CVSDOTWRAPPER "$cvswrappers"
#define CVSDOTIGNORE "$cvsignores"
#define CVSTAG "$cvsignors"
#define CVSDOTWRAPPER "$cvswrappers"

/* Command attributes -- see function lookup_commandattribute(). */
#define CVS_CMD_IGNORE_ADMROOT 1

#define CVS_CMD_USES_WORK_DIR 2

/* miscellaneous CVS defines */

/* This is the string which is at the start of the non-log-message lines that we put up for the user when they edit the log message. */
#define CVSEDITPREFIX "$es:

/* Number of characters in CVSEDITPREFIX to compare when deciding to strip off those lines. We don't check for the space, to accommodate users who have editors which strip trailing spaces. */
#define CVSEDITPREFIXLEN 4

#define CVSLOCKAGE (60*60) /* 1-hour old lock files cleaned up */
#define CVSLCKSLEEP 30 /* wait 30 seconds before retrying */
#define CVSBRANCH "1.1.1" /* RCS branch used for vendor srcs */

ifdef USE_VMS_FILENAMES
#define BARPREFIX "$A" /* RCS branch used for vendor srcs */
define DEVNULL "HEAD"
else /* USE_VMS_FILENAMES */
define BARPREFIX "$-" /* when remerge'ing */
endif
#endif

#define DEVNULL="/dev/null"

define USE_VMS_FILENAMES /*

*/ Special tags. -rHEAD refers to the head of an RCS file, regardless of any + sticky tags. -rBASE refers to the current revision the user has checked + out. This mimics the behaviour of RCS.

*/

#define TAG_HEAD "HEAD"
#define TAG_BASE "BASE"

/* Environment variable used by CVS */
#define CVSSREAD_ENV "CROSS" /* make files read-only */
#define CVSREAD_DFLT 0 /* writable files by default */
#define TMPDIR_ENV "TMPDIR" /* Temporary directory */
/ * #define TMPDIR_DFLT Set by options.h */
#define EDITOR1_ENV "VISUAL" /* which editor to use */
#define EDITOR2_ENV "EDITOR1" /* which editor to use */
#define EDITOR3_ENV "EDITOR2" /* which editor to use */
/* #define EDITOR_DFLT Set by options.h */
#define CVSROOT_ENV "CVSROOT" /* source directory root */
#define CVSROOT_DFLT NULL /* No dflt, must set for checkout */
#define IGNORE_ENV "CVSIGNORE" /* More files to ignore */
#define WRAPPER_ENV "CVSWRAPPER" /* name of the wrapper file */
#define CVSUMASK_ENV "CVSUMASK" /* Effective mask for repository */
/* #define CVSUMASK_DFLT Set by options.h */
/
* If the beginning of the Repository matches the following string, strip it
* so that the output to the logfile does not contain a full pathname.
* If the CVSROOT environment variable is set, it overrides this define.
*/
#define REPOS_DIR "master/"
/* Large enough to hold DATEFORM. Not an arbitrary limit as long as
it is used for that purpose, and not to hold a string from the
command line, the client, etc. */
#define MAXDATELEN 50
/
* The type of an entnode. */
enum ent_type
*
{ ENT_FILE, ENT_SUBDIR
};
/* structure of a entry record */
struct entnode
*
{
 enum ent_type type;
 char *user;
 char *version;
/
* Timestamp, or "" if none (never NULL). */
 char *timestamp;
/
* Keyword expansion options, or "" if none (never NULL). */
 char *options;
 char *tag;
 char *date;
 char *conflict;
 char *root;
 char *repository;
};

typedef struct entnode Entnode;
/
* The type of request that is being done in do_module() */
enum ntype
*
{ CHECKOUT, TAG, PATCH, EXPORT
};
/
* structure used for list-private storage by Entries_open() and
* Version_T2() and Find_Directories() . */
*
struct stickydirtag
{
 /
* These fields pass sticky tag information from Entries_open() to
* Version_T2(). */
 int affag;
 char *tag;
 char *date;
 int nonbranch;
 /
* This field is set by Entries_open() if there was subdirectory
* information; Find_Directories() uses it to see whether it needs
* to scan the directory itself. */
 int subdirs;
};
/
* Flags for find(names,dirs) routines */
#define W_LOCAL 0x01 /* look for files locally */
#define W_REPOS 0x02 /* look for files in the repository */
#define W_ATTIC 0x04 /* look for files in the attic */
/
* Flags for return values of direnter procs for the recursion processor */
enum direnter_type
*
{ R_PROCESS = 1, /* process files and maybe dirs */
 R_SKIP_FILES, /* don't process files in this dir */
typedef enum diretortype Dtype;

def

extern char *program_name, *program_path, *command_name;
extern char *Tempdir, +Editor;

extern int cvadmin_root;
extern char *CurDir;
extern int reallyquiet, quiet;
extern int use_editor,
extern int cvwrite;
extern int cvsadmin
extern int first_filearg;
extern int handling_remotes;
extern int fetch_remotes;

extern char *method_names[]; /* change this in root.c if you change the enum above */

extern char *CVSroot_original; /* the active, complete CVSroot string */
extern int clientactive; /* nonzero if we are doing remote access */
extern char *CVSroot_method; /* one of the r enum values above */

/* Flags used by RCS functions. See the description of the individual
 functions for which flags mean what for each function. */
#endif

#define RCS_FLAGS_FORCE 1
#define RCS_FLAGS_DEAD 2
#define RCS_FLAGS_QUIET 4
#define RCS_FLAGS_MODTIME 8
#define RCS_FLAGS_KEEPFILE 16

extern int RCS_exec_rdiff PROTO ((RCSNode *rcsfile,
char *opts, char *options,
char *rev1, char *rev2,
char +label1, char +label2,
char +workfile));

extern int diff_exec PROTO ((char +file1, char +file2, char +options,
char *out));

extern int diff_execv PROTO ((char +file1, char +file2,
char +label1, char +label2,
char +options, char +out));

#include "error.h"

DBM *open_module PROTO((void));
FILE *open_file PROTO((const char *, const char *));
List +FilesDirectories PROTO((char +repository, int which, List +entries));


void Entries_Close PROTO((List +entries));
List +Entries_Open PROTO ((int alflag, char +update_dir));
void Subdir_Known PROTO((List +entries));
void Subdir_Register PROTO((List +, const char +, const char +));
void Subdir_unregister PROTO((List +, const char +, const char +));
char +Make_Date PROTO((char +rawdate));
char +date_from_time PROTO ((time_t));
char *Name = Repository PROTO(char *dir, char *update, dir);
void Sanitize_Repository PROTO(char *repository);
char *Name = Root PROTO(char *dir, char *update, dir);
int parse PROTO(char *CVSroot);
void set_local PROTO(char *dir);
void Create_PROTO(char *dir, char *rootdir);
void rootallowadd PROTO((char *));
void rootallowfree PROTO(void);
int rootallowlock PROTO((char *));

char *gca PROTO(const char *rev1, const char *rev2);
extern void check_numeric PROTO(const char *, int, char **);
char *getcaller PROTO(void);
char *time stamp PROTO(char *file);
char *xmalloc PROTO((size_t bytes));
void xcalloc PROTO(void *, int, size_t bytes);
void expand_string PROTO((char **, size_t *, size_t));
char *xstrdup PROTO((const char *));

int stripparentnewlines PROTO(char *str);
int pathname_levels PROTO((char *path));
typedef int (*CALLPROC) PROTO((char *repository, char *value));
int Parse_info PROTO(char *infofile, char *repository, CALLPROC callproc, int all);
extern int parse_config PROTO((char *));
typedef RETSIGTYPE (*SIGCLEANUPPROC) PROTO(void);
int SIG_register PROTO((sig int sig, SIGCLEANUPPROC sigcleanup));
int isdir PROTO(const char *file);
int isfile PROTO(const char *file);
int islink PROTO(const char *file);
int isdevice PROTO((const char *));
int isxdev PROTO(const char *file);
int iswritable PROTO(const char *file);
int isreadable PROTO(const char *file);
int islink PROTO(const char *file);
int isdir PROTO(const char *file);
int islink PROTO(const char *file);
int isabsolute PROTO(const char *);  
char *xreadlink PROTO(const char *link);
char *lastcomponent PROTO((char *path));
char *gethomeidir PROTO(void);
char *gethomeidir PROTO(void);

int numdots PROTO((const char *));
char violation_revnunm PROTO(const char *); 
int compare_revnunms PROTO(const char *, int, const char *);
int unlink_file_PROTO(const char *);
int lslink file_PROTO(const char *);
int lslink file_dir PROTO(const char *);
int update_PROTO(int argc, char *argv[]);
int xcmp_PROTO(const char *file1, const char *file2);
int yesno PROTO(void);

void svalloc PROTO((size_t bytes));
time_t getdate_PROTO(char *date, struct timeb *now);
int Create_Admin_PROTO((char *dir, char *update, dir, 
char *repository, char *tag, char *date, 
int nonbranch, int warn));
extern int expandwildcardsigns PROTO((char *, off_t, FILE *));

/* Locking subsystem (implemented in lock.c). */

int Reader_Lock_PROTO((char *repository));

void LockCleanup_PROTO(void);

/* Write lock an entire subtree, well the part specified by ARGC, ARGV, LOCAL, 
and AFLAG; anyway. */
void lockfile_LOCK_WRITE_PROTO((int argc, char **argv, int local, int aflag));

/* See lock.c for description. */
extern void lockfile_LOCK_WRITE_PROTO((char *));

void Scratch_Entry_PROTO((List * list, char *fname));
void ParseTag_PROTO((char **tag, char **date, int *nonbranch));
void WriteTag_PROTO((char **dir, char *tag, char *date, int nonbranch, 
char *update, dir, char *repository));
void cat_module_PROTO((int status));
void checkentries_PROTO(char *dir);
void close_module_PROTO(DBM * db);
void copyfile_PROTO((const char *from, const char *to));
void perror_PROTO((FILE * fp, int status, int errno, char *message...));
void free_names PROTO((int *pargs, char **args));

extern int ign_name_PROTO((char *name));
void ign_add_PROTO(char *sign, int hold);
void ign_add_file_PROTO(char *file, int hold);
void ign_setup_PROTO(void);
void ign_alladd_PROTO(char *name);
int ignore_directory_PROTO(char *name);
typedef void (*Ignore_proc) PROTO((char *, char *));
extern void ignore_files_PROTO((List * list, char * sign, Ignore_proc));
extern int ign_inhibit_server;
...
220 – Source code

630

640

650

char ∗repository));
int start recursion PROTO((FILEPROC ﬁleproc, FILESDONEPROC ﬁlesdoneproc,
DIRENTPROC direntproc, DIRLEAVEPROC dirleaveproc,
void ∗callerdat,
int argc, char ∗argv[ ], int local, int which,
int aﬂag, int readlock, char ∗update preload,
int dosrcs));
void SIG beginCrSect PROTO((void));
void SIG endCrSect PROTO((void));
void read cvsrc PROTO((int ∗argc, char ∗∗∗argv, char ∗cmdname));
char ∗make message rcslegal PROTO((char ∗message));
extern int ﬁle has markers PROTO ((const struct ﬁle info ∗));
extern void get ﬁle PROTO ((const char ∗, const char ∗, const char ∗,
char ∗∗, size t ∗, size t ∗));
/∗ ﬂags
#deﬁne
#deﬁne
#deﬁne
#deﬁne
#deﬁne
#deﬁne
#deﬁne

for run exec(), the fast system() for CVS ∗/
RUN NORMAL 0x0000 /∗ no special behaviour ∗/
RUN COMBINED 0x0001 /∗ stdout is duped to stderr ∗/
RUN REALLY 0x0002 /∗ do the exec, even if noexec is on ∗/
RUN STDOUT APPEND 0x0004 /∗ append to stdout, don’t truncate ∗/
RUN STDERR APPEND 0x0008 /∗ append to stderr, don’t truncate ∗/
RUN SIGIGNORE 0x0010 /∗ ignore interrupts for command ∗/
RUN TTY (char ∗)0 /∗ for the beneﬁt of lint ∗/

void run arg PROTO((const char ∗s));
void run print PROTO((FILE ∗ fp));
void run setup PROTO ((const char ∗prog));
int run exec PROTO((const char ∗stin, const char ∗stout, const char ∗sterr,
int ﬂags));

660

/∗ other similar-minded stuﬀ from run.c. ∗/
FILE ∗run popen PROTO((const char ∗, const char ∗));
int piped child PROTO((char ∗∗, int ∗, int ∗));
void close on exec PROTO((int));
int ﬁlter stream through program PROTO((int, int, char ∗∗, pid t ∗));
pid t waitpid PROTO((pid t, int ∗, int));

670

680

690

/∗
∗ a struct vers ts contains all the information about a ﬁle including the
∗ user and rcs ﬁle names, and the version checked out and the head.
∗
∗ this is usually obtained from a call to Version TS which takes a
∗ tag argument for the RCS ﬁle if desired
∗/
struct vers ts
{
/∗ rcs version user ﬁle derives from, from CVS/Entries.
It can have the following special values:
NULL = ﬁle is not mentioned in Entries (this is also used for a
directory).
“” = ILLEGAL! The comment used to say that it meant “no user ﬁle”
but as far as I know CVS didn’t actually use it that way.
Note that according to cvs.texinfo, “” is not legal in the
Entries ﬁle.
0 = user ﬁle is new
-vers = user ﬁle to be removed. ∗/
char ∗vn user;
/∗ Numeric revision number corresponding to ->vn tag (->vn tag
will often be symbolic). ∗/
char ∗vn rcs;
/∗ If ->tag is a simple tag in the RCS ﬁle–a tag which really
exists which is not a magic revision–and if ->date is NULL,
then this is a copy of ->tag. Otherwise, it is a copy of
->vn rcs. ∗/
char ∗vn tag;
/∗ -> vn remote is the remote tag, in the server:tag form ∗/
char ∗vn remote;

700

710

/∗ This is the timestamp from stating the ﬁle in the working directory.
It is NULL if there is no ﬁle in the working directory. It is
“Is-modiﬁed” if we know the ﬁle is modiﬁed but don’t have its
contents. ∗/
char ∗ts user;
/∗ Timestamp from CVS/Entries. For the server, ts user and ts rcs
are computed in a slightly diﬀerent way, but the fact remains that
if they are equal the ﬁle in the working directory is unmodiﬁed
and if they diﬀer it is modiﬁed. ∗/
char ∗ts rcs;
/∗ Options from CVS/Entries (keyword expansion), malloc’d. If none,
then it is an empty string (never NULL). ∗/
char ∗options;
/∗ If non-NULL, there was a conﬂict (or merely a merge? See merge ﬁle)
and the time stamp in this ﬁeld is the time stamp of the working
directory ﬁle which was created with the conﬂict markers in it.


This is from CVS/Entries.

char *s_conflict;

/* Tag specified on the command line, or if none, tag stored in CVS/Entries. */
char *tag;

/* Date specified on the command line, or if none, date stored in CVS/Entries. */
char *date;

/* If this is 1, then tag is not a branch tag. If this is 0, then tag may or may not be a branch tag. */
int nonbranch;

/* Pointer to entries file node. */
Entnode *entdata;

/* Pointer to parsed src file info */
RCSNode *srcfile;

};
typedef struct vers_ts Vers_TS;

Vers_TS *Version_TS PROTO ({{struct file_info *finfo, char *options, char *tag, char *date, int force_tag_match, int set_time}});

void freevers_TS PROTO ({{Vers_TS ** vers}});

/* Miscellaneous CVS infrastructure which layers on top of the recursion processor (for example, needs struct file_info). */
int Checkin PROTO ({{int type, struct file_info *finfo, char *rcs, char *rev, char *tag, char *options, char *message}});

int No_Difference PROTO ({{struct file_info *finfo, Vers_TS *vers}});

/* TODO: can the finfo argument to special_file_mismatch be changed? */
int special_file_mismatch PROTO ({{struct file_info *finfo, char *rev1, char *rev2}});

/* CVSADM/DELABASE stuff, from entries.c. */
extern char *base_directory PROTO ({{struct file_info *}});
extern void base_register PROTO ({{struct file_info *+, char *s}});
extern void base_unregister PROTO ({{struct file_info *s}});

/* * defines for Classify() to determine the current state of a file. * These are also used as types in the data field for the list we make for * Update_Logfile in commit, import, and add. */
enum classify_type {
    T_UNKNOWN = 1, /* no old-style analog existed */
    T_CONFLICT,  /* C (conflict) list */
    T_NEEDS_MERGE,  /* G (needs merging) list */
    T_MODIFIED,  /* M (needs checked in) list */
    T_CHECKOUT, /* O (needs checkout) list */
    T_ADDED,  /* A (added file) list */
    T_REMOVED,  /* R (removed file) list */
    T_REMOVEENTRY, /* W (removed entry) list */
    T_UPTODATE,  /* File is up-to-date */
    SIGNED,  /* P like C, but can patch */
    T_PATCH,
    T_TITLE,  /* title for node type */
    T_REMOTE,  /* go to another server and ask */
};
typedef enum classify_type Ctype;

Ctype ClassifyFile PROTO ({{struct file_info *finfo, char *tag, char *date, char *options, int force_tag_match, int aflag, Vers_TS *vers, int pipeout}});

/* * structure used for list nodes passed to Update_Logfile() and * for editor(). */
struct logfile_info {
    enum classify_type type;
    char *tag;
    char *revold;  /* rev number before a commit/modify, NULL for add or import */
    char *revnew;  /* rev number after a commit/modify, add, or import, NULL for remove */
};

/* Wrappers. */
typedef enum { WRAP_MERGE, WRAP_COPY } WrapMergeMethod;
typedef enum {
    /* -t and -f wrapper options. Treating directories as single files. */
    WRAP_TOCVS, WRAP_TROMCVS,
extern void *wrap_setupPROTO(void);

void wrap_setupPROTO(void);

int wrap_name_hasPROTO(const char *name, WrapMergeHas *has);
char *wrap ExcelPROTO(const char *fileName, int asFlag);
char *wrap_doct POLPROTO(const char *fileName);
int wrap_merge_in POLPROTO(const char *fileName);
void wrap from POLPROTO(const char *fileName);
void wrap_add POLPROTO(const char *fileName);

820 void wrap_add PROTOr(char *line, int temp);
void wrapデン POLPROTO(void);

 ifdef SERVER_SUPPORT | defined(CLIENT_SUPPORT)
void wrap_unparse POLPROTO((char *)), int);
tendif /* SERVER_SUPPORT | CLIENT_SUPPORT */

 /* Pathname expansion */
char *expand_path POLPROTO((char *name, char *file, int line));

 /* User variables. */

 extern List *variable list;
 extern void variableset PROTOr((char *nameval));
 int watch PROTO((int argc, char **argv));
 int edit PROTO((int argc, char **argv));
 int unedit PROTO((int argc, char **argv));
 int editors PROTO((int argc, char **argv));
 int watchers PROTO((int argc, char **argv));
 external int annotate POLPROTO((int argc, char **argv));
 external int add POLPROTO((int argc, char **argv));
 external int admin PROTO((int argc, char **argv));
 external int commit PROTO((int argc, char **argv));
 external int diff PROTO((int argc, char **argv));
 external int history PROTO((int argc, char **argv));
 external int import PROTO((int argc, char **argv));
 external int cvstag POLPROTO((int argc, char **argv));

 ifdef AUTH_CLIENT_SUPPORT
 external int login POLPROTO((int argc, char **argv));

 ifdef /* AUTH_CLIENT_SUPPORT */
 external int logout POLPROTO((int argc, char **argv));

tendif /* AUTH_CLIENT_SUPPORT */

define AUTH_KERBEROS_V4 2

define AUTH_PASSWORD 3

870 ifdef AUTH_CLIENT_SUPPORT
char *get PASSWD PASSWORD PROTO(void);

tendif /* AUTH_CLIENT_SUPPORT */

 external void tag_check valid POLPROTO((char *), int, char **, int, int, char *);

 extern void cv u output POLPROTO((const char *, size_t));
 extern void cv u output POLPROTO((char *, size_t));
 extern void cv u output POLPROTO((const char *, size_t));
 extern void cv u output POLPROTO((void));
 extern void cv u output POLPROTO((void));
 extern void cv u output POLPROTO((char *, char *));

 ifdef SERVER_SUPPORT | defined(CLIENT_SUPPORT)
 include "server.h"

tendif
# include "cvs.h"
# include "getline.h"

/* this file is to be found in the user's home directory */

#ifndef CVSRC
#define CVSRC_FILENAME ".cvsrc"
#endif

define CVSRC_FILENAME "">cvsrc.c – 223

#define GROW 10

define strtok () ;

/* Read cvsrc, processing options matching CMDNAME ("cvs" for global
options, and update *ARGC and *ARGV accordingly. */

void

read_cvsrc (argc, argv, cmdname)

{ 

int *argc;
char ***argv;
char *cmdname;

char *homedir;
char *homeinit;
FILE *cvsrcfile;

char *line;
int line_length;
size_t line_chars_allocated;

char *optstart;

int command_len;
int found = 0;

int i ;

int new_argc;
int new_argv;
char **new_argv;

/* old_argc and old_argv hold the values returned from the
previous invocation of read_cvsrc and are used to free the
allocated memory. The first invocation of read_cvsrc gets argv
from the system, this memory must not be free'd. */

static int old_argc = 0;
static char ** old_argv = NULL;

/* don't do anything if argc is -1, since that implies "help" mode */

if ( argc == -1 )

return ;

/* determine filename for "/.cvsrc * /

homedir = get_homedir () ;
if ( homedir )

return ;

homeinit = ( char *) xmalloc ( strlen ( homedir ) + strlen ( cvsrc ) + 10);
strcpy ( homeinit, homedir);
strcat ( homeinit, "/" );
strcat ( homeinit, cvsrc );

/* if it can't be read, there's no point to continuing */

if ( ! isreadable ( homeinit ) )

{ 

free ( homeinit );

return ;
}

/* now scan the file until we find the line for the command in question */

line = NULL;
line_chars_allocated = 0;
command_len = strlen ( cmdname );
cvsrcfile = openfile (homeinit, "r");
while ((line_length = getline (line, &line_chars_allocated, cvsrcfile))
  >= 0)
{
  /* skip over comment lines */
  if (line[0] == '#')
    continue;
  /* stop if we match the current command */
  if (strncmp (line, cmdname, command_len) & & isspace (* (+line + command_len)))
    { found = 1;
      break;
    }
if (line_length < 0 & & feof (cvsrcfile))
  error (0, errno, "cannot read \%s", homeinit);
  fclose (cvsrcfile);
/* setup the new options list */
new_argc = 1;
max_new_args = (argc + GROW);
new_argv = (char **) xmalloc (max_new_args + sizeof (char*));
new_argv[0] = xstrdup ((argv)[0]);
if (found)
  { /* skip over command in the options line */
    for (optstart = strtok (line + command_len, "\t \n");
       optstart;
       optstart = strtok (NULL, "\t \n");
    new_argv [new_argc++] = xstrdup (optstart);
    if (new_argc >= max_new_args)
      { max_new_args += GROW;
        new_argv = (char **) xrealloc (new_argv, max_new_args + sizeof (char*));
      }
  }
if (line != NULL)
  free (line);
/* now copy the remaining arguments */
if (new_argc + argc > max_new_args)
  { max_new_args = new_argc + argc;
    new_argv = (char **) xrealloc (new_argv, max_new_args + sizeof (char*));
  }
for (i=1; i < argc; i++)
  { new_argv [new_argc++] = argv[i];
  }
if (old_argv != NULL)
  { /* Free the memory which was allocated in the previous read/argv call */
    free (old_argv);
  }
old_argv = new_argv;
old_argc = new_argc;
free (homeinit);
return;
}
A.14  diff.c

/* * Copyright (c) 1992, Brian Berliner and Jeff Polk *
 * Copyright (c) 1989-1992, Brian Berliner *
 * You may distribute under the terms of the GNU General Public License as *
 * specified in the README file that comes with the CVS source distribution. *
 * *
 * Difference *
 * *
 * Run diff against versions in the repository. Options that are specified are *
 * passed on directly to "rcsdiff". *
 * *
 * Without any file arguments, runs diff against all the currently modified *
 * files. *
 */

#include "cvs.h"

enum diff_file
{
    DIFF_ERROR,
    DIFF_ADDDED,
    DIFF_REMOVED,
    DIFF_DIFFERENT,
    DIFFSAME
};

static Dtype diffdirproc PROTO ((void *callerdat, char *dir, char *repos, char *update_dir, const char *err, char *dir, List entries));

static int difflesdoneproc PROTO ((void *callerdat, int err, char *repos, char *update_dir, List entries));

static int diff_dirleavproc PROTO ((void *callerdat, char *dir, int err, char *update_dir, List entries));

static enum diff_file diff_file_modif Diff ((struct file_info *info, VersT8 vers, enum diff_file));

static int diff_fileproc PROTO ((void *callerdat, struct file_info *info));

static void diff_mark_errors PROTO ((int err));

static char *diff_rev1, *diff_rev2;
static char *diff_date1, *diff_date2;
static char *user_file;
static char *options;
static char *opts;
static size_t opts_allocated = 1;
static int diff_errors;
static int empty_files = 0;

/* FIXME: should be documenting all the options here. They don't *
 * perfectly match rcsdiff options (for example, we always support *
 * -ifdef and -context, but rcsdiff only does if diff does). */

static const char *const diff_usage[] =
{
    "Usage: da Za [-199] [rcsdiff-options]n",
    " [-r rev1 | -D date1] [-r rev2 | -D date2] [files... ] n",
    "[-l]Local directory only, not recursive"n",
    "[-t]Diff revision for date against working file."n",
    "[-c]Diff revision for date against working file."n",
    "[-o] Diff revision for date against working file."n",
    "[-I] Revert file to diff against working file."n",
    "[-i] If a directory, output diffs in ifdef format."n",
    "[-I] If a directory, output diffs in ifdef format."n",
    "[Specify the --help global option for a list of other help options]n",
    NULL
};

/* I copied this array directly out of diff.c in diffutils 2.7, after *
 * removing the following entries, none of which seem relevant to use *
 * with CVS: *
 * -help *
 * -version *
 * -recursive *
 * -unidirectional-new-file *
 * -starting-file *
 * -exclude *
 * -exclude-from
I changed the options which take optional arguments (`-context` and `-unified`) to return a number rather than a letter, so that the optional argument could be handled more easily. I changed the `-paginate` and `-brief` options to return a number, since `-l` and `-q` mean something else to `cvs diff`.

The numbers that appear in the fourth element of some entries tell the big switch in `diff` how to process those options. — Ian

The following options, which `diff` lists as “An alias, no longer recommended” have been removed: `-file-label` `-entire-new-file` `-ascii` `-print`.

/*
static struct option const longopts[] =
{
    {"ignore-blank-lines", 0, 0, 'B'},
    {"context", 2, 0, 143},
    {"ifdef", 1, 0, 133},
    {"show-function-line", 1, 0, 'I'},
    {"ignore-matching-lines", 1, 0, 'I'},
    {"label", 1, 0, 'L'},
    {"new-file", 0, 0, 'N'},
    {"initial-tab", 0, 0, 148},
    {"width", 1, 0, 'W'},
    {"text", 0, 0, 'a'},
    {"ignore-space-change", 0, 0, 'b'},
    {"minimal", 0, 0, 'd'},
    {"ed", 0, 0, 'e'},
    {"forward-ed", 0, 0, 'f'},
    {"ignore-case", 0, 0, 'i'},
    {"paginate", 0, 0, 144},
    {"rcs", 0, 0, 'n'},
    {"show-c-function", 0, 0, 'p'},
    {"brief", 0, 0, 145},
    {"report-identical-files", 0, 0, 's'},
    {"expand-tabs", 0, 0, 't'},
    {"ignore-all-space", 0, 0, 'w'},
    {"side-by-side", 0, 0, 147},
    {"unified", 2, 0, 146},
    {"left-column", 0, 0, 129},
    {"suppress-common-lines", 0, 0, 130},
    {"old-line-format", 1, 0, 132},
    {"new-line-format", 1, 0, 133},
    {"unchanged-line-format", 1, 0, 134},
    {"line-format", 1, 0, 135},
    {"old-group-format", 1, 0, 136},
    {"new-group-format", 1, 0, 137},
    {"unchanged-group-format", 1, 0, 138},
    {"changed-group-format", 1, 0, 139},
    {"horizon-lines", 1, 0, 140},
    {"binary", 0, 0, 142},
    [0, 0, 0, 0]
};

/* CVS 1.9 and similar versions seemed to have pretty weird handling of -y and -T. In the cases where it called rcsdiff, they would have the meanings mentioned below. In the cases where it called diff, they would have the meanings mentioned in “longopts”. Noone seems to have missed them, so I think the right thing to do is just to remove the options altogether (which I have done).

In the case of -z and -q, “cvs diff” did not accept them even back when we called rcsdiff (at least, it hasn’t accepted them recently).

In comparing rcsdiff to the new CVS implementation, I noticed that the following rcsdiff flags are not handled by CVS diff:

- `-y` perform diff even when the requested revisions are the same revision number
- `-q` run quietly
- `-T` preserve modification time on the RCS file
- `-z` specify timezone for use in file labels

I think these are not really relevant. -y is undocumented even in RCS 5.7, and seems like a minor change at best. According to RCS documentation, -T only applies when a RCS file has been modified because of lock changes; doesn’t CVS sidestep RCS’s entire lock structure? -z seems to be unsupported by CVS diff, and has a different meaning as a global option anyway. (Adding it could be a feature, but if it is left out for now, it should not break...
anything.) For the purposes of producing output, CVS diff appears mostly to ignore -q. Maybe this should be fixed, but I think it’s a larger issue than the changes included here. */

static void strcat_and_allocate PROTO ((char **, size_t *, const char *));
/* sSTR is a pointer to a malloc’d string. *LENP is its allocated length. Add SRC to the end of it, reallocating if necessary. */

static void strcat_and_allocate (str, lenp, src)
char **str;
size_t *lenp;
const char *src;
{
size_t new_size;

new_size = strlen (*str) + strlen (src) + 1;
if (*lenp == NULL || new_size >= *lenp)
{
    while (new_size >= *lenp)
        *lenp *= 2;

    *str = xrealloc (*str, *lenp);
}
strcat (*str, src);
}

int diff (argc, argv)
int argc;
char *argv[];
{
char tmp[50];
int c, err = 0;
int local = 0;
int which;
int option_index;

if (argc == -1)
    usage (diff_usage);

have_rev1_label = have_rev2_label = 0;

exit (0);

exit (0);

while ((c = getopts_long (argc, argv, 
longopts, &option_index)) != -1)
{
    switch (c)
    {
    case ’a’:
    case ’b’:
    case ’c’:
    case ’d’:
    case ’e’:
    case ’f’:
        break;
    case ’h’:
    case ’i’:
    case ’j’:
    case ’k’:
    case ’l’:
        break;
    case ’n’:
    case ’o’:
    case ’p’:
    case ’q’:
    case ’r’:
        break;
    case ’s’:
    case ’t’:
    case ’u’:
    case ’v’:
    case ’w’:
    case ’x’:
    case ’y’:
    case ’z’:
    case ’0’:
    case ’1’:
    case ’2’:
    case ’3’:
    case ’4’:
    case ’5’:
    case ’6’:
    case ’7’:
    case ’8’:
    case ’9’:
        break;
    case ’v’:
    case ’V’:
    case ’W’:
    case ’w’:
        break;
    case ’c’:
    case ’C’:
    case ’F’:
    case ’f’:
    case ’F’:
    case ’G’:
    case ’g’:
        break;
    case ’d’:
    case ’D’:
    case ’I’:
    case ’i’:
    case ’L’:
    case ’l’:
    case ’R’:
        break;
    (void) sprintf (tmp, "%-5s", (char *) c); strcat_and_allocate (&opts, &opts_allocated, tmp);
    break;
    case ’L’:
        if (have_rev1_label++)
            error (0, 0, "extra -L arguments ignored");
 break;
    }
strcat_and_allocate (&opts, &opts_allocated, "-L"); strcat_and_allocate (&opts, &opts_allocated, optarg);
break;
    case ’G’:
    case ’F’:
    case ’I’:
    case ’O’:
    case ’P’:
    case ’W’:
        (void) sprintf (tmp, "%-5s", (char *) c); strcat_and_allocate (&opts, &opts_allocated, tmp);
    break;
strcat_and_allocate (&opts, &opts_allocated, optarg);
break;
    case ’C’:
    case ’F’:
    case ’I’:
    case ’O’:
    case ’P’:
    case ’W’:
        (void) sprintf (tmp, "%-5s", (char *) c); strcat_and_allocate (&opts, &opts_allocated, tmp);
    break;
strcat_and_allocate (&opts, &opts_allocated, optarg);
break;
    case 131:

*/

}
/* -ifdef. */
struct_andAllocate (&options, opts_allocated, "-?*");
struct_andAllocate (&options, opts_allocated, optarg);
break;
case 129: case 130: case 132: case 133: case 134:
case 135: case 136: case 137: case 138: case 139: case 140:
case 141: case 142: case 143: case 144: case 145: case 146:
case 147: case 148:
struct_andAllocate (&options, opts_allocated, "-?*");
struct_andAllocate (&options, opts_allocated, optarg);
break;
if ((longopt[option_index].have_arg == 1
 || (longopt[option_index].have_arg == 2
 && optarg != NULL))
{
struct_andAllocate (&options, opts_allocated, "-?*");
struct_andAllocate (&options, opts_allocated, optarg);
break;
case 'k':
local = 0;
break;
case 'l':
local = 1;
break;
case 'r':
if (options)
free (options);
options = RCS_check_kflag (optarg);
break;
case 'r':
if (diff_rev2 != NULL || diff_date1 != NULL)
error (1, 0,
"no more than two revisions/dates can be specified");
if (diff_rev1 != NULL || diff_date1 != NULL)
diff_rev2 = optarg;
else
diff_rev1 = optarg;
break;
case '0':
if (diff_rev2 != NULL || diff_date2 != NULL)
error (1, 0,
"no more than two revisions/dates can be specified");
if (diff_rev1 != NULL || diff_date1 != NULL)
diff_date2 = Make_Date (optarg);
else
diff_date1 = Make_Date (optarg);
break;
case 'x':
empty_files = 1;
break;
case 'y':
default:
usage (diff_usage);
break;
}
if (client_active && ! handling_remotes)
first_filename = arg;
arg = optarg;
argv += optarg;
/* make sure options is non-null */
if (options)
options = xstrdup (**);
#endif CLIENT_SUPPORT
if (client_active) {
/* We're the client side. Fire up the remote server. */
start_server ();
ign_setup ();
if (local)
send_arg("-1");
if (empty_files)
send_arg("-F");
send_option_string (opts);
if (options[0] != "-O")
send_arg (options);
if (diff_rev1)
option_with_arg ("-r", diff_rev1);
if (diff_date1)
client_senddate (diff_date1);
if (diff_rev2)
option_with_arg ("-r", diff_rev2);
if (diff_date2)
client_senddate (diff_date2);
360  send_file_names (argc, argv, SEND_EXPAND_WILD);

    /* Send the current files unless diffing two revs from the archive */
    if (diff_rev1 != NULL & diff_rev2 != NULL)
        send_files (argc, argv, local, 0, 0);
    else
        send_files (argc, argv, local, 0, SEND_NO_CONTENTS);

    send_server (*diff2*, 0);
    err = get_responses (endfile);  /* close */
    if (handling_err)
        return 2;

    if (options != NULL) {
        free (options);
        options = NULL;
    }
    if (diff_rev1 != NULL) {
        free (diff_rev1);
        diff_rev1 = NULL;
    }
    if (diff_rev2 != NULL) {
        free (diff_rev2);
        diff_rev2 = NULL;
    }
    if (diff_date1 != NULL) {
        free (diff_date1);
        diff_date1 = NULL;
    }
    if (diff_date2 != NULL) {
        free (diff_date2);
        diff_date2 = NULL;
    }
    return (err);
}

senddiff

    if (diff_rev1 != NULL)
        tag_check_valid (diff_rev1, argc, argv, local, 0, **);
    if (diff_rev2 != NULL)
        tag_check_valid (diff_rev2, argc, argv, local, 0, **);

    which = W_LOCAL;
    if (diff_rev1 != NULL || diff_date1 != NULL)
        which = W_REPOS | W_ATTIC;

    wrap_setup ();

    /* start the recursion processor */
    err = start_recursion (diff_fileproc, diff_findproc, diff_dirproc,
        diff_chdirproc, NULL, argc, argv, local, 0, 1, (char *) NULL, 1);

    /* clean up */
    free (options);
    return (err);

    /*
    * Do a file diff
    */
    *ARGSUSED */
    static int
diff_fileproc (callerdat, finfo)
        callerdat;
        void  *callerdat;
    {
        int status, err = 2;  /* 2 == trouble, like rsdiff */
        Ver = FS *vers;
        enum diff_file empty_file = DIFF_DIFFERENT;
        char *tmp;
        char *to cvsPath;
        char *fname;

        / * Initialize these solely to avoid warnings from gcc -Wall about
           variables that might be used uninitialized. */
        tmp = NULL;
        fname = NULL;

        user_file_local = 0;
        vers = Version_FS (finfo, NULL, NULL, NULL, 1, 0);

    stdef SERVER_SUPPORT
    {
        int have_remote = 0;
        Version_FS *vers1 = Version_FS (finfo, NULL, diff_rev1, diff_date1, 1, 0);
        Version_FS *vers2 = Version_FS (finfo, NULL, diff_rev2, diff_date2, 1, 0);
if ((vers1 != NULL) && (vers1 -> vn_remote != NULL)) {
    server_output_not_carried_for_file (info, vers1);
    have_remote = 1;
}

if ((vers2 != NULL) && (vers2 -> vn_remote != NULL)) {
    server_output_not_carried_for_file (info, vers2);
    have_remote = 1;
}

if (have_remote)
    return 0;
}

sendif

if (diffrev2 != NULL || diffdate2 != NULL)
{
    /* Skip all the following checks regarding the user file; we're
     * not using it. */
}
else if (vers->vn_user == NULL)
{
    /* The file does not exist in the working directory. */
    if (diffrev1 != NULL || diffdate1 != NULL)
        vers->arcfile != NULL)
    {
        /* The file does exist in the repository. */
        if (empty_views)
            emptyfile = DIFF_REMOVED;
        else
        {
            int exists = 0;
            /* special handling for TAG_HEAD */
            if (diffrev1 && strcmp (diffrev1, TAG_HEAD) == 0)
            {
                char *head =
                    (vers->vn_cs == NULL
                        ? NULL
                        : RCSbranchhead (vers->arcfile, vers->vn(s)));
                exists = head != NULL;
                if (head != NULL)
                    free (head);
            }
            else
            {
                Version_Ts *xvers;
                xvers = Version_Ts (info, NULL, diffrev1, diffdate1,
                    1, 0);
                exists = xvers->vn_cs != NULL;
                freesversion (xvers);
            }
            if (exists)
                error (0, 0,
                    "The file does not exist in the working directory."
                    "file", info->fullname);
                freesversion (xvers);
                diffmarkerrors (err);
                return (err);
        }
    }
    else
    {
        error (0, 0, "I know nothing about %s", info->fullname);
        freesversion (xvers);
        diffmarkerrors (err);
        return (err);
    }
}

else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '\'0')
{
    if (empty_views)
        emptyfile = DIFF_ADDED;
    else
    {
        error (0, 0, "The file no longer exists, no comparison available",
            info->fullname);
        freesversion (xvers);
        diffmarkerrors (err);
        return (err);
    }
}
else if (vers->vn_user[0] == '\t')
{
    if (empty_views)
        emptyfile = DIFF_REMOVED;
    else
    {
error (0, 0, "Isa was removed, no comparison available", finfo->fullname);
freevers_ts (&vers);
diffmark_errors (err);
return (err);
}
else {
  if (vers->vn_ts == NULL && vers->srcfile == NULL)
    error (0, 0, "cannot find revision control file for Isa", finfo->fullname);
  freevers_ts (&vers);
  diffmark_errors (err);
  return (err);
}
else if (ltrimcmp (vers->ts_user, vers->ts_rev))
  /
  * The user file matches some revision in the repository Diff against the repository (for remote CVS, we might not have a copy of the user file around). */
  user_file_rev = vers->vn_user;
}
}
emptyfile = difffilemodify (finfo, vers, emptyfile);
if (emptyfile == DIFFSAME || emptyfile == DIFFERROR)
  { freevers_ts (&vers);
    if (emptyfile == DIFFSAME)

      /* In the server case, would be nice to send a "Checked-in" response, so that the client can rewrite its timestamp. server_checkedln by itself isn't the right thing (it needs a server_registers), but I'm not sure what is. It isn't clear to me how "cvs status" handles this (that is, for a client which sends Modified not Is-modified to cvs status"), but it does. */
    return (0);
  }
else
  { diffmark_errors (err);
    return (err);
  }
}
if (emptyfile == DIFFDIFFERENT)
  { int dead1, dead2;
    if (userev1 == NULL)
      dead1 = 0;
    else
dead1 = RCSdead (vers->srcfile, userev1);
    if (userev2 == NULL)
      dead2 = 0;
    else
dead2 = RCSdead (vers->srcfile, userev2);

510
if (dead1 && dead2)
  { freevers_ts (&vers);
    return (0);
  } else if (dead1)
    { if (emptyfile) emptyfile = DIFFADDED;
      else
        { error (0, 0, "Isa is a new entry, no comparison available", finfo->fullname);
          freevers_ts (&vers);
          diffmark_errors (err);
          return (err);
        }
    } else if (dead2)
520
    { 
530
    }
{ if (empty_file) 
    empty_file = DIFF_REMOVED;
else 
{ 
    error (0, 0, "is was removed, no comparison available"; 
    finfo->fullname); 
    freeversinfo (vers); 
    diffmarkerrors (err); 
    return (err); 
} 
} 

/* Output an "Index:" line for patch to use */

cvs_output (*"Index: ", 0);
cvs_output (finfo->fullname, 0);
cvs_output (*"\n", 1);

tocvsPath = wrap_tocvs_process_file(finfo->file);

if (tocvsPath)
{ 
    /* Backup the current version of the file to CVS/>.filenam */
    fname = xmalloc (strlen (finfo->file) 
+ sizeof CVSADM 
+ sizeof CVSREFIX 
+ 10);

    if (rename_file (fname) < 0)
    { 
        error (1, errno, "cannot rename "fname".

    rename_file (finfo->file, fname);

    /* Copy the wrapped file to the current directory then go to work */
    copy_file (tocvsPath, finfo->file);
}

    if (empty_file == DIFF_ADDED || empty_file == DIFF_REMOVED)
{ 
    /* This is file, not fullname, because it is the "Index:" line which 
    is supposed to contain the directory. */
    cvs_output (*"\n"

    RCV files: *. 0);
cvs_output (finfo->file, 0);
cvs_output (*"\n", 1);

cvs_output (*"diff -X ", 0);
cvs_output (finfo->file, 0);
cvs_output (*"\n", 1);

    if (empty_file == DIFF_ADDED)
{ 
    if (use_rev2 == NULL)
        status = diffexec (DEVNULL, finfo->file, opts, RUN_TTY);
else 
{ 
    int retcode;

    tmp = cvs_tempname ();
    retcode = RCScheckout (vers->srcfile, (char *) NULL, 
    use_rev2, (char *) NULL, 
    options ? options : vers->options),
    tmp, (BCSCHECKOUTPROC) NULL,
    (void *) NULL);

    if (retcode != 0)
    { 
        diffmarkerrors (err);
        return err;
    }

    status = diffexec (DEVNULL, tmp, opts, RUN_TTY);
}
else 
{ 
    int retcode;

    tmp = cvs_tempname ();
    retcode = RCScheckout (vers->srcfile, (char *) NULL, 
    use_rev1, (char *) NULL, 
    options ? options : vers->options),
    tmp, (BCSCHECKOUTPROC) NULL,
    (void *) NULL);

    if (retcode != 0)
    { 
        diffmarkerrors (err);
        return err;
    }
}
status = diff_exec (tmp, DEVNULL, opts, RUN_TTY);
}
}
else
{
    char *label1 = NULL;
    char *label2 = NULL;

    if (*have_rev1-label)
        label1 = make_label (finfo->fullname, use->rev1, vers->srcfile);
    if (*have_rev2-label)
        label2 = make_label (finfo->fullname, use->rev2, vers->srcfile);

    status = RCSExec_rcsdiff (vers->srcfile, opts,
        +options ? options : vers->options,
        use->rev1, use->rev2,
        label1, label2,
        finfo->file);

    if (label1) free (label1);
    if (label2) free (label2);
}

switch (status)
{
    case -1: /* fork failed */
        error (1, errno, "fork failed while diffing %s",
            vers->srcfile->path);
        break;
    case 0: /* everything ok */
        err = 0;
        break;
    default: /* other error */
        err = status;
        break;
}

if (tocvsPath)
{
    if (unlink_file (finfo->file) < 0)
        if (! existence_error (errno))
            error (1, errno, "cannot remove %s", finfo->file);

        rename_file (finfo, finfo->file);
    if (unlink_file (tocvsPath) < 0)
        error (1, errno, "cannot remove %s", tocvsPath);
    free (finfo);
}

if (empty_file == DIFF_REMOVED ||
    (empty_file == DIFF_ADDED && use_rev2 != NULL))
{
    if (CVS unlink (tmp) < 0)
        error (0, errno, "cannot remove %s", tmp);
    free (tmp);
}

for (; ) {
    if (RCS_rewrite (vers->srcfile, xstrdup("1.1.3"),
            xstrdup("1.1.3"), 1) != 0)
        break;

    RCS_rewrite (vers->srcfile, NULL, NULL);
    freeverspes (&vers);
    diffmarkerrors (err);
    return (err);
}

/* * Remember the exit status for each file. */
static void
diffmarkerrors (err)
{
    int err;
    { if (err > differrors)
            differrors = err;
    }

    /* * Print a warm fuzzy message when we enter a dir
      * Don't try to diff directories that don't exist! - DW */
    /* ARGUSED */
    static Dtype
diffdirproc (callerdat, dir, pos, epos, update, dirproc, entries)
void callerdat;
char *dir;
char *repos;
char *update_dir;
List *entries;

/* XXX - check for dirs we don't want to process?? */
/* YES ... for instance dirs that don't exist!!! - DW */
if (isdir (dir))
    return (RcSPROCESS);

if (quiet)
    error (0, 0, "Diffing %s", update_dir);

return (RcSPROCESS);
}

/* Concoct the proper exit status - done with files
*/
/* ARGSUSED */
830 static int diff_fileoneproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
    int err;
    char *repos;
    char *update_dir;
    List *entries;
    { return (diff_errors);
    }

/* Concoct the proper exit status - leaving directories
*/
/* ARGSUSED */
850 char *update_dir;
    List *entries;
    { return (diff_errors);
    }

/* verify that a file is different
*/
860 static enum diff_file
diff_file_modified (finfo, vers, empty_file) struct file_info **finfo;
    Version_TS xvers;
    enum diff_file empty_file;
    { Version_TS *xvers;
      int retcode;
      /* free up any old use vars and react 'em */
      if (useRev1)
        free (useRev1);
      if (useRev2)
        free (useRev2);
      useRev1 = useRev2 = (char *) NULL;
      if (diffRev1 || diffDate1)
        { /* special handling for TAGHEAD */
          if (diffRev1 && strcmp (diffRev1, TAGHEAD) == 0)
            useRev1 = ((vers->vn_rev == NULL || vers->srcfile == NULL)
              && NULL
            ) - RCSbranchhead (vers->srcfile, vers->vn_rev);
          else
            { xvers = Version_TS (finfo, NULL, diffRev1, diffDate1, 1, 0);
              if (xvers->vn_rev != NULL)
                useRev1 = xstrdup (xvers->vn_rev);
              freevers (xvers);
            }
        }
      if (diffRev2 || diffDate2)
        { /* special handling for TAGHEAD */
          if (diffRev2 && strcmp (diffRev2, TAGHEAD) == 0)
            useRev2 = ((vers->vn_rev == NULL || vers->srcfile == NULL)
              && NULL
            ) - RCSbranchhead (vers->srcfile, vers->vn_rev);
          else
            { /* do something with dir */
            }
        }
    }
xvers = VersionTS (finfo, NULL, diffrev2, diffdate2, 1, 0);
if (xvers->vn_rev != NULL)
    userev2 = xstrdup (xvers->vn_rev);
free (xvers);
}

if (userev1 == NULL)
{
    /* The first revision does not exist. If EMPTY_FILES is
       true, treat this as an added file. Otherwise, warn
       about the missing tag. */
    if (userev2 == NULL)
        /* At least in the case where DIFFREV1 and DIFFREV2
           are both numeric, we should be returning some kind
           of error (see basic-8a0 in testsuite). The symbolic
           case may be more complicated. */
        return DIFFSAME;
    else if (empty_files)
        return DIFFADDED;
    else if (diffrev1)
        error (0, 0, "tag %s is not in file %s", diffrev1,
               finfo->fullname);
    else
        error (0, 0, "no revision for date %s in file %s",
               diffdate1, finfo->fullname);
    return DIFFERROR;
}

if (userev2 == NULL)
{
    /* The second revision does not exist. If EMPTY_FILES is
       true, treat this as a removed file. Otherwise warn
       about the missing tag. */
    if (empty_files)
        return DIFFADDED;
    else if (diffrev2)
        error (0, 0, "tag %s is not in file %s", diffrev2,
               finfo->fullname);
    else
        error (0, 0, "no revision for date %s in file %s",
               diffdate2, finfo->fullname);
    return DIFFERROR;
}

/* now, see if we really need to do the diff */
if (strcmp (userev1, userev2) == 0)
    return DIFFSAME;
else
    return DIFFDIFFERENT;

if ((diffrev1 || diffdate1) && userev1 == NULL)
{
    /* The first revision does not exist, and no second revision
       was given. */
    if (empty_files)
    {
        if (empty_file == DIFFADDED)
            return DIFFSAME;
        else
        {
            if (user_file_rev && userev2 == NULL)
                userev2 = xstrdup (user_file_rev);
            return DIFFADDED;
        }
    }
    else
    {
        if (diffrev1)
            error (0, 0, "tag %s is not in file %s", diffrev1,
                   finfo->fullname);
        else
            error (0, 0, "no revision for date %s in file %s",
                   diffdate1, finfo->fullname);
        return DIFFERROR;
    }
}

if (user_file_rev)
{
    /* drop user_file_rev into first unused userev */
    if (userev1)
        userev1 = xstrdup (user_file_rev);
    else if (userev2)
        userev2 = xstrdup (user_file_rev);
    /* and if not, it wasn't needed anyhow */
    use_file_rev = 0;
}

/* now, see if we really need to do the diff */
if (use_rev1 && use_rev2)
{
    if (strcmp(use_rev1, use_rev2) == 0)
        return DIFFSAME;
    else
        return DIFFDIFFERENT;
}

if (use_rev1 == NULL || (vers->vn_user != NULL && strcmp(use_rev1, vers->vn_user) == 0))
{
    if (empty_file == DIFFDIFFERENT
        && vers->ts_user != NULL
        && strcmp(vers->ts_rcs, vers->ts_user) == 0
        && (!(*options) || strcmp(options, vers->options) == 0))
    {
        return DIFFSAME;
    }
    if (use_rev1 == NULL
        && (vers->vn_user[0] != '0' || vers->vn_user[1] != '^'))
    {
        use_rev1 = xstrdup(vers->vn_user + 1);
    }
    else
    {
        use_rev1 = xstrdup(vers->vn_user);
    }
}
/* If we already know that the file is being added or removed,
then we don't want to do an actual file comparison here. */
if (empty_file != DIFFDIFFERENT)
{
    retcode = RCScmpfile(vers->srcfile, use_rev1,
        *options ? options : vers->options,
        finfo->file);
    return retcode == 0 ? DIFFSAME : DIFFDIFFERENT;
}
A.15 edit.c

/* Implementation for “cvs edit”, “cvs watch on”, and related commands */

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. */

#include "cvs.h"
#include "getline.h"
#include "watch.h"
#include "edit.h"
#include "fileattr.h"

static int watch_onoff PROTO ((int, char **));
static int setting_default;
static int turning_on;
static int setting_tedit;
static int setting_tunedit;
static int setting_tcommit;

static int onoff_fileprocPROTO ((void *callerdat, struct fileinfo *finfo));

static int onoff_fileproc (callerdat, finfo)
    void *callerdat;
    struct fileinfo *finfo;
{
    fileattr_set (finfo->file, ".watched", turning_on ? "" : NULL);
    return 0;
}

static int onoff_filesdoneprocPROTO ((void *callerdat, int err, char *repository, char *update_dir, List *entries));

static int onoff_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    if (setting_default)
        fileattr_set (NULL, ".watched", turning_on ? "" : NULL);
    return err;
}

static int watch_onoff (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int local = 0;
    int err;

    optind = 0;
    while ((c = getopt (argc, argv, "+lR")) != -1)
    {
        switch (c)
            case 'l':
                local = 1;
            break;
            case 'R':
                local = 0;
            break;
            case '?':
                default:
                    usage (watch_usage);
                break;
    }

    argc -= optind;
    argv += optind;

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();
        ign_setup ();
    }
#endif
if (local)
    send_arg ("-l");
send_string (argc, argv, SEND_EXPAND_WILD);
send_string (argc, argv, local, &SEND_NOCONTENT);
send_string (turning_on ? "watch-on\012" : "watch-off\012", 0);
return get_response_end ()
} sendif /* CLIENT_SUPPORT */

int watch_on (argc, argv)
int argc,
char **argv;
{ turning_on = 1;
  return watch_onoff (argc, argv);
}

int watch_off (argc, argv)
int argc,
char **argv;
{ turning_on = 0;
  return watch_onoff (argc, argv);
}

static int dummy_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
dummy_fileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;
{ /* This is a pretty hideous hack, but the gist of it is that recurse.c
   won't call notify_check unless there is a fileproc, so we can't just
   pass NULL for fileproc. */
  return 0;
}

static int ncheck_fileproc PROTO ((void *callerdat, struct file_info *finfo));

/* Check for and process notifications. Local only. I think that doing
this as a fileproc is the only way to catch all the
cases (e.g. foo/bar.c), even though that means checking over and over
for the same CVSADM_NOTIFY file which we removed the first time we
processed the directory. */

static int
ncheck_fileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;
{ int notify_type;
  char *filename;
  char *v1;
  char *v2;
  char *watches;

  FILE *fp;
  char *line = NULL;
  size_t line_len = 0;

  /* We send notifications even if noexec. I'm not sure which behavior
  is most sensible. */
  fp = CVS_FOPEN (CVSADM_NOTIFY, "r");
  if (fp == NULL)
    { if (lexistence_error (errno))
        error (0, errno, "cannot open \x", CVSADM_NOTIFY);
        return 0;
      }
  while (getline (&line, &line_len, fp) > 0)
    { /* send_string (argc, argv, local, &SEND_HEADER);
    send_string (argc, argv, &SEND_PROJECT);
    send_string (argc, argv, &SEND_LICENSE);
    send_string (argc, argv, &SEND_REVISION);
    send_string (argc, argv, &SEND_DATE);
    send_string (argc, argv, &SEND_AUTHOR);
    send_string (argc, argv, &SEND_DESCRIPTION);
    send_string (argc, argv, &SEND_COMMENT);
    send_string (argc, argv, &SEND_SOURCE);
    send_string (argc, argv, &SEND_END);";*/

    /* This is a pretty hideous hack, but the gist of it is that recurse.c
    won't call notify_check unless there is a fileproc, so we can't just
    pass NULL for fileproc. */
    return 0;
}
notify_type = line[0];
if (notify_type == '\n')
    continue;
filename = line + 1;
cp = strchr (filename, '\t');
if (cp == NULL)
    continue;
    *cp++ = '\n';
    cp = strchr (cp, '\t');
if (cp == NULL)
    continue;
    *cp++ = '*';
    cp = strchr (cp, '\t');
if (cp == NULL)
    continue;
    *cp++ = '\n';
    cp = strchr (cp, '\t');
if (cp == NULL)
    continue;
    *

notify_do (notify_type, filename, getcaller (), val, watches,
find->repository);
}
free (line);

if (error (fp))
    error (0, errno, "cannot read file", CVSADM_NOTIFY);
if (fclose (fp) < 0)
    error (0, errno, "cannot close file", CVSADM_NOTIFY);

if (CVS_UNLINK (CVSADM_NOTIFY) < 0)
    error (0, errno, "cannot remove file", CVSADM_NOTIFY);
return 0;

static int send_notifications PROTO ((int, char **, int));

/* Look through the CVSADM_NOTIFY file and process each item there
accordingly. */
static int
send_notifications (argc, argv, local)
int argc;
char **argv;
int local;
{
    int err = 0;

    if (client_active)
    {
        if (strcmp (command_name, "release") != 0)
        {
            start_server ();
            ignore_setup ();
        }

        err += start_recursion (dummy_fileproc, (FILESDONEPROC) NULL,
(DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL, argc, argv, local,
W_ALOCAL, 0, 0, (char **)NULL, 0);
    }
send_notifications ("noop\n", 0);
if (strcmp (command_name, "release") == 0)
    err += get_server_responses ();
else
    err += get_responses_and_close ();
}
else
sendif
{
/* Local. */
lock_tree_for_write (argc, argv, local, 0);
err += start_recursion (ncheck_fileproc, (FILESDONEPROC) NULL,
    (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL, argc, argv, local,
W_ALOCAL, 0, 0, (char **)NULL, 0);
LockCleanup ();
}
return err;

static int edit_fileproc PROTO ((void *callerdat, struct fileinfo *finfo);

static int edit_fileproc [callerdat, finfo]

(void) *callerdat;
struct fileinfo *finfo;
{
FILE *fp;
time_t now;
char *asctime;
char *basefilename;

if [noexec]
return 0;

/* This is a somewhat screwy way to check for this, because it
doesn't help errors other than the nonexistence of the file
(e.g. permissions problems). It might be better to rearrange
the code so that CVSADM_NOTIFY gets written only after the
various actions succeed (but what if only some of them
succeed). */
if [fisfile (finfo->file)]
{
error (0, 0, "no such file %s; ignored", finfo->filename);
return 0;
}

fp = open (CVSADM_NOTIFY, "a");

(void) time (now);
asctime = asctime (gmtime (now));
asctime[23] = '\0';
fprintf (fp, "%s Time %s
", asctime, finfo->filename,
asctime, hostname, CurDir);
if [setting(calledit)
fprintf (fp, "%E");
if [setting(unedit)
fprintf (fp, "%U");
if [setting(commit)
fprintf (fp, "%C");
fprintf (fp, "%A");

if [fclose (fp) < 0 ]
{
if [finfo->update
error (0, 0, "cannot close %s", CVSADM_NOTIFY);
else
error (0, 0, "cannot close %s", finfo->filename,
CVSADM_NOTIFY);
}

chmod (finfo->file, 1);

/* Now stash the file away in CVSADM so that unedit can revert even if
it can't communicate with the server. We stash away a writable
so that if the user removes the working file, then restores it
with "cos update" (which clears editors but does not update
CVSADM_BASE), then a future "cos edit" can still win. */

/* Could save a system call by only calling makedir_if_needed if
trying to create the output file fails. But copy_to_file isn't
set up to facilitate that. */
makedir_if_needed (CVSADM_BASE);
strcpy (basefilename, xmalloc (10 + sizeof CVSADM_BASE + strlen (finfo->filename)));
strcpy (basefilename, CVSADM_BASE);
strcat (basefilename, ");
strcat (basefilename, finfo->filename);
copy_to_file (finfo->filename, basefilename);
free (basefilename);

Node *node;
node = findnode_in (finfo->entries, finfo->filename);
if [node != NULL]
base->register (finfo, (Entnode *) node->data->version);
}
return 0;

static const char *const edit_usage[] =
{"Usage: %s [-1R] [files...]",
"-1: Local directory only, not recursive\n", "-R: Process directories recursively\n", "-a: Specify what actions for temporary watch, one of:\n", "edit, unedit, commit, all, none\n"}.}
"(Specify the --help global option for a list of other help options)\n", NULL);

int edit (argc, argv)
int argc;
char **argv;
{
    int local = 0;
    int c;
    int err;
    int _omitted;
    if (argc == -1)
        usage (edit_usage);
    _omitted = 1;
    setting_edit = 0;
    setting_unedit = 0;
    setting_commit = 0;
    optind = 0;
    while ((c = getopt (argc, argv, "-lRa:")) != -1)
    {
        switch (c)
        {
        case 'l':
            local = 1;
            break;
        case 'R':
            local = 0;
            break;
        case 'a':
            _omitted = 0;
            if (strcmp (optarg, "edit") == 0)
                setting_edit = 1;
            else if (strcmp (optarg, "unedit") == 0)
                setting_unedit = 1;
            else if (strcmp (optarg, "commit") == 0)
                setting_commit = 1;
            else if (strcmp (optarg, "all") == 0)
            {
                setting_edit = 1;
                setting_unedit = 1;
                setting_commit = 1;
            }
            else if (strcmp (optarg, "none") == 0)
            {
                setting_edit = 0;
                setting_unedit = 0;
                setting_commit = 0;
            }
            else
                usage (edit_usage);
            break;
        case '?':
            default:
            usage (edit_usage);
            break;
        }
    }
    argc -= optind;
    argv += optind;
    if (_omitted)
    {
        setting_edit = 1;
        setting_unedit = 1;
        setting_commit = 1;
    }
    /* No need to relock since we aren't doing anything to the
 repository. */
    err = start_recursion (edit_fileproc, (FILESDONEPROC) NULL,
(DHENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
argc, argv, local, W_LOCAL, 0, 0, (char *)NULL, 0);
    err += send_notifications (argc, argv, local);
    return err;
}

static int unedit_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
unedit_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
    {
FILE *fp;
time_t now;
char *ascnow;
char *basefilename;
if (noexec) return 0;

basefilename = xmalloc((10 + sizeof CVSADM_BASE + strlen (finfo->file)));
strcpy (basefilename, CVSADM_BASE);
strcat (basefilename, "/");
strcat (basefilename, finfo->file);
if (isfile (basefilename))
{
    /* This file apparently was never cvs edit'd (e.g. we are unediting
     a directory where only some of the files were cvs edit'd. */
    free (basefilename);
    return 0;
}

if (xcmp (finfo->file, basefilename) != 0)
{
    printf ("%s has been modified; revert changes? *, finfo->fullname);
    if (!yesno ())
    {
        /* "no" */
        free (basefilename);
        return 0;
    }

    rename (basefilename, finfo->file);
    free (basefilename);
    fp = openfile (CVSADM_NOTIFY, "+*");

    (void) time (now);
    asctime = asctime (gmtime (&now));
    asctime[24] = '0';
    fprintf (fp, "%s\%s %s\%s\%s\n", finfo->file,
            asctime, hostname, CurDir);

    if (fclose (fp) < 0)
    {
        if (finfo->update_dir[strlen ("\")])
            error (0, ernno, "cannot close \", CVSADM_NOTIFY);
        else
            error (0, ernno, "cannot close \%s", finfo->update_dir,
                  CVSADM_NOTIFY);
    }

    /* Now update the revision number in CVS/Entries from CVS/Base.
     The basic idea here is that we are reverting to the revision
     that the user edited. If we wanted "cvs update" to update
     CVS/Base as we go along (so that an unedit could revert to the
     current repository revision), we would need:

     update (or all sendfiles?) (client) needs to send revision in
     new Entry-base request. update (server/local) needs to check
     revision against repository and send new Update-base response
     (like Update-existing in that the file already exists. While
     we are at it, might try to clean up the syntax by having the
     node only in a "Mode" response, not in the Update-base itself). */

    char *baserev;
    Node node;
    Entnode entdata;

    baserev = basename (finfo);
    node = findnode in (finfo->entries, finfo->file);
    /* The case where node is NULL probably should be an error or
     something, but I don't want to think about it too hard right
     now. */
    if (node != NULL)
    {
        entdata = (Entnode *) node->data;
        if (baserev == NULL)
        {
            /* This can only happen if the CVS/Base rev file got
               corrupted. We suspect it might be possible if the
               user interrupts CVS, although I haven't verified
               that. */
            error (0, 0, "%s not mentioned in \", finfo->fullname,
                   CVSADM_BASEREV);
        }
        /* Since we don't know what revision the file derives from,
           keeping it around would be asking for trouble. */
        if (strlen (finfo->file) < 0)
            error (0, ernno, "cannot remove \", finfo->fullname);

        /* This is cheezy, in a sense; why shouldn't we do the
update for the user? However, doing that would require contacting the server, so maybe this is OK. */
error (0, 0, "run update to complete the unedit");
return 0;
}
}
}
}
}
free (baserev);
base deregister (finfo);
}
}
free (baserev);
base deregister (finfo);
}
return 0;
}
}
void mark up to date (file)
char *file;
{
char *base;

/ * The file is up to date, so we better get rid of an out of
date file in CVSADM_BASE. * /
base = xmalloc (strlen (file) + 80);
strcpy (base, CVSADM_BASE);
strcat (base, "/");
strcat (base, file);
if (unlink (base) < 0 && !existence_error (errno))
error (0, errno, "cannot remove %s", file);
free (base);
}
}
void editor set (filename, editor, val)
char *filename;
char *editor;
char *val;
{ char *edlist;
char *newlist;
edlist = fileattr get0 (filename, "._editors");
newlist = fileattr modify (edlist, editor, val, "+", ",");
/* If the attributes is unchanged, don’t rewrite the attribute file. */
630 if (!((edlist == NULL && newlist == NULL)
  || (edlist != NULL
    && newlist != NULL
    && strcmp(edlist, newlist) == 0)))
  fileattr_set (filename, "_editors", newlist);
if (edlist != NULL)
  free(edlist);
if (newlist != NULL)
  free(newlist);
}

struct notify_proc_args {
  /* What kind of notification, “edit”, “tedit”, etc. */
  char *type;
  /* User who is running the command which causes notification. */
  char *who;
  /* User to be notified. */
  char *notifyee;
  /* File. */
  char *file;
};

/* Pass as a static until we get around to fixing Parse_Info to pass along
  a void * where we can stash it. */
static struct notify_proc_args *notify_arg;

static int notify_proc PROTO ((char *repository, char *filter));

static int notify_proc (repository, filter)
660 char *repository;
  char *filter;
{
  FILE *pipefp;
  char *prog;
  char *expanded_prog;
  char *q;
  char *xrepos;

  struct notify_proc_args *args = notify_arg;

  srepos = Short_Repository(repository);
  prog = xmalloc (strlen (filter) + strlen (args->notifyee) + 1);
  /* Copy FILTER to PROG, replacing the first occurrence of %s with
     the notifyee. We only allocated enough memory for one %s, and I doubt
     there is a need for more. */
  for (p = filter, q = prog; p != ‘\0’; ++p)
  {
    if (p[0] == ‘%’)
      {
        if (p[1] == ‘s’)
          {
            strcpy (q, args->notifyee);
            q += strlen (q);
            strcpy (q, p + 2);
            q += strlen (q);
            break;
          }
        else
          continue;
      }
    *q++ = *p;
  }
  *q = ‘\0’;

  /* FIXME: why are we calling expand_prog? Didn’t we already
     expand it in Parse_Info, before passing it to notify_proc? */
  expanded_prog = expandpath (prog, "notify", 0);
  if (expanded_prog)
  {
    free (prog);
    return 1;
  }
  pipefp = run_open (expanded_prog, "a");
  if (pipefp == NULL)
  {
    error (0, errno, "cannot write entry to notify filter: %a", prog);
    free (prog);
    free (expanded_prog);
    return 1;
  }
  fprintf (pipefp, "%s %s
", repository, expanded_prog);
  fprintf (pipefp, "Triggered %s watch on %a", args->type, repository);
  fprintf (pipefp, "By %a", args->who);

  /* Lots more potentially useful information we could add here; see
     logfile_write for inspiration. */
free (prog);
free (expanded_prog);
return (pclose (pipefp));
}

void
notify_do (type, filename, who, val, watches, repository)

int type;
char *filename;
char *who;
char *val;
char *watches;
char *repository;
{
static struct addremove_args blank;
struct addremove_args args;
char *watchers;
char *p;
char *endp;
char *nextp;

/* Initialize fields to 0, NULL, or 0.0. */
args = blank;
switch (type)
{
    case 'E':
        editor_set (filename, who, val);
        break;
    case 'U':
    case 'C':
        editor_set (filename, who, NULL);
        break;
    default:
        return;
}

watchers = fileattr_get0 (filename, "_watchers");
p = watchers;
while (p != NULL)
{
    char *q;
    char *endq;
    char *nextq;
    char *notif;
    endp = strchr (p, '>');
    if (endp == NULL)
        break;
    nextp = strchr (p, ',');
    if ((size_t)(endp - p) == strlen (who) && strncmp (who, p, endp - p) == 0)
    {
        /* Don't notify user of their own changes. Would perhaps
         * be better to check whether it is the same working
         * directory, not the same user, but that is hairy. */
        p = nextp == NULL ? nextp : nextp + 1;
        continue;
    }
    else
        nextq = endq + 1;
    /* Now we point q at a string which looks like
     * "edit-unedit-commit,"... and walk down it. */
    q = endp + 1;
    notif = NULL;
    while (q != NULL)
    {
        endq = strchr (q, '*');
        if (endq == NULL || (nextp != NULL && endq > nextp))
        {
            if (nextp == NULL)
                endq = q + strlen (q);
            else
                endq = nextp;
            nextq = NULL;
        }
   /* If there is a temporary and a regular watch, send a single
    * notification, for the regular watch. */
    if (type == 'E' && endq - q == 4 && strncmp (*edit*, q, 4) == 0)
    {
        notif = *edit*;
    }
   else if (type == 'U' && endq - q == 6 && strncmp (*unedit*, q, 6) == 0)
    {
        notif = *unedit*;
    }
   else if (type == 'C')
810
    
    820
    if (notif != NULL)
    {
        struct notify_proc_args args;
        size_t len = endp - p;
        char *usersname = NULL;
        char *line = NULL;
        size_t line_len = 0;
        args.notify = NULL;
        usersname = xmalloc(strlen(CVSroot_directory) + sizeof CVSROOTADM
            + sizeof CVSROOTADM_USERS
            + 20);
        strcpy(usersname, CVSroot_directory);
        strcat(usersname, "@");
        strcat(usersname, CVSROOTADM);
        strcat(usersname, ";");
        strcat(usersname, CVSROOTADM_USERS);
        if (fp == NULL || fexists_error(errno))
            error(0, errno, "cannot read %s", usersname);
        if (fp != NULL)
        {  
            while (getline (&line, &line_len, fp) >= 0)
            {  
                if (strn cmp(line, p, len) == 0)
                    endp = p + 1;
            
                if (error(fp))
                    error(0, errno, "cannot read %s", usersname);
                if (fclose(fp) < 0)
                    error(0, errno, "cannot close %s", usersname);
            
            free(usersname);
            free(line);
        
344
        if (args.notify == NULL)
        {  
            args.notify = xmalloc(endp - p + 1);
            strcpy(args.notify, p, endp - p);
            args.notify[strlen(endp - p) + 1] = "\0";
        
        notify_args = &args;
        argstype = notif;
        args.who = who;
        args.file = filename;
        
425
        (void) Parse_Info (CVSROOTADM_NOTIFY, repository, notify_proc, 1);
        free(args.notify);
    
434
    p = nextp;
    }
if (watchers != NULL)
  free (watchers);

switch (type)
{
  case 'E':
    if (*watches == 'E')
      {
        args.add_c_edit = 1;
        ++watches;
      }
    if (*watches == 'U')
      {
        args.add_c_update = 1;
        ++watches;
      }
    if (*watches == 'C')
      {
        args.add_c_commit = 1;
      }
    watch_modify_watchers (filename, &args);
    break;
  case 'V':
  case 'C':
    args.remove_temp = 1;
    watch_modify_watchers (filename, &args);
    break;
}
#endif

#define CLIENT_SUPPORT

/* Check and send notifications. This is only for the client. */
void notify_check (repository, update_dir)
char *repository,
char *update_dir;
{
  FILE *fp;
  char *line = NULL;
  size_t line_len = 0;

  if (!server_started)
    /* We are in the midst of a command which is not to talk to
     the server (e.g. the first phase of a cvs edit). Just chill
     out, we'll catch the notifications on the flip side. */
    return;

  /* We send notifications even if noexec. I'm not sure which behavior
   is most sensible. */
  fp = CVS_FOPEN (CVSADM_NOTIFY, "r");

  while (getline (&line, &line_len, fp) > 0)
    {
      int notif_type;
      char *filename;
      char *val;
      char *cp;

      notif_type = line[0];
      if (notif_type == 'V')
        continue;
      filename = line + 1;
      cp = strchr (filename, '\t');
      if (cp == NULL)
        continue;
      *cp++ = '\n';
      val = cp;

      client_notify (repository, update_dir, filename, notif_type, val);
    }
  if (line)
    free (line);
  if (error (fp))
    error (1, errno, "cannot read %s", CVSADM_NOTIFY);
  if (fclose (fp) < 0)
    error (1, errno, "cannot close %s", CVSADM_NOTIFY);

  /* Leave the CVSADM_NOTIFY file there, until the server tells us it
   has dealt with it. */
}
#endif
#endif

static const char *const editors_usage[] =

990 { "Usage: X \ [-1h] \ [files ...] \n",  "\t-1\nProcess this directory only (not recursive). \n",  "\t-R\nProcess directories recursively. \n",  "(Specify the --help global option for a list of other help options) \n",  NULL };  

static int editors_fileproc PROTO ((void *callerdat, struct file *info *finfo));  

static int  
1000 editors_fileproc (callerdat, finfo)  
void *callerdat;  
struct file *info *finfo;  
{  
  char *them;  
  char *p;  
  them = fileattrget0 (finfo->file, "_editors");  
  if (them == NULL)  
    return 0;  
1010 fputs (finfo->fullname, stdout);  
  p = them;  
  while (1)  
  {  
    putc ("\t", stdout);  
    while (*p != '>' && *p != '\0')  
      putc (*p++, stdout);  
    if (*p == '\0')  
      /* Only happens if attribute is misformed. */  
      putc ("\n", stdout);  
      break;  
    ++p;  
    putc ("\t", stdout);  
    while (1)  
    {  
      while (*p != ' ' && *p != '\' && *p != '\0')  
        putc (*p++, stdout);  
      if (*p == '\0')  
        {  
          putc ("\n", stdout);  
          goto out;  
        }  
      if (*p == ' ')  
        {  
          ++p;  
          break;  
        }  
      ++p;  
      putc ("\t", stdout);  
    }  
    putc ("\n", stdout);  
  }  
  out:;  
  return 0;  
}  

1050 int  
int editors (argc, argv)  
int argc;  
char *argv;  
{  
  int local = 0;  
  int c;  
  if (argc == -1)  
    usage (editors_usage);  
1060 optind = 0;  
  while ((c = getopt (argc, argv, "-lR")) != -1)  
  {  
    switch (c)  
    {  
      case 'l':  
        local = 1;  
        break;  
      case 'R':  
        local = 0;  
        break;  
      default:  
        usage (editors_usage);  
        break;  
    }  
  }  
  argc == optind;
argv += optind;

ifdef CLIENT_SUPPORT
if (client_active)
{
    start_server ();
    ign_setup ();

    if (local)
        send_arg ("-l");
    send_file_names (argc, argv, SEND_EXPAND_WILD);
else
    send_files (argc, argv, 0, SEND_NO_CONTENTS);
endif /* CLIENT_SUPPORT */

return start_recursion (editors_fileproc, (FILESDONEPROC) NULL,
                       (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                       argc, argv, local, W_LOCAL, 0, 1, (char *)NULL, 0);

endif /* CLIENT_SUPPORT */
/* Interface to “cvs edit”, “cvs watch on”, and related features

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

extern int watch_on PROTO ((int argc, char **argv));
extern int watch_off PROTO ((int argc, char **argv));

#ifndef CLIENT_SUPPORT
/* Check to see if any notifications are sitting around in need of being
sent. These are the notifications stored in CVSADM_NOTIFY (edit,unedit);
commit calls notify_do directly. */
extern void notify_check PROTO ((char *repository, char *update_dir));
#endif /* CLIENT_SUPPORT */

/* Issue a notification for file FILENAME. TYPE is ‘E’ for edit, ‘U’
for unedit, and ‘C’ for commit. WHO is the user currently running.
For TYPE ‘E’, VAL is the time+host+directory data which goes in
editors, and WATCHES is zero or more of E,U,C, in that order, to specify
what kinds of temporary watches to set. */
extern void notify_do PROTO ((int type, char *filename, char *who,
char *val, char *watches, char *repository));

/* Set attributes to reflect the fact that EDITOR is editing FILENAME.
VAL is time+host+directory, or NULL if we are to say that EDITOR is
not editing FILENAME. */
extern void editor_set PROTO ((char *filename, char *editor, char *val));

/* Take note of the fact that FILE is up to date (this munges CVS/Base;
processing of CVS/Entries is done separately). */
extern void mark_up_to_date PROTO ((char *file));
A.17 entries.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.

 * Entries file to Files file

 * Creates the file Files containing the names that comprise the project, from
 * the Entries file.
 */

#include "cvs.h"
#include "getline.h"

static Node *AddEntryNode PROTO((List * list, Entnode *entnode));
static Entnode *(getentent PROTO((FILE *, char *, int *)));
static void *Construct an Entnode PROTO((entnode *entnode));
static void *Destruct an Entnode PROTO((entnode *entnode));
static int *nullentent PROTO((int, const char *, const char *));

/* Construct an Entnode */

static Entnode *Entnode_Create PROTO ((entnode *entnode, enum type, const char *,
                                      const char *, const char *,
                                      const char *, const char *,
                                      const char *, const char *));

Entnode_Create(type, user, vn, ts, options, tag, date, ts_conflict, root, repository)
  enum type;  
  const char *user;  
  const char *vn;  
  const char *ts;  
  const char *options;  
  const char *tag;  
  const char *date;  
  const char *ts_conflict;  
  const char *root;  
  const char *repository;

  {  
      Entnode *ent;

      if (repository == NULL) {
          fprintf(stderr, "Entnode_Create, repository is NULL!");
      }

    /* Note that timestamp and options must be non-NULL */
    ent = (Entnode *) xmalloc(sizeof (Entnode));  
    ent->type = type;  
    ent->user = xstrdup(user);  
    ent->version = xstrdup(vn);  
    ent->timestamp = xstrdup(ts ? ts : "");  
    ent->options = xstrdup(options ? options : "");  
    ent->tag = xstrdup(tag);  
    ent->date = xstrdup(date);  
    ent->ts_conflict = xstrdup(ts_conflict);  
    ent->root = xstrdup(root);  
    ent->repository = xstrdup(repository);  

    return ent;
    }

/* Destruct an Entnode */

static void Entnode_Destroy PROTO ((Entnode *ent));

Entnode_Destroy (ent)
    Entnode *ent;

    {  
        free (ent->user);  
        free (ent->version);  
        free (ent->timestamp);  
        free (ent->options);  
        if (ent->tag)  
            free (ent->tag);  
        if (ent->date)  
            free (ent->date);  
        if (ent->ts_conflict)  
            free (ent->ts_conflict);  

        free (ent);
free (ent->conflict);
free (ent);
}

/* Write out the line associated with a node of an entries file */
static int write_ent ProcPROTO ((Node *, void *));
static int write_entProc (node, closure)
Node *node;
void *closure;
{
  Entnode *entnode;
  entnode = (Entnode *) node->data;
  if (closure != NULL & entnode->type != ENT_FILE)
    *(int *) closure = 1;
  if (fputentent (entfile, entnode))
    error (1, errno, "cannot write %s", entfilename);
  return (0);
}

/* write out the current entries file given a list, making a backup copy */
static void write_entries (list) List *list;
{
  int sawdir;
  sawdir = 0;
  /* open the new one and walk the list writing entries */
  entfilename = CVSADM_ENTBAK;
  entfile = CVS_FOPEN (entfilename, "w+");
  if (entfile == NULL)
    {
      /* Make this a warning, not an error. For example, one user might
       * have checked out a working directory which, for whatever reason,
       * contains an Entries.Log file. A second user, without write access
       * to that working directory, might want to do a "cvs log". The
       * problem rewriting Entries shouldn't affect the ability of "cvs log"
       * to work, although the warning is probably a good idea so that
       * whether Entries gets rewritten is not an inexcusable process. */
      /* FIXME: should be including update_dir in message. */
      error (0, errno, "cannot rewrite %s", entfilename);
    }
  if (fclose (entfile) == EOF)
    error (1, errno, "error closing %s", entfilename);
  /* Now just return. We leave the Entries.Log file around. As far
   * as I know, there is never any data lying around in 'list' that
   * is not in Entries.Log at this time (if there is an error writing
   * Entries.Log that is a separate problem). */
  return;
}
(void) walklist (list, write_ent Proc, (void *) &sawdir);
if (! sawdir)
  {
    struct stickydirtag *sdtp;
    /* We didn't write out any directories. Check the list
     * private data to see whether subdirectory information is
     * known. If it is, we need to write out an empty D line. */
    sdtp = (struct stickydirtag *) list->data;
    if (sdtp != NULL & sdtp->subdirs)
      if (fprintf (entfile, "%s", "D") < 0)
        error (1, errno, "cannot write %s", entfilename);
  }
  if (fclose (entfile) == EOF)
    error (1, errno, "error closing %s", entfilename);
  /* now, atomically (on systems that support it) rename it */
  rename_file (entfilename, CVSADM_ENT);
  /* now, remove the log file */
  unlink_file (CVSADM_ENTLOG);
}

/* Removes the argument file from the Entries file if necessary. */
void ScratchEntry (list, fname)
List *list;
char *fname;
{  
    Node *node;
    
    if (trace)
server SUPPORT
    {  
        (void) fprintf(stderr, "Scratch_Entry(node)\n");
        (server active) ? 'S' : ' ', node);
    } else
    {  
        (void) fprintf(stderr, "Scratch_Entry(node), node);
    }
}  

if (inode)
    {  
        entfile = CVSADM_ENTLOG;
        entfile = openfile (entfile, "a");

        if (fprintf (entfile, "%s")  < 0)
            error (1, errno, "cannot write %s", entfile);

        writeProc (node, NULL);
        if (fclose (entfile) == EOF)
            error (1, errno, "error closing %s", entfile);
    }  

    delnode (node);  /* delete the node */
}  

if (server active)
server scratch6 (fname);

}  

/* hashlookup to see if it is there */
if ((node = findnode (list, fname)) != NULL)
    {  
        if (noexec)
            {  
                entfile = CVSADM_ENTLOG;
                entfile = openfile (entfile, "a");

                if (fprintf (entfile, "%s")  < 0)
                    error (1, errno, "cannot write %s", entfile);

                writeProc (node, NULL);
                if (fclose (entfile) == EOF)
                    error (1, errno, "error closing %s", entfile);
            }  

            delnode (node);  /* delete the node */
    }  

if (server active)
server register (fname, vn, ts, options, tag, date, ts conflict, root, repository)
    {  
        char *fname;
        char *vn;
        char *ts;
        char *options;
        char *tag;
        char *date;
        char *ts conflict;
        char *root;
        char *repository;

        Entnode *entnode;
        Node *node;

        if (trace)
            {  
                (void) fprintf(stderr, "Register(node, %s, %s, %s, %s, %s, %s)\n");

                if (server active) ? 'S' : ' ', fname, vn, ts, options, tag, date, ts conflict, root, repository);
            }  

server SUPPORT  
if (server active)
    {  
        server register (fname, vn, ts, options, tag, date, ts conflict, root, repository);
    }
}

if (trace)
    {  
        (void) fprintf(stderr, "Register(node, %s, %s, %s, %s, %s, %s)\n");

        if (server active) ? 'S' : ' ', fname, vn, ts, options, tag, date, ts conflict, root, repository);
    }  

entnode = EntnodeCreate (ENT FILE, fname, vn, ts, options, tag, date, ts conflict, root, repository);
node = AddEntryNode (list, entnode);
if (noexec)
    {  
        entfile = CVSADM ENTLOG;
        entfile = openfile (entfile, "a");

        if (fprintf (entfile, "%s")  < 0)
```c
    error (l, errno, "cannot write %s", entfilename);
    write_entProc (node, NULL);
    if (fclose (entfile) == EOF)
        error (l, errno, "error closing %s", entfilename);
}

/* Node delete procedure for list-private sticky dir/date info */
static void freesdt (Node *p)
{
    struct stickydirtag *sdtp;
    sdtp = (struct stickydirtag *)p->data;
    if (sdtp->tag)
        free (sdtp->tag);
    if (sdtp->date)
        free (sdtp->date);
    free ((char *)sdtp);
}

/* Return the next real Entriesline. On end of file, returns NULL. 
   On error, prints an error message and returns NULL. */
static Entnode *fgetentent (FILE *fpin, char *cmd, int *sawdir)
{
    Entnode *ent;
    char *line;
    size_t line_chars_allocated;
    register char *cp;
    enum ent_type type;
    char *tag, *date, *ts_conflict;
    char *root;
    char *repository;
    int line_length;
    line = NULL;
    line_chars_allocated = 0;
    ent = NULL;
    while ((line_length = getline (&line, &line_chars_allocated, fpin)) > 0)
    {
        l = line;
        /* If CMD is not NULL, we are reading an Entries.Log file. 
           Each line in the Entries.Log file starts with a single 
           character command followed by a space. For backward 
           compatibility, the absence of a space indicates an add 
           command. */
        if (cmd != NULL)
            { /* 
               * If CMD is not NULL, we are reading an Entries.Log file. 
                  Each line in the Entries.Log file starts with a single 
                  character command followed by a space. For backward 
                  compatibility, the absence of a space indicates an add 
                  command. */
                if (l[1] == ' ')
                    *cmd = 'A';
                else
                    {
                        *cmd = l[0];
                        l += 2;
                    }
                type = ENT_FILE;
            }
        if (l[0] == 'D')
            { /* An empty D line is permitted; it is a signal that this 
                   Entries file lists all known subdirectories. */
                type = ENT_SUBDIR;
                *sawdir = 1;
                ++l;
            }
        if (l[1] == ' ')
            continue;
        user = l + 1;
        if ((cp = strchr (user, ' ')) == NULL)
            continue;
        *cp++ = '\0';
        vn = cp;
        if ((cp = strchr (vn, '/')) == NULL)
```
continue;
    *cp++ = '\0';
    ts = cp;
    if ((cp = strchr (ts, '\')) == NULL)
        continue;
    *cp++ = '\0';
    options = cp;
    if ((cp = strchr (options, '\')) == NULL)
        continue;
    *cp++ = '\0';
    tag_asdate = cp;
    if ((cp = strchr (tag_asdate, '\')) == NULL)
        continue;
    *cp++ = '\0';
    root = cp;
    if ((cp = strchr (root, '\')) == NULL)
        continue;
    if ((cp = strchr (cp, '\')) == NULL)
        continue;
    if ((cp = strchr (cp, '\')) == NULL)
        continue;
    *cp++ = '\0';
    repository = cp;
    if ((cp = strchr (repository, '\')) == NULL)
        continue;
    *cp = '\0';
    tag = (char *) NULL;
    date = (char *) NULL;
    if (*tag_asdate == 'T')
        tag = tag_asdate + 1;
    else if (*tag_asdate == 'Y')
        date = tag_asdate + 1;
    if ((ts_conflict = strchr (ts, '\')))
        ts_conflict++ = '\0';
    /*
     * XXX - Convert timestamp from old format to new format.
     */
    /*
     * If the timestamp doesn't match the file's current
     * mtime, we'd have to generate a string that doesn't
     * match anyway, so cheat and base it on the existing
     * string; it doesn't have to match the same mod time.
     */
    /*
     * For an unmodified file, write the correct timestamp.
     */
    {
        struct stat sb;
        if (strlen (ts) > 30 && CVS_STAT (user, &sb) == 0)
            {
                char *c = ctime (&sb.st_utime);
                if (!strncmp (ts + 25, c, 24))
                    ts = timestamp (user);
                else
                    {
                        ts += 24;
                        ts[0] = '\*';
                    }
            }
        ent = EntnodeCreate (type, user, vn, ts, options, tag, date,
            ts_conflict, root, repository);
        break;
    }

    if (strlen(line) < 0 && (errno == ENOENT))
        error (0, errno, "cannot read entries file");
    free (line);
    return ent;
}

static int
fputentent(fp, p)
    FILE *fp;
    
    Entnode *p;
{
    switch (p->type)
        
    case ENTFILE:
        break;
    case ENT_SUBDIR:
        if (fputstat (fp, "*") < 0)
            return 1;
break;

if (fprintf (fp, "/*/%s/*/", p->user, p->version, p->timestamp) < 0)
  return 1;
if (p->conflict)
  return 1;
if (fprintf (fp, "%s", p->conflict) < 0)
  return 1;
if (fprintf (fp, "/*/", p->options) < 0)
  return 1;

if (p->tag)
  return 1;
else if (p->date)
  return 1;
/* kludge around the fact that local repository has no ::: */
if (p->root[0] == ' ') {
  if (fprintf (fp, "/::%s", p->root) < 0) {
    return 1;
  }
}
else {
  if (fprintf (fp, "/*/", p->root) < 0)
    return 1;
}
return 0;

/* Read the entries file into a list, hashing on the file name. UPDATE_DIR is the name of the current directory, for use in error messages, or NULL if not known (that is, none has gotten around to updating the caller to pass in the information). */

List *Entries_Open (aflag, update_dir)
  int aflag;
  char *update_dir;
  {  
    List *entries;
    struct stickydirtag *sdtp = NULL;
    Entnode *ent;
    char *dirtag, *dirdate;
    int dirnonbranch;
    int do_rewrite = 0;
    FILE *fpin;
    int sawdir;
    /* get a fresh list... */
    entries = getlist();
    /* /* Parse the CVS/Tag file, to get any default tag/date settings. Use * list-private storage to tuck them away for Version_T2/. */
    /* ParseTag (&dirtag, &dirdate, &dirnonbranch);
    if (aflag || dirtag || dirdate)
      {
        sdtp = (struct stickydirtag *) malloc (sizeof (sdtp));
        memset (char *) sdtp, 0, sizeof (sdtp);
        sdtp->aflag = aflag;
        sdtp->dirtag = xstrdup (dirtag);
        sdtp->dirdate = xstrdup (dirdate);
        sdtp->dirnonbranch = dirnonbranch;
        /* feed it into the list-private area */
        entries->list->data = (char *) sdtp;
        entries->list->delproc = freesdt;
      }
    sawdir = 0;
    fpin = CVS_FOPEN (CVSADM_ENT, "r");
    if (fpin == NULL)
      {
        if (update_dir != NULL)
          error (0, 0, "in directory %s", update_dir);
        error (0, errno, "cannot open %s for reading", CVSADM_ENT);
      }
    else {
      }
while ((ent = fgetsentent (fpin, (char *) NULL, &sawdir)) != NULL)
{
    (void) AddEntryNode (entries, ent);
}
fclose (fpin);
}

fpin = CVS_FOPEN (CVSADM_ENTLOG, *r*);
if (fpin != NULL)
{
    char cmd;
    Node *node;
    while (((ent = fgetsentent (fpin, &cmd, &sawdir)) != NULL))
    {
        switch (cmd)
        {
        case 'A':
            (void) AddEntryNode (entries, ent);
            break;
        case 'R':
            node = findnode_fn (entries, ent->>user);
            if (node != NULL)
                delnode (node);
            Entnode_Destroy (ent);
            break;
        default:
            /* Ignore unrecognized commands. */
            break;
        }
    }
    do_rewrite = 1;
    fclose (fpin);
}

/* Update the list private data to indicate whether subdirectory
information is known. Nonexistent list private data is taken
to mean that it is known. */
if (sdtp != NULL)
    sdtp->subdirs = sawdir;
else if (l sawdir)
{
    sdtp = (struct stickydirtag *) xmalloc (sizeof (sdtp));
    memset ((char *) sdtp, 0, sizeof (sdtp));
    sdtp->subdirs = 0;
    entries->list->>data = (char *) sdtp;
    entries->list->>delproc = freesdt;
}

if (do_rewrite && !noexec)
    write_entries (entries);
/* clean up and return */
if (dirtag)
    free (dirtag);
if (dirdate)
    free (dirdate);
return (entries);
}

void
Entries_Close(list)
List *list;
{
    if (list)
    {
        if (noexec)
            if (isfile (CVSADM_ENTLOG))
                write_entries (list);
        dellist(&list);
    }
}

/*
* Free up the memory associated with the data section of an ENTRIES type
* node
*/

static void
Entries_delproc (node)
Node *node;
{
    Entnode *p;
    p = (Entnode *) node->>data;
    Entnode_Destroy (p);
}
/*
 * Get an Entries file list node, initialize it, and add it to the specified
 * list
 */
static Node *
AddEntryNode (list, entdata)
List *list;
Entnode *entdata;
{
    Node *p;

    /* was it already there? */
    if (((p = findnode(fn (list, entdata->user)) != NULL)
    {
        /* take it out */
        delnode (p);
    }
    /* get a node and fill in the regular stuff */
    p = getnode ();
    p->type = ENTRIES;
    p->delproc = Entries_delproc;
    /* this one gets a key of the name for hashing */
    /* FIXME This results in duplicated data — the hash package shouldn't
     * assume that the key is dynamically allocated. The user's free proc
     * should be responsible for freeing the key. */
    p->key = xstrdup (entdata->user);
    p->data = (char *) entdata;
    /* put the node into the list */
    addnode (list, p);
    return (p);
}

/* Write out/Clear the CVS/Tag file. */
void
WriteTag (dir, tag, date, nonbranch, update, repository)
char *dir;
char *tag;
char *date;
int nonbranch;
char *update;
char *repository;
{
    FILE *fout;
    char *tmp;

    if (noexec)
        return;

    tmp = xmalloc ((dir ? strlen (dir) : 0)
        + sizeof (CVSADM_TAG)
        + 10);
    if (dir == NULL)
        strcpy (tmp, CVSADM_TAG);
    else
        (void) sprintf (tmp, "%s/%s", dir, CVSADM_TAG);

    if (tag || date)
    {
        fout = open (tmp, "w+");
        if (tag)
        {
            if (nonbranch)
            {
                if (fprintf (fout, "%s\n", tag) < 0)
                    error (1, errno, "write to %s failed", tmp);
            }
            else
            {
                if (fprintf (fout, "%s\n", date) < 0)
                    error (1, errno, "write to %s failed", tmp);
            }
        }
        else
        {
            if (fclose (fout) == EOF)
                error (1, errno, "cannot close %s", tmp);
        }
        if (unlink (tmp) < 0 && existence (errno))
            error (1, errno, "cannot remove %s", tmp);
        free (tmp);
    }
ifdef 

/* Parse the CVS/Tag file for the current directory. */ 

If it contains a date, sets *DATEP to the date in a newly malloc'd 
string, *TAGP to NULL, and *NONBRANCHP to an unspecified value. 

If it contains a branch tag, sets *TAGP to the tag in a newly 
malloc'd string, *NONBRANCHP to 0, and *DATEP to NULL. 

If it contains a nonbranch tag, sets *TAGP to the tag in a newly 
malloc'd string, *NONBRANCHP to 1, and *DATEP to NULL. 

If it does not exist, or contains something unrecognized by this 
version of CVS, set *DATEP and *TAGP to NULL and *NONBRANCHP to 
an unspecified value. 

If there is an error, print an error message, set *DATEP and *TAGP 
to NULL, and return. */ 

void 

ParseTag (tagp, datep, nonbranchp) 

char **tagp; 

char **datep; 

int *nonbranchp; 

{ 

FILE *fp; 

if (tagp) 

*tagp = (char *) NULL; 

if (datep) 

*datep = (char *) NULL; 

/* Always store a value here, even in the 'D' case where the value 
is unspecified. Shuts up tools which check for references to 
uninitialized memory. */ 

if (nonbranchp != NULL) 

*nonbranchp = 0; 

fp = CVS=fopen (CVSADM_TAG, "r"); 

if (fp) 

{ 

char *line; 

int line_length; 

size_t line_chars_allocated; 

line = NULL; 

line_chars_allocated = 0; 

if ((line_length = getline (&line, &line_chars_allocated, fp)) > 0) 

{ 

/* Remove any trailing newline. */ 

if (line[line_length - 1] == '\n') 

line[--line_length] = '\0'; 

switch (*line) 

{ 

case 'T': 

if (tagp != NULL) 

*tagp = xstrdup (line + 1); 

break; 

case 'D': 

if (datep != NULL) 

$datep = xstrdup (line + 1); 

break; 

case 'B': 

if (tagp != NULL) 

*tagp = xstrdup (line + 1); 

if (nonbranchp != NULL) 

*nonbranchp = 1; 

break; 

default: 

/* Silently ignore it; it may have been 
written by a future version of CVS which extends the 
syntax. */ 

break; 

} 

if (line_length < 0) 

{ 

/* PIXME-update-dir: should include update_dir in messages. */ 

if (feof (fp)) 

error (0, 0, "cannot read file; and of file", CVSADM_TAG); 

else 

error (0, errno, "cannot read file", CVSADM_TAG); 

} 

if (fclose (fp) < 0) 

/* PIXME-update-dir: should include update_dir in message. */
error (0, errno, "cannot close %s", CVSADM_TAG);

free (line);
}
}
} else if ( existerror (errno))
  /* FIXME: update-dir: should include update-dir in message. */
  error (0, errno, "cannot close %s", CVSADM_TAG);

#endif

/* This is called if all subdirectory information is known, but there
 * aren’t any subdirectories. It records that fact in the list
 * private data.
 */
void
Subdirs_Known (entries)
List *entries;
{
  struct stickydirtag *sdtp;

  /* If there is no list private data, that means that the
   * subdirectory information is known.
   */
  sdtp = (struct stickydirtag *) entries->list->data;
  if ( sdtp != NULL && !sdtp->subdirs)
    {
      FILE *fp;
      sdtp->subdirs = 1;
      if ( !noexec) {
        /* Create Entries.Log so that Entries.Close will do something. */
        fp = CVS=fopen (CVSADM_ENTLOG, "a");
        if ( fp == NULL)
          {
            int save_errno = errno;
            /* As in subdir_record, just silently skip the whole thing
             * if there is no CVSADM directory. */
            if ( !isdir (CVSADM))
              return;

            error (1, save_errno, "cannot open %s", entfilename);
          }
        else
          {
            if ( fclose(fp) == EOF)
              error (1, errno, "cannot close %s", CVSADM_ENTLOG);
          }
      }
    }

  /* Record subdirectory information. */
  static Entnode *
  subdir_record (cmd, parent, dir)
    int cmd;
    const char *parent;
    const char *dir;
  Entnode *entnode;

  /* None of the information associated with a directory is
   * currently meaningful. */
  entnode = Entnode_Create (ENT_SUBDIR, dir, **
    (char *) NULL, (char *) NULL,
    (char *) NULL, CVSroot_original, **);

  if ( !noexec) {
    if (parent == NULL)
      entfilename = CVSADM_ENTLOG;
    else
      {
        entfilename = xmalloc (strlen (parent) +
          sizeof CVSADM_ENTLOG + 10);
        sprintf (entfilename, "%s/%s", parent, CVSADM_ENTLOG);
      }
    entfile = CVS_FOPEN (entfilename, "a");
    if (entfile == NULL)
      {
        int save_errno = errno;
        /* It is not an error if there is no CVS administration
         * directory. Permitting this case simplifies some
         * calling code. */
        if (parent == NULL)
          }
    else
      {
        fwrite (line, 1, strlen (line), entfile);
        fclose (entfile);
        if (parent == NULL)
          }
void SubdirRegister (List *entries, char *parent, char *dir)
{
    Entnode *entnode;
    entnode = subdirrecord ('A', parent, dir);
    if (entries != NULL && (parent == NULL || strcmp (parent, ".") == 0))
        AddEntryNode (entries, entnode);
    else
        EntnodeDestroy (entnode);
}

void SubdirDeregister (List *entries, char *parent, char *dir)
{
    Entnode *entnode;
    entnode = subdirrecord ('R', parent, dir);
    if (entries != NULL && (parent == NULL || strcmp (parent, "..") == 0))
        DelEntryNode (entries, entnode);
    else
        EntnodeDestroy (entnode);
}

/*
 * Record the addition of a new subdirectory DIR in PARENT. PARENT
 * may be NULL, which means the current directory. ENTRIES is the
 * current entries list; it may be NULL, which means that it need not
 * be updated.
 */
void SubdirRegister (entries, parent, dir)
{
    Entnode *entnode;
    entnode = subdirrecord ('A', parent, dir);
    if (entries != NULL && (parent == NULL || strcmp (parent, ".") == 0))
        AddEntryNode (entries, entnode);
    else
        EntnodeDestroy (entnode);
}

/*
 * Record the removal of a subdirectory. The arguments are the same
 * as for SubdirRegister.
 */
void SubdirDeregister (entries, parent, dir)
{
    Entnode *entnode;
    entnode = subdirrecord ('A', parent, dir);
    if (entries != NULL && (parent == NULL || strcmp (parent, ".") == 0))
        DelEntryNode (entries, entnode);
    else
        EntnodeDestroy (entnode);
}
/ * OK, the following base_w code tracks the revisions of the files in
CVS/Base. We do this in a file CVS/Base. Separate from
CVS/Entries because it needs to go in separate data structures
anyway (the name in Entries must be unique), so this seemed
cleaner. The business of rewriting the whole file in
base_reregister and base_reregister is the kind of thing we used to
do for Entries and which turned out to be slow, which is why there
is now the Entries.Log machinery. So maybe from that point of
view it is a mistake to do this separately from Entries, I dunno.

We also need something analogous for:

1. CVS/Template (so we can update the Template file automatically
   without the user needing to check out a new working directory).
   Updating would probably print a message (that part might be
   optional, although probably it should be visible because not all
cvs commands would make the update happen and so it is a
user-visible behavior). Constructing version number for template
is a bit hairy (base it on the timestamp on the server? Or see if
the template is in checkoutlist and if yes use its versioning and
if no don't version it?) . . . .

2. cvsgnore (need to keep a copy in the working directory to do
"cvs release" on the client side; see comment at src/release.c
(release). Would also allow us to stop needing Questionable. */

enum base_walk { /* Set the revision for FILE to *REV. */
    BASE_REGISTER,
    /* Get the revision for FILE and put it in a newly malloc'd string
     * REV, or put NULL if not mentioned. */
    BASE_GET,
    /* Remove FILE. */
    BASE_DEREGISTER
};

static void base_walk_PROTO ((enum base_walk, struct file_info * fi,
                               char *rev));

/* Read through the lines in CVS/Base, taking the actions as documented
for CODE. */

static void
base_walk (code, finfo, rev)
enum base_walk code;
struct file_info *finfo;
char *rev;
{
    FILE *fp;

code line;
size_t line_allocated;
FILE *newf;
char *baserev_fullname;
char *baserevtmp_fullname;

line = NULL;
line_allocated = 0;
newf = NULL;

/* First compute the fullnames for the error messages. This
computation probably should be broken out into a separate function,
as recurse.c does it too and places like Entries.Open should be
doing it. */
baserev_fullname = xmalloc (sizeof (CVSADM_BASEREV)
                           + strlen (finfo->update_dir)
                           + 2);
baserev_fullname[0] = '\0';
baserevtmp_fullname = xmalloc (sizeof (CVSADM_BASEREVTMP)
                            + strlen (finfo->update_dir)
                            + 2);
baserevtmp_fullname[0] = '\0';
if (finfo->update_dir[0] != '\0')
    {
        strcat (baserev_fullname, finfo->update_dir);
        strcat (baserevtmp_fullname, finfo->update_dir);
        strcat (baserevtmp_fullname, "*/");
    }
strcat (baserev_fullname, CVSADM_BASEREV);
strcat (baserevtmp_fullname, CVSADM_BASEREVTMP);
fp = CVSFOPEN (CVSADM_BASEREV, "r");
if (fp == NULL)
    {
        if (errno)
            { error (0, errno, "cannot open file for reading",  baserev_fullname);
              goto out;
            }
        
        fp = CVSFOPEN (CVSADM_BASEREVTMP, "r");
        if (fp == NULL)
            { error (0, errno, "cannot open file for reading",  baserev_fullname);
              goto out;
            }
    }

/* Compute the fullnames for the error messages. This
computation probably should be broken out into a separate function,
as recurse.c does it too and places like Entries.Open should be
doing it. */
baserev_fullname = xmalloc (sizeof (CVSADM_BASEREV)
                           + strlen (finfo->update_dir)
                           + 2);
baserev_fullname[0] = '\0';
baserevtmp_fullname = xmalloc (sizeof (CVSADM_BASEREVTMP)
                            + strlen (finfo->update_dir)
                            + 2);
baserevtmp_fullname[0] = '\0';
if (finfo->update_dir[0] != '\0')
    {
        strcat (baserev_fullname, finfo->update_dir);
        strcat (baserevtmp_fullname, finfo->update_dir);
        strcat (baserevtmp_fullname, "*/");
    }
strcat (baserev_fullname, CVSADM_BASEREV);
strcat (baserevtmp_fullname, CVSADM_BASEREVTMP);
fp = CVSFOPEN (CVSADM_BASEREVTMP, "r");
if (fp == NULL)
    { error (0, errno, "cannot open file for reading",  baserev_fullname);
      goto out;
    }
switch (code)
{
    case BASE_REGISTER:
        case BASE_DEREGISTER:
            newf = CVS_FOPEN (CVSADM_BASEREVTMP, "w");
            if (newf == NULL)
            {
                error (0, errno, "cannot open %s for writing",
                        baseretmpfullname);
                goto out;
            }
            break;
    case BASE_GET:
        *rev = NULL;
        break;
}
if (fp != NULL)

    while (getline (&line, &line_allocated, fp) >= 0)
    {
        char *linefile;
        char *p;
        char *linerev;
        if (line[0] != 'B')
            /* Ignore, for future expansion. */
            continue;
        linefile = line + 1;
        p = strchr (linefile, '/');
        if (p == NULL)
            /* Syntax error, ignore. */
            continue;
        linerev = p + 1;
        p = strchr (linerev, '/');
        if (p == NULL)
            continue;
        linerev[-1] = '/';
        if (strncmp (linefile, finfo->file) == 0)
        {
            switch (code)
            {
                case BASE_REGISTER:
                case BASE_DEREGISTER:
                    /* Don't copy over the old entry, we don't want it. */
                    break;
                case BASE_GET:
                    *p = '/';
                    *rev = xstrdup (linerev);
                    *p = '/';
                    goto gotit;
            }
        }
        else
        {
            linerev[-1] = '/';
            switch (code)
            {
                case BASE_REGISTER:
                case BASE_DEREGISTER:
                    if (fprintf (newf, "%s%0a%0a
", finfo->file, *rev) < 0)
                        error (0, errno, "error writing %s",
                                baseretmpfullname);
                    break;
                case BASE_GET:
                    break;
            }
        }
    }
    if (error (fp))
        error (0, errno, "cannot read %s", baseretmpfullname);
    goto out:
    if (code == BASE_REGISTER)
    {
        if (fprintf (newf, "%s%0a%0a%0a", finfo->file, *rev) < 0)
            error (0, errno, "error writing %s",
                    baseretmpfullname);
    }
out:
    if (line != NULL)
        free (line);
if (fp != NULL) {
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", baserevfullname);
} if (newf != NULL) {
    if (fclose (newf) < 0)
        error (0, errno, "cannot close %s", baserevtmpfullname);
    rename_file (CVSADM_BASEREVTMP, CVSADM_BASEREV);
}
free (baserevfullname);
free (baserevtmpfullname);
/* Return, in a newly malloc'd string, the revision for FILE in CVS/Baserev,
 or NULL if not listed. */
char *
baseget (finfo)
struct file_info *finfo;
{
    char *rev;
    basewalk (BASE_GET, finfo, &rev);
    return rev;
}
/* Set the revision for FILE to REV. */
void
baseregister (finfo, rev)
struct file_info *finfo,
char *rev;
{
    basewalk (BASE_REGISTER, finfo, &rev);
}
/* Remove FILE. */
void
basederegister (finfo)
struct file_info *finfo;
{
    basewalk (BASE_DEREGISTER, finfo, NULL);
}
error.c

/* error.c – error handler for noninteractive utilities
   Copyright (C) 1990-1992 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details. */

/* David MacKenzie */
/* Brian Berliner added support for CVS */

#include "cvs.h"

#include <stdio.h>

/* If non-zero, error will use the CVS protocol to stdout to report error
   messages. This will only be set in the CVS server parent process;
   most other code is run via da_cvs_command, which forks off a child
   process and packages up its stderr in the protocol. */
int error_use_protocol;

ifdef HAVE_VPRINTF

ifndef __STDC__

#include <stdarg.h>

define va_START(args, lastarg) va_start(args, lastarg)
else /* __STDC__ */

#include <varargs.h>

define va_START(args, lastarg) va_start(args, lastarg)
endif /* __STDC__ */

else /* !HAVE_VPRINTF */

eendif /* __STDC__ */
endif !HAVE_VPRINTF */

define HAVE_DOPRINT

define va_alist args

define va_del int args;
else /* !HAVE_DOPRINT */

define va_alist a1, a2, a3, a4, a5, a6, a7, a8

define va_del char *a1, *a2, *a3, *a4, *a5, *a6, *a7, *a8;
endif /* !HAVE_DOPRINT */

dendif /* HAVE_VPRINTF */

ifdef STDC_HEADERS

#include <string.h>

ifndef __STDC__

void exit(int status);
else /* __STDC__ */

void exit();
endif /* __STDC__ */

dendif /* STDC_HEADERS */

difdef strerror

extern char *strerror ();

dendif

eextern int vasprintf ();

evoid error_exit PROTO ((void))
{
  LockCleanup();

ifdef SERVER_SUPPORT

    if (server_active)
      server_cleanup (0);
endif

ifdef SYSTEM_CLEANUP

    /* Hook for OS specific behavior, for example socket subsystems on
       NT and OS2 or dealing with windows and arguments on Mac. */
    SYSTEM_CLEANUP ();
endif

    exit (EXIT_FAILURE);

eendif /* !HAVE_VPRINTF */

eendif /* __STDC__ */

eendif /* HAVE_VPRINTF */

eendif /* HAVE_DOPRINT */

eendif /* __STDC__ */

eendif /* HAVE_VPRINTF */

/* Print the program name and error message MESSAGE, which is a printf-style
   format string with optional args.
   If ERRNUM is nonzero, print its corresponding system error message.
   Exit with status EXIT_FAILURE if STATUS is nonzero. If MESSAGE is "", no need to print a message.

   I think this is largely cleaned up to the point where it does the right
thing for the server, whether the normal server is active (child process) or
the error_use_protocol (parent process) case. The one exception
is that STATUS nonzero for error_use_protocol probably doesn't work yet;
in that case still need to use the pending error machinery in server.c.

error() does not molest errno; some code (e.g. EntriesWrite) depends
on being able to say something like:

error (0, 0, "foo");

error (0, errno, "bar");

*/

/** VARARGS */
void
if defined (HAVE_VPRINTF) & & defined (_STDC_
error (int status, int errnum, const char *message, ...)
else
error (status, errnum, message, va_list)
va_end;
#endif
{ /* Prevent strtoul (via int vasprintf) from clobbering it. */
int save_errno = errno;
#endif
if (message[0] != '\0')
{
 va_list args;
 char *mess = NULL;
 char *entire;
 size_t len;
 VA_START (args, message);
 vasprintf (&mess, message, args);
 va_end (args);
 if (mess == NULL)
 { entire = NULL;
 status = 1;
 }
 else
 { len = strlen (mess) + strlen (program_name) + 80;
 if (command_name != NULL)
 len += strlen (command_name);
 if (errno != 0)
 len += strlen (strerror (errno));
 entire = malloc (len);
 if (entire == NULL)
 { free (mess);
 status = 1;
 }
 else
 { strcpy (entire, program_name);
 if (command_name != NULL & & command_name[0] != '\0')
 { strcat (entire, command_name);
 if (status != 0)
 strcat (entire, "":");
 strcat (entire, strerror (errno));
 strcat (entire, "\n");
 free (mess);
 }
 cvs_outerr (entire ? entire : "out of memory\n", 0);
 if (entire != NULL)
 free (entire);
 }
#endif

else /* No HAVE_VPRINTF */
{ /* I think that all relevant systems have vprintf these days. But
just in case, I'm leaving this code here. */
if (message[0] != '\0')
# If errnum is nonzero, print its corresponding system error message.
if (errnum)
    fprintf (out, "%s: %s", sterror (errnum));
putc (’\n’, out);

/* In the error-unprotocol case, this probably does
something useful. In most other cases, I suspect it is a
noop (either stderr is line buffered or we haven’t written
anything to stderr) or unnecessary (if stderr is not line
buffered, maybe there is a reason…). */
flush (out);

#endif /* No HAVE_VPRINTF */

if (status)
    error = exit ();
errnum = save_errno;

/* Print the program name and error message MESSAGE, which is a printf-style
format string with optional args to the file specified by FP:
If ERRNUM is nonzero, print its corresponding system error message.
Exit with status EXIT_FAILURE if STATUS is nonzero. */
void
if defined (HAVE_VPRINTF) && defined (__STDC__) ||
fprintf (FILE *fp, int status, int errnum, char *message, ...
else
fprintf (fp, status, errnum, message, va_list)
FILE *fp;
int status;
char *message;
va_list;
#endif

#endif /* HAVE_VPRINTF */

va_list args;
#endif

fprintf (fp, "%s", program_name);
#endif

#endif

if (errnum)
    fprintf (fp, "%s", sterror (errnum));
putc (’\n’, fp);
flush (fp);
if (status)
    error = exit ();
}
# A.19  error.h

/* error.h – declaration for error-reporting function

Copyright (C) 1995 Software Foundation, Inc.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

#ifndef _error_h_
#define _error_h_

/* Add prototype support. Normally this is done in cvs.h, but that
doesn't get included from lib/savecwd.c. */
#ifndef PROTO
#define PROTOARGS void
#else
#define PROTOARGS ()
#endif
#endif

#ifndef __attribute__
/* This feature is available in gcc versions 2.5 and later. */
#define __attribute__(()) __attribute__((format(printf, 3, 4)));
#else
#define __attribute__(())
#endif

#ifndef __STDC__
void error(int, int, const char *, ...)

__attribute__((format(printf, __print__, __print__, 3, 4)));
#else
void error();
#endif

/* Exit due to an error. Similar to error (1, 0, “message”), but call
it in the case where the message has already been printed. */
extern void errorexit PROTO (;;) ;

/* If non-zero, error will use the CVS protocol to report error
messages. This will only be set in the CVS server parent process;
most other code is run via do_cvs_command, which forks off a child
process and packages up its stderr in the protocol. */
extern int error_use_protocol;
#endif /* _error_h_ */
A.20 expand_path.c

/* expand_path.c - expand environmental variables in passed in string */
/* The main routine is expand_path(), it is the routine that handles */
/* the `~' character in four forms: */
/*  `name' */
/*  `name/' */
/*  `'` */
/* and handles environment variables contained within the pathname */
/* which are defined by: */
/*  `${name}' (var_name is the name of the environ variable.) */
/*  `var_name' (var_name ends w/ non-alphanumeric char other than '`) */
*/
#include "cvs.h"
#include <sys/types.h>

static char *expand_variable PROTO((char *env, char *file, int line));

/* User variables. */
List *variable_list = NULL;

static void variable_delproc PROTO ((Node *));

static void variable_delproc (node)
Node *node;

{
  free (node->data);
}

/* Currently used by -s option; we might want a way to set user */
/* variables in a file in the $CVSROOT/CVSROOT directory too. */
void
variable_set (nameval)
char *nameval;

{
  char *p;
  char *name;
  Node *node;

  p = nameval;
  while (isalnum (*p) || *p == '_')
      + + p;
  if (*p != 'a')
      error (1, 0, "illegal character in user variable name in $\alpha", nameval);
  if (p == nameval)
      error (1, 0, "empty user variable name in $\alpha", nameval);
  name = xmalloc (p - nameval + 1);
  strcpy (name, nameval, p - nameval);
  name[p - nameval] = '\0';
  /* Make p point to the value. */
  + + p;
  if (strchr (p, '\012') != NULL)
      error (1, 0, "linefeed in user variable value in $\alpha", nameval);
  if (variable_list == NULL)
      variable_list = getlist ();
  node = findnode (variable_list, name);
  if (node == NULL)
    {
      node = getnode ();
      node->type = VARIABLE;
      node->delproc = variable_delproc;
      node->key = name;
      node->data = xstrdup (p);
      (void) addnode (variable_list, node);
    }
  else
    {
      /* Replace the old value. For example, this means that -s */
      /* options on the command line override ones from .cvsrc. */
      free (node->data);
      node->data = xstrdup (p);
      free (name);
    }

}

/* This routine will expand the pathname to account for `~' and $ */
/* characters as described above. Returns a pointer to a newly */
/* malloc'd string. If an error occurs, an error message is printed */
/* via error() and NULL is returned. FILE and LINE are the filename */
/* and linenumber to include in the error message. FILE must point */
/* to something; LINE can be zero to indicate the line number is not */
known. */

char *expand_path (name, file, line)
char *name;
char *file;
int line;
{
char *c;
char *d;

char *mybuf = NULL;
size_t mybuf_size = 0;
char *buf = NULL;
size_t buf_size = 0;
size_t doff;
char *result;

/* Sorry this routine is so ugly; it is a head-on collision
between the 'traditional' unix $d++ style and the need to
dynamically allocate. It would be much cleaner (and probably
faster, not that this is a bottleneck for CVS) with more use of
streepy & friends, but I haven't taken the effort to rewrite it
thuyl. */

/* First copy from NAME to MYBUF, expanding $<foo> as we go. */
s = name;
d = mybuf;
doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
for (d = mybuf + doff;
while (((d += + = s)))
{
if (s++; + = ' ')
{
char *p = d;
char *e;
int flag = (s == ' ');
doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
d = mybuf + doff;
for (; (d[++ = ++); s++)
{
if (flag && s == ' ')
: isalnum (*s) == 0 & & s != ' ')
brake;
doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
d = mybuf + doff;
}

---> d = ' \0';
e = expand_variable (&p[flag], file, line);

if (s)
{
doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
d = mybuf + doff;
for (d = &p[-1]; (d++ = *e++))
{

doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
d = mybuf + doff;
}
}

if (flag & & s)
++;
}
else
  /* expand_variable has already printed an error message. */
goto error_exit;
}
doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
d = mybuf + doff;
}
doff = d = mybuf;
expand_string (&mybuf, &mybuf_size, doff + 1);
d = mybuf + doff;
*d = "\0";

/* Then copy from MYBUF to BUF, expanding ". */
s = mybuf;
d = buf;
/* If you don't want "username "/ to be expanded simply remove
 * This entire if statement including the else portion
 */

if \(s++ == '\''\)
{
    char *t;
    char *p = s;
    if ((s== '/' | s==0)
        t = gethomedir();
    else
        #ifndef GETPWNAM_MISSING
        for (; *p!='/' & *p; p++)
            *p = 0;
        if (line != 0)
            error(0, 0, "%s:tilde expansion not supported on this system", file, line);
        else
            error(0, 0, "%s:tilde expansion not supported on this system", file);
        return NULL;
    #else
        struct passwd *ps;
        for (; *p!='/' & *p; p++)
            *p = 0;
        ps = getpwnam(s);
        if (ps == 0)
            if (line != 0)
                error(0, 0, "%s:%d: no such user %s", file, line, s);
            else
                error(0, 0, "%s: no such user %s", file, s);
        return NULL;
    #endif
}

doff = d - buf;
expand_string(&buf, &bufsize, doff + 1);
d = buf + doff;
while (("s++ = '+s++")
    doff = d - buf;
    expand_string(&buf, &bufsize, doff + 1);
    d = buf + doff;
}
--d;
if (*p == 0)
    *p = '"'; /* always add */
    s=p;
}
else
    --s;
    /* Kill up to here */
    doff = d - buf;
    expand_string(&buf, &bufsize, doff + 1);
    d = buf + doff;
    while (("s++ = '+s++")
        doff = d - buf;
        expand_string(&buf, &bufsize, doff + 1);
        d = buf + doff;
    }
    --d;
    if (*p == 0)
        *p = '\n'; /* always add */
    s=p;
}
else
    /* Kill up to here */
    doff = d - buf;
    expand_string(&buf, &bufsize, doff + 1);
    d = buf + doff;
    while (("s++ = '+s++")
        doff = d - buf;
        expand_string(&buf, &bufsize, doff + 1);
        d = buf + doff;
    }
    --d;
    /* OK, buf contains the value we want to return. Clean up and return it. */
free (mybuf);
    /* Save a little memory with xstrdup; buf will tend to allocate more than it needs to. */
    result = xstrdup(buf);
    free (buf);
    return result;
}

error_exit:
if (mybuf != NULL)
    free (mybuf);
if (buf != NULL)
    free (buf);
    return NULL;
}

static char *
expand_variable (name, file, line)
char *name;
char *file;
```c
int line;

270  {  
if (strcmp (name, CVSROOT_ENV) == 0)  
return CVSroot_original;
else if (strcmp (name, RCSBIN) == 0)  
{  
    error (0, 0, "RCSBIN internal variable is no longer supported");
    return NULL;
  }
else if (strcmp (name, EDITOR1_ENV) == 0)  
return Editor;
else if (strcmp (name, EDITOR2_ENV) == 0)  
return Editor;
else if (strcmp (name, "USER") == 0)  
return getcaller ();  
else if (isalpha (name[0]))  
{  
    /* These names are reserved for future versions of CVS,  
       so that it is why it is an error. */
    if (line != 0)  
error (0, 0, "%s:%d: no such user variable \$%s",  
file, line, name);
else  
error (0, 0, "%s: no such user variable \$%s",  
file, name);
    return NULL;
  }
else if (name[0] == '=')  
{  
    Node *node;  
    /* Crazy syntax for a user variable. But we want  
       *something* that lets the user name a user variable  
       anything he wants, without interference from  
       (existing or future) internal variables. */  
    node = findnode (variable_list, name + 1);  
    if (node == NULL)  
    {  
        if (line != 0)  
error (0, 0, "%s:%d: no such user variable $\%s\$",  
file, line, name);
else  
error (0, 0, "%s: no such user variable $\%s\$",  
file, name);
    return NULL;
    }  
return node->data;  
}  
else  
{  
    /* It is an unrecognized character. We return an error to  
       reserve these for future versions of CVS; it is plausible  
       that various crazy syntaxes might be invented for inserting  
       information about revisions, branches, etc. */
    if (line != 0)  
error (0, 0, "%s:%d: unrecognized variable syntax \%s",  
file, line, name);
else  
error (0, 0, "%s: unrecognized variable syntax \%s",  
file, name);
return NULL;
    }
}
}
A.21 fileattr.c

/* Implementation for file attribute munging features. */

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

#include "cvs.h"
#include "getline.h"
#include "cvs.h"
#include "assert.h"

static void fileattr_read PROTO ((void));
static int writeattr_proc PROTO ((Node *, void *));

#include <assert.h>

static void fileattr_read PROTO ((void));
static int writeattr_proc PROTO ((Node *, void *));

/* Where to look for CVSREP_FILEATTR. */
static char *fileattr_stored_repos;

/* The in-memory attributes. */
static List *attrlist;
static char *fileattr_default_attr;

/* We have already tried to read attributes and failed in this directory
(for example, there is no CVSREP_FILEATTR file). */
static int attr_read_attempted;

/* Have the in-memory attributes been modified since we read them? */
static int attr_modified;

/* More in-memory attributes: linked list of unrecognized
fileattr lines. We pass these on unchanged. */
struct unrecog {
  char *line;
  struct unrecog *next;
};

static struct unrecog *unrecog_head;

/* Note that if noone calls fileattr_set, this is very cheap. No stat(),
no open(), no nothing. */
void
fileattr_startdir (repos)
char *repos;
{
  assert (fileattr_stored_repos == NULL);
  fileattr_stored_repos = xstrdup (repos);
  assert (attrlist == NULL);
  attr_read_attempted = 0;
  assert (unrecog_head == NULL);
}

static void
fileattr_stoppedproc (node)
Node *node;
{
  assert (node->data != NULL);
  free (node->data);
  node->data = NULL;
}

/* Read all the attributes for the current directory into memory. */
static void
fileattr_read ()
{
  char *fname;
  FILE *fp;
  char *line = NULL;
  size_t line_len = 0;

  /* If there are no attributes, don't waste time repeatedly looking
for the CVSREP_FILEATTR file. */
  if (attr_read_attempted) return;

  /* If NULL was passed to fileattr_startdir, then it isn't kosher to look
at attributes. */
  assert (fileattr_stored_repos != NULL);

  fname = xmalloc (strlen (fileattr_stored_repos)
                 + 1
                 + sizeof (CVSREP_FILEATTR) + 1);
  strcpy (fname, fileattr_stored_repos);
  strcat (fname, /* */);
strcat (fname, CVSREPFILEATTR);

attr_read_attempted = 1;
fp = CVS_FOPEN (fname, FOPEN_BINARYREAD);
if (fp == NULL)
{
  if (fexistname, error (errno))
    error (0, errno, "cannot read %s", fname);
  free (fname);
  return;
}

attrlist = getlist ();
while (1) {
  int nread;
  nread = getline (&line, &line_len, fp);
  if (nread < 0)
    break;
  /* Remove trailing newline. */
  line[nread - 1] = '\0';
  if (line[0] == 'F') {
    char *p;
    Node *newnode;
    p = strchr (line, '\t');
    if (p == NULL)
      error (1, 0,
        "File attribute database corruption: tab missing in %s",
        fname);
    *p++ = '\0';
    newnode = getnode ();
    newnode->type = FILEATTR;
    newnode->delproc = fileattr_delproc;
    newnode->key = xstrdup (line + 1);
    newnode->data = xstrdup (p);
    if (addnode (attrlist, newnode) != 0)
      /* If the same filename appears twice in the file, discard
       * any line other than the first for that filename. This
       * is the way that CVS has behaved since file attributes
       * were first introduced. */
      free (newnode);
    else if (line[0] == 'D')
    {
      char *p;
      /* Currently nothing to skip here, but for future expansion,
       * ignore anything located here. */
      p = strchr (line, '\t');
      if (p == NULL)
        error (1, 0,
          "File attribute database corruption: tab missing in %s",
          fname);
      ++p;
      fileattr_defaultattrs = xstrdup (p);
    }
    else {
      /* Unrecognized type, we want to just preserve the line without
       * changing it, for future expansion. */
      struct unrecog *new;
      new = (struct unrecog *) xmalloc (sizeof (struct unrecog));
      new->line = xstrdup (line);
      new->next = unrecoghead;
      unrecoghead = new;
      if (ferror (fp))
        error (0, errno, "cannot read %s", fname);
      if (line != NULL)
        free (line);
      if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", fname);
      attr_modified = 0;
      free (fname);
    }
  char *fileattr_read (filename, attrname)
const char *filename;
const char *attrname;
{
  Node *node;
  size_t attrname_len = strlen (attrname);
  char *p;
  if (attrlist == NULL)
    fileattr_read ();
  if (attrlist == NULL)
    /* Either nothing has any attributes, or fileattr_read already printed
    */
    return;
  if (p = strchr (filename, '\t'))
    filename = & filename[1];
  if (p = strchr (attrname, '\t'))
    attrname = & attrname[1];
  if (p)
    return;
  if (p = strchr (filename, '\t'))
    filename = & filename[1];
  if (p = strchr (attrname, '\t'))
    attrname = & attrname[1];
  if (p)
    return;
  attrlist = getlist ();
  while (1) {
    int nread;
    nread = getline (&line, &line_len, fp);
    if (nread < 0)
      break;
    /* Remove trailing newline. */
    line[nread - 1] = '\0';
    if (line[0] == 'F') {
      char *p;
      Node *newnode;
      p = strchr (line, '\t');
      if (p == NULL)
        error (1, 0,
          "File attribute database corruption: tab missing in %s",
          fname);
      *p++ = '\0';
      newnode = getnode ();
      newnode->type = FILEATTR;
      newnode->delproc = fileattr_delproc;
      newnode->key = xstrdup (line + 1);
      newnode->data = xstrdup (p);
      if (addnode (attrlist, newnode) != 0)
        /* If the same filename appears twice in the file, discard
         * any line other than the first for that filename. This
         * is the way that CVS has behaved since file attributes
         * were first introduced. */
        free (newnode);
      else if (line[0] == 'D')
      {
        char *p;
        /* Currently nothing to skip here, but for future expansion,
         * ignore anything located here. */
        p = strchr (line, '\t');
        if (p == NULL)
          error (1, 0,
            "File attribute database corruption: tab missing in %s",
            fname);
        ++p;
        fileattr_defaultattrs = xstrdup (p);
      }
      else {
        /* Unrecognized type, we want to just preserve the line without
         * changing it, for future expansion. */
        struct unrecog *new;
        new = (struct unrecog *) xmalloc (sizeof (struct unrecog));
        new->line = xstrdup (line);
        new->next = unrecoghead;
        unrecoghead = new;
        if (ferror (fp))
          error (0, errno, "cannot read %s", fname);
        if (line != NULL)
          free (line);
        if (fclose (fp) < 0)
          error (0, errno, "cannot close %s", fname);
        attr_modified = 0;
        free (fname);
      }
    }
  }
}
an error message. */
return NULL;
if (filename == NULL)
p = fileattrdefaultattrs;
else
{  
    node = findnode (attrlist, filename);
    if (node == NULL)
/* A file not mentioned has no attributes. */
    return NULL;
}
while (p)
{
    if (strcmp (filename, p + strlenp) == 0)
        return p + strlenp + 1;
}
p = strchr (p, ';
return NULL;
*/ The file doesn't have this attribute. */

char * fileattr_get0 (filename, attrname)
const char * filename;
const char * attrname;
{
    char * cp,
    char * cpdend;
    char * retval;
    cp = fileattr_get (filename, attrname);
    if (cp == NULL)
return NULL;
    cpdend = strchr (cp, ';
if (cpdend == NULL)
    cpdend = cp + strlen (cp);
    retval = xmalloc (cpdend - cp + 1);
    strncpy (retval, cp, cpdend - cp);
    retval[cpdend - cp] = '\0';
    return retval;
}

char * fileattr_modify (list, attrname, attrval, namevalsep, entsep)
char * list;
const char * attrname;
const char * attrval;
int namevalsep;
int entsep;
{
    char *retval;
    char *cp;
    size_t attrname_len = strlen (attrname);
    /* Portion of list before the attribute to be replaced. */
    char *pre;
    char * pend;
    /* Portion of list after the attribute to be replaced. */
    char *post;
    char *p;
    char *p2;
    p = list;
    pre = list;
    pend = NULL;
    /* post is NULL unless set otherwise. */
    /* post = NULL;
    p2 = NULL;
    if (list != NULL)
    {
        while (1) {
            p2 = strchr (p, entsep);
            if (p2 == NULL)
            {
                p2 = p + strlen (p);
                if (preend == NULL)
                preend = p2;
            }
            else
                ++p2;
            if (p2 == NULL)
            {
                /* p2 is NULL. */
                p = p2;
                if (pend == NULL)
                pend = p2;
            }
if (strncmp (attrname, p, attrname_len) == 0 && p[attrname_len] == namevalsep)
{
    /* Found it. */
    preend = p;
    if (preend > list)
        /* Don't include the preceding entsep. */
        post = p2;
}
if (p2[0] == '\0')
    break;
}
if (post == NULL)
    post = p2;
if ((preend == pre && attrval == NULL && post == p2))
    return NULL;
retval = xmalloc ((preend - pre) + 1 + (attrval == NULL ? 0 : (attrname_len + 1 + strlen(attrval))) + 1 + (p2 - post) + 1);
if (preend != pre)
{
    strncpy (retval, pre, preend - pre);
    rp = retval + (preend - pre);
    if (attrval != NULL)
        *rp++ = entsep;
    *rp = '\0';
}
else
    if (attrval != NULL)
    {
        strcat (retval, attrname);
        rp = retval + strlen (retval);
        *rp++ = namevalsep;
        strcpy (rp, attrval);
    }
    if (post != p2)
    {
        rp = retval + strlen (retval);
        *rp++ = entsep;
        rp = retval;
        rp += p2 - post;
        *rp = '\0';
    }
    return retval;
}
void
fileattr_set (filename, attrname, attrval)
const char *filename;
const char *attrname;
const char *attrval;
{
    Node *node;
    char *p;
    if (filename == NULL)
    {
        p = fileattr_modify (fileattr_default_attrs, attrname, attrval, '-1', '-1');
    }
    if (fileattr_default_attrs != NULL)
    {
        free (fileattr_default_attrs);
        fileattr_default_attrs = p;
        attrlist_modified = 1;
        return;
    }
    if (attrlist == NULL)
        fileattr_read ();
    if (attrlist == NULL)
    {
        /* Not sure this is a graceful way to handle things
in the case where fileattr_read was unable to read the file. */
        /* No attributes existed previously. */
        attrlist = getlist ();
    }
    node = findnode (attrlist, filename);
    if (node == NULL)
    {

if (attrval == NULL)
   /* Attempt to remove an attribute which wasn't there. */
   return;

   /* First attribute for this file. */
   node = getnode();
   node->type = FILEATTR;
   node->delproc = fileattrdelproc;
   node->key = xstrdup(filename);
   node->data = xmalloc(strlen(attrname) + 1 + strlen(attrval) + 1);
   strpy (node->data, attrname);
   strcat (node->data, "=");
   strcat (node->data, attrval);
   addnode (attrlist, node);
}

p = fileattrmodify (node->data, attrname, attrval, '='; ');
if (p == NULL)
   delnode (node);
else
   {
      free (node->data);
      node->data = p;
   }

attr_modified = 1;

char * 
fileattrgetall (filename)
const char * filename;
{
   Node *node;
   char *p;
   
   if (attrlist == NULL)
      fileattrread ();
   if (attrlist == NULL)
      /* Either nothing has any attributes, or fileattrread already printed
       an error message. */
      return NULL;

   if (filename == NULL)
      p = fileattrdefaultattrs;
   else
   {
      node = findnode (attrlist, filename);
      if (node == NULL)
         /* A file not mentioned has no attributes. */
         return NULL;
      p = node->data;
   }
   return xstrdup (p);
}

void 
fileattrsetall (filename, attrs)
const char * filename;
const char * attrs;
{
   Node *node;
   
   if (filename == NULL)
   {
      if (fileattrdefaultattrs != NULL)
         free (fileattrdefaultattrs);
      fileattrdefaultattrs = xstrdup (attrs);
      attr_modified = 1;
      return;
   }
   if (attrlist == NULL)
      fileattrread ();
   if (attrlist == NULL)
   {
      /* Not sure this is a graceful way to handle things
         in the case where fileattrread was unable to read the file. */
      /* No attributes existed previously. */
      attrlist = getlist ();
   }

   node = findnode (attrlist, filename);

   if (node == NULL)
   {
      /* The file had no attributes. Add them if we have any to add. */
      if (attrs != NULL)
      {
         node = getnode ();
         node->type = FILEATTR;
         node->delproc = fileattrdelproc;
         node->key = xstrdup (filename);
      }
278  Source code

```c
    node->data = xstrdup (attrs);
    addnode (attrlist, node);
    } } else {  
      if (attrs == NULL)  
        delnode (node);  
      else {  
        free (node->data);  
        node->data = xstrdup (attrs);
      }
    }
    attrs_modified = 1;
  }
void fileattr_newfile (filename)
const char *filename;
470  {
  Node *node;
  if (attrlist == NULL)
    fileattr_read ();
  if (fileattr_defaultattrs == NULL)
    return;
  if (attrlist == NULL)
    {* Not sure this is a graceful way to handle things
     * in the case where fileattr_read was unable to read the file. */  
    /* No attributes existed previously. */  
    attrlist = getlist ();
  }
  node = getnode ();
  node->type = FILEATTR;
  node->delproc = fileattr_delproc;
  node->key = xstrdup (filename);
  node->data = xstrdup (fileattr_defaultattrs);
  addnode (attrlist, node);
  attrs_modified = 1;
}
static int writeattr_proc (node, data)
Node *node;
void *data;
500  {  
  FILE *fp = (FILE *)data;  
  fputs ("F", fp);
  fputs (node->key, fp);
  fputs ("\t", fp);
  fputs (node->data, fp);
  fputs (__STRICT__, fp);
  return 0;
}
void fileattr_write ()
{
  FILE *fp;  
  char *fname;  
  mode_t umask;
  if (attrs_modified)
    return;
510  {  
    if (nosec)  
      return;
    /* If NULL was passed to fileattr_startdir, then it isn't kosher to set 
     * attributes. */  
    assert (fileattr_storedrepos != NULL);
    fname = xmalloc (strlen (fileattr_storedrepos) + 1 + 1);  
    strcpy (fname, fileattr_storedrepos);  
    strcat (fname, "/");  
    strcat (fname, CVSREP_FILEATTR);
    if (list_isempty (attrlist) && fileattr_defaultattrs == NULL 
      && unrecogread == NULL)
```
540  
/* There are no attributes. */
if (unlink_file (name) < 0)
{
    if (errno)
    {
        error (errno, "cannot remove %s", name);
    }
}

550  
/* Now remove CVSREP directory, if empty. The main reason we bother
is that CVS 1.6 and earlier will choke if a CVSREP directory
exists, so provide the user a graceful way to remove it. */
strncpy (name, fileattr_stored_repos),
strcat (name, "/./");
strcat (name, CVSREP);
if (CVS_RMDIR (name) < 0)
{
    if (errno != ENOTEMPTY)
    {
        /* Don't know why we would be here if there is no CVSREP
directory, but it seemed to be happening anyway, so
check for it. */
        & if (existence_error (errno))
        {
            error (errno, "cannot remove %s", name);
        }
        free (name);
        return;
    }

570  
      omask = umask (cvsumask);
fp = CVS_FOPEN (name, FOPEN_BINARY_WRITE);
if (fp == NULL)
{
    if (errno)
    {
        /* Maybe the CVSREP directory doesn't exist. Try creating it. */
        char *repname;
        repname = xmalloc (strlen (fileattr_stored_repos) + 1 +
                          sizeof (CVSREP) + 1);
        strcpy (repname, fileattr_stored_repos);
        strcat (repname, "/.");
        strcat (repname, CVSREP);
        if (CVS_MKDIR (repname, 0777) == 0 & &
            errno != EEXIST)
        {
            error (errno, "cannot make directory%s", repname);
        (void) umask (omask);
        free (repname);
        return;
        }
        free (repname);
        fp = CVS_FOPEN (name, FOPEN_BINARY_WRITE);
    }
    if (fp == NULL)
    {
        error (errno, "cannot write %s", name);
        (void) umask (omask);
        return;
    }
}

590  
    (void) umask (omask);
/ * First write the "F" attributes. */
    walklist (attrlist, writeattrproc, fp);

610  
/* Then the "D" attribute. */
if (fileattr_default_attr != NULL)
{
    fputs (^D, fp);
    fputs (fileattr_default_attr, fp);
    fputs (^"D", fp);
}
/* Then any other attributes. */
while (unrecog_head != NULL)
{
    struct unrecog *p;
    p = unrecog_head;
    fputs (p->line, fp);
    fputs (^"D", fp);
    unrecog_head = p->next;
    free (p->line);
free (p);
}

if ((close (fp) < 0)
  error (0, errno, "cannot close %s", fname);
attr_modified = 0;
free (fname);
}

void

fileattr_free ()

{ /* Note that attr_modified will ordinarily be zero, but there are
   a few cases in which fileattr_write will fail to zero it (if
   noexec is set, or error conditions). This probably is the way
   it should be. */
dellist (&attrlist);
if (fileattr_stored_repos != NULL)
  free (fileattr_stored_repos);
fileattr_stored_repos = NULL;
if (fileattr_default_attrs != NULL)
  free (fileattr_default_attrs);
fileattr_default_attrs = NULL;
}
A.22 fileattr.h

/* Declarations for file attribute munging features.*/

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

#ifndef FILEATTR_H
/* File containing per-file attributes. Format is a series of entries:
ENT-TYPE FILENAME <tab> ATTRNAME = ATTRVAL
{}; ATTRNAME = ATTRVAL <linefeed>
*/

ENT-TYPE is 'F' for a file, in which case the entry specifies the
attributes for that file.

ENT-TYPE is 'D', and FILENAME empty, to specify default attributes
to be used for newly added files.

Other ENT-TYPE are reserved for future expansion. CVS 1.9 and older
will delete them any time it writes file attributes. Current versions
of CVS will preserve them.

Note that the order of the line is not significant; CV2 is free to
rearrange them at its convenience.

There is currently no way of quoting tabs or linefeeds in the
filename, '=' in ATTRNAME, ';' in ATTRVAL, etc. I'm not sure
whether I think we need one. Note: the current implementation also
doesn't handle '\0' in any of the fields.

By convention, ATTRNAME starting with ':' is for an attribute given
special meaning by CVS; other ATTRNAMEs are for user-defined attributes
(or will be, once we add commands to manipulate user-defined attributes).

Built-in attributes:

watched: Present means the file is watched and should be checked out
read-only.

watchers: Users with watches for this file. Value is
WATCHER > TYPE { WATCHER > TYPE }
where WATCHER is a username, and TYPE is edit,unedit,commit separated by
+ (or nothing if none; there is no "none" or "all" keyword).

editors: Users editing this file. Value is
EDITOR > VAL { EDITOR > VAL }
where EDITOR is a username, and VAL is TIME+HOSTNAME+PATHNAME, where
TIME is when the "cvs edit" command happened, and
HOSTNAME and PATHNAME are for the working directory. */

#define CVSREP_FIELDATTR "CVS/fileattr"
/* Prepare for a new directory with repository REPOS. If REPOS is NULL,
then prepare for a "non-directory"; the caller can call fileattr_write
and fileattr_free, but must not call fileattr_get or fileattr_set. */
extern void fileattr_startdir PROTO ((char *repos));

/* Get the attribute ATTRNAME for file FILENAME. The return value
points into memory managed by the fileattr routines, should not be
altered by the caller, and is only good until the next call to
fileattr_clear or fileattr_set. It points to the value, terminated
by '\0' or ':' . Return NULL if said file lacks said attribute.
If FILENAME is NULL, return default attributes (attributes for
files created in the future). */
extern char *fileattr_get PROTO ((const char *filename, const char *attrname));

/* Like fileattr_get, but return a pointer to a newly malloc'd string
terminated by '\0' (or NULL if said file lacks said attribute). */
extern char *fileattr_get0 PROTO ((const char *filename,
const char *attrname));

/* This is just a string manipulation function; it does not manipulate
file attributes as such.

LIST is in the format
ATTRNAME NAMEVALSEP ATTRVAL (ENTSEP ATTRNAME NAMEVALSEP ATTRVAL)
And we want to put in an attribute with name NAME and value VAL,
replacing the already-present attribute with name NAME if there is
one. Or if VAL is NULL remove attribute NAME. Return a new

malloc’d list; don’t muck with the one passed in. If we are removing the last attribute return NULL. LIST can be NULL to mean that we started out without any attributes.

Examples:

```c
fileattr_modify ("abc=def", "xxx", "val", ‘=’, ‘;’) => “abc=def;xxx=val"
fileattr_modify ("abc=def", "abc", "val", ‘=’, ‘;’) => “abc=val"
fileattr_modify ("abc=def=def=def", "abc", "val", ‘=’, ‘;’) => “abc=val"
fileattr_modify ("abc=def=def=def", "def", "val", ‘=’, ‘;’) => “abc=def=val;ghi=v3"
```

*/

extern char *fileattr_modifyPROTO (char *list, const char *attrname,
 const char *attrval, int namevalsep,
 int entsep);

}/* Set attribute ATTRNAME for file FILENAME to ATTRVAL. If ATTRVAL is NULL,
the attribute is removed. Changes are not written to disk until the next
call to fileattr_write. If FILENAME is NULL, set attributes for files
created in the future. If ATTRVAL is NULL, remove that attribute. */

extern void fileattr_setPROTO ((const char *filename,
 const char *attrname, const char *attrval));

}/* Get all the attributes for file FILENAME. They are returned as malloc’d
data in an unspecified format which is guaranteed only to be good for passing
to fileattr_setall, or NULL if no attributes. If FILENAME is NULL, get default attributes. */

extern char *fileattr_getallPROTO ((const char *filename));

}/* Set the attributes for file FILENAME to ATTRS, overwriting all previous
attributes for that file. ATTRS was obtained from a previous call to fileattr_getall (malloc’d data or NULL). */

extern void fileattr_setallPROTO ((const char *filename,
 const char *attrs));

}/* Set the attributes for file FILENAME in whatever manner is appropriate
for a newly created file. */

extern void fileattr_newfilePROTO ((const char *filename));

}/* Write out all modified attributes. */

extern void fileattr_writePROTO ( (void));

}/* Free all memory allocated by fileattr*. */

#define FILEATTRH 1

#endif /* fileattr.h */
A.23 filesubr.c

/* filesubr.c — subroutines for dealing with files
   Jim Blandy <jimb@cyclic.com>

This file is part of GNU CVS.

GNU CVS is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2, or (at your option) any
later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

These functions were moved out of subr.c because they need different
definitions under operating systems (like, say, Windows NT) with different
file system semantics.

#include "cvs.h"

static int deep_remove_dirPROTO((const char *path));

/* Copies "from" to "to".
 */
void copy_file((from), (to))
const char *from;
const char *to;
{
    struct stat sb;
    struct utimbuf t;
    int fdin, fdout;
    if (trace)
        #ifdef SERVER_SUPPORT
            (void) fprintf(stderr, "%c-> copy(%s,%s)\n", (server_active) ? 'S' : '', from, to);
        #else
            (void) fprintf(stderr, "-> copy(%s,%s)\n", from, to);
        #endif
    if (noexec)
        return;
    /* If the file to be copied is a link or a device, then just create
       the new link or device appropriately. */
    if (islink(from))
        { char source = xreadlink(from);
          symlink(source, to);
          free(source);
          return;
        }
    if (isdevice(from))
        { if (stat(from, &sb) < 0)
            error(1, errno, "cannot stat \%s", from);
            mknod(to, sb.st_mode, sb.st_dev);
        }
    else
        { /* Not a link or a device... probably a regular file. */
          if ((fdin = open(from, O_RDONLY)) < 0)
            error(1, errno, "cannot open \%s for copying", from);
          if (fstat(fdin, &sb) < 0)
            error(1, errno, "cannot fstat \%s", from);
          if ((fdout = creat(to, (int) sb.st_mode & 07777)) < 0)
            error(1, errno, "cannot create \%s for copying", to);
          if (sb.st_size > 0)
              { char buf[BUFSIZ];
                int n;
                for (; ;)
                    { n = read(fdin, buf, sizeof(buf));
                      if (n == -1)
                          break;
                      if (fwrite(buf, 1, n, fdout) != n)
                          error(1, errno, "cannot write \%s for copying", to);
                      if (fchmod(fdout, (int) sb.st_mode & 07777) < 0)
                          error(1, errno, "cannot chmod \%s", to);
                  }
            }
        }
If the write(fdout, buf, n) != n) {
    error (1, errno, "cannot write file %s for copying", to);
}

// if (xsync(fdout))
//    error (1, errno, "cannot fsync file %s after copying");

if (close(fdin) < 0)  
    error (0, errno, "cannot close %s");
if (close(fdout) < 0)  
    error (1, errno, "cannot close %s");

/* now, set the times for the copied file to match those of the original */
memset((char *) & t, 0, sizeof(t));
t.actime = sb.statime;
t.mtime = sb.statime;
t.mtimetime = sb.statime;
(void) utime(to, & t);

/* FIXME-krp: these functions would benefit from caching the char * & stat buf. */

int
isdir(file)
const char *file;
{
    struct stat sb;
    if (stat(file, & sb) < 0)
        return (0);
    return (S_ISDIR(sb.st_mode));
}

int
islink(file)
const char *file;
{
    struct stat sb;
    if (CVS_LSTAT(file, & sb) < 0)
        return (0);
    return (S_ISLNK(sb.st_mode));
}

int
isdevice(file)
const char *file;
{
    struct stat sb;
    if (CVS_LSTAT(file, & sb) < 0)
        return (0);
    // if (S_ISBLK(sb.st_mode))
    //     return 1;
    // else
    return (0);
}

int
isfile(file)
const char *file;
{
    struct stat sb;
    if (CVS_LSTAT(file, & sb) < 0)
        return (0);
    // if (S_ISBLK(sb.st_mode))
    //     return 1;
    // else
    return 0;
}
```c
{ return isaccessible(file, F_OK); }

/*
 * Returns non-zero if the argument file is readable.
 */
int isreadable (file)
    const char *file;
{ return isaccessible(file, R_OK); }

/*
 * Returns non-zero if the argument file is writable.
 */
int iswritable (file)
    const char *file;
{ return isaccessible(file, W_OK); }

/*
 * Returns non-zero if the argument file is accessible according to
 * mode. If compiled with SETXID_SUPPORT also works if cvs has setuid
 * bits set.
 */
int isaccessible (file, mode)
    const char *file;
    const int mode;
{ #ifdef SETXID_SUPPORT
    struct stat sb;
    int umask = 0;
    int gmask = 0;
    int omask = 0;
    int uid;
    if (stat(file, &sb) == -1)
        return 0;
    if (mode == F_OK)
        return 1;
    uid = geteuid();
    if (uid == 0) /* superuser */
        return 1;
    if (mode & X_OK)
    { umask = S_IRUSR;
        gmask = S_IWGRP;
        omask = S_IWOTH;
        if (mode & R_OK)
        { umask = S_IWUSR;
            gmask = S_IWGRP;
            omask = S_IWOTH;
        }
    }
    else
    { umask = S_IWUSR;
        gmask = S_IWGRP;
        omask = S_IWOTH;
    }
    if (mode & W_OK)
    { umask = S_IRUSR;
        if (ab.st_mode == umask)
            return (ab.st_mode & gmask) == gmask;
    else if (ab.st_gid == getegid())
        return (ab.st_mode & gmask) == gmask;
    else
        return (ab.st_mode & omask) == omask;
    }
    return access(file, mode) == 0;
#endif
}

/*
 * Open a file and die if it fails
 */
FILE *
open_file (name, mode)
```
const char *name;
const char *mode;

FILE *fp;
if ((fp = fopen (name, mode)) == NULL)
  error (1, errno, "cannot open %s", name);
return (fp);

/**
 * Make a directory and die if it fails
 */
void make_directory (name)
const char *name;
{
  struct stat sb;
  if (stat (name, &sb) == 0 && !S_ISDIR (sb.st_mode))
    error (1, errno, "cannot make directory %s", name);
}

/**
 * Make a directory and die if it fails
 */
void make_directories (name)
const char *name;
{
  char *cp;
  if (noexec)
    return;
  if (mkdir (name, 0777) == 0 || errno == EEXIST)
    return;
  if (!existence_error (errno))
    { error (0, errno, "cannot make path to %s", name);
      return;
    }
  if ((cp = strrchr (name, '/')) == NULL)
    return;
  *cp = '\0';
  make_directories (name);
  *cp = '\0';
  if (*cp == '/')
    return;
  (void) mkdir (name, 0777);
}

/** Create directory NAME if it does not already exist; fatal error for other errors. Returns 0 if directory was created; 1 if it already existed. */
int mkdir_if_needed (name)
char *name;
{
  if (mkdir (name, 0777) < 0)
    {
      if (errno == EEXIST)
        { error (1, errno, "cannot make directory %s", name);
          return 1;
        }
      return 0;
    }
}

/**
 * Change the mode of a file, either adding write permissions, or removing
 * all write permissions. Either change honors the current umask setting.
 * Don't do anything if PreservePermissions is set to 'yes'. This may
 * have unexpected consequences for some uses of chmod.
 */
void xchmod (fname, writable)
char *fname;
int writable;
{
  struct stat sb;
  mode_t mode, umask;
  if (preserve_perms)
    return;
...
if (stat (fname, &sb) < 0)
{
    if (!noexec)
    	    error (0, errno, "cannot stat %s", fname);
    return;
}
oumask = umask (0);
(void) umask (oumask);
if (writable)
{
    mode = sb.st_mode | ( businessmen mode ! 
                         & ((stat.mode & S_IRUSR) ? S_IWUSR : 0) 
                             | ((stat.mode & S_IRGRP) ? S_IWGRP : 0) 
                             | ((stat.mode & SYROTH) ? S_IWOTH : 0));
}
else
{
    mode = sb.st_mode & ~ (S_IWRITE | S_IWGRP | S_IWOTH) & ~ oumask;
}
if (trace)
#endif
if (chmod (fname, mode) < 0)
    error (0, errno, "cannot change mode of file %s", fname);
}
/∗ Rename a file and die if it fails ∗/
void
rename_file (from, to)
{
    if (trace)
#endif
    if (!noexec)
    return;
    if (rename (from, to) < 0)
        error (1, errno, "cannot rename file %s to %s", from, to);
}
/∗ Unlink a file, if possible. ∗/
int
unlink_file (f)
{
    if (trace)
#endif
    if (!noexec)
    return (0);
    return (unlink (f));
}
/∗ Unlink a file or dir, if possible. If it is a directory do a deep ∗ removal of all of the files in the directory. Return -1 on error ∗ (in which case errno is set). ∗/
int
unlink_file_dir (f)
{
    if (trace)
    struct stat sb;
    if
*/ This is called by the server parent process in contexts where it is not OK to send output (e.g. after we sent "ok" to the client). */

def sendf()
{
    (void) fprintf(stderr, "unlink_file_dir(%s)\n", f);

    if (noexec)
        return (0);

    /* For at least some unices, if root tries to unlink() a directory, instead of doing something rational like returning EISDIR, the system will gleefully go ahead and corrupt the filesystem. So we first call stat() to see if it is OK to call unlink(). This doesn't quite work — if someone creates a directory between the call to stat() and the call to unlink(), we'll still corrupt the filesystem. Where is the Unix Haters Handbook when you need it? */

    if (stat(f, &sb) < 0)
    {
        if (existence_error(errno))
        {
            /* The file or directory doesn't exist anyhow. */
            return -1;
        }
        else if (S_ISDIR(sb.st_mode))
            return deep_remove_dir(f);
    }

    return unlink(f);
}

/ * Remove a directory and everything it contains. Returns 0 for success, -1 for failure (in which case errno is set). */

static int deep_remove_dir(path)
const char *path;
{
    DIR *dirp;
    struct dirent *dp;

    if (rmdir(path) != 0)
    {
        if (errno == ENOTEMPTY
            || errno == EEXIST
            || Ugly workaround for ugly AIX 4.1 (and 3.2) header bug (it defines ENOTEMPTY and EEXIST to 17 but actually returns 87))
        {
            if ((dirp = opendir(path)) == NULL)
            { /* If unable to open the directory return * an error */
                return -1;
            }

            while ((dp = readdir(dirp)) != NULL)
            {
                char *buf;

                if (strcmp(dp->d_name, ".") == 0 ||
                    strcmp(dp->d_name, ".\nname", ".", ".") == 0)
                    continue;

                buf = xmalloc(strlen(path) + strlen(dp->d_name) + 5);
                sprintf(buf, "%s/%s\nname", path, dp->d_name);

                /* See comment in unlink_file_dir explanation of why we use isdir instead of just calling unlink and checking the status. */

                if (isdir(buf))
                {
                    if (deep_remove_dir(buf))
                    {
                        closedir(dirp);
                        free(buf);
                        return -1;
                    }
                }
                else
                {
                    if (unlink(buf) != 0)
                    {
                        closedir(dirp);
                        free(buf);
                        return -1;
                    }
                }
            }

            closedir(dirp);
            free(buf);
        }
    }

    return unlink(f);
}
free (buf);
    close(dirp);
    return rmdir (path);
  }
else
  return -1;
}

/* Was able to remove the directory return 0 */
return 0;

/* Read NCHARS bytes from descriptor FD into BUF.
   The number returned is always NCHARS unless end-of-file or error. */
static size_t block_read (fd, buf, nchars)
int fd;
char *buf;
size_t nchars;
{
  char *bp = buf;
  size_t nread;
  do
    { nread = read (fd, bp, nchars);
      if (nread == (size_t)−1)
        {#ifdef EINTR
          if (errno == EINTR)
            continue;
        #endif
          return (size_t)−1;
        }
      if (nread == 0)
        break;
    }
    while (nchars != 0);
  return bp − buf;
}

/* Compare "file1" to "file2". Return non-zero if they don't compare exactly.
   If FILE1 and FILE2 are special files, compare their salient characteristics
   (i.e. major/minor device numbers, links, etc.) */
int xcmp (file1, file2)
const char *file1;
const char *file2;
{
  char *buf1, *buf2;
  struct stat sb1, sb2;
  int fd1, fd2;
  int ret;
  if (CVS_LSTAT (file1, &sb1) < 0)
    error (1, errno, "cannot lstat %s", file1);
  if (CVS_LSTAT (file2, &sb2) < 0)
    error (1, errno, "cannot lstat %s", file2);
/* If FILE1 and FILE2 are not the same file type, they are unequal. */
if ((sb1.st_mode & S_IFMT) != (sb2.st_mode & S_IFMT))
  return 1;
/* If FILE1 and FILE2 are symlinks, they are equal if they point to the same thing. */
if (S_ISLNK (sb1.st_mode) && S_ISLNK (sb2.st_mode))
  { int result;
    buf1 = xreadlink (file1);
    buf2 = xreadlink (file2);
    result = (strcmp (buf1, buf2) == 0);
    free (buf1);
    free (buf2);
    return result;
  }
/* If FILE1 and FILE2 are devices, they are equal if their device numbers match. */
if (S_ISBLK (sb1.st_mode) || S_ISCHR (sb1.st_mode))
{  
  if (sb1.st_size == sb2.st_size)  
    return 0;  
  else  
    return 1;  
}  
if ((fd1 = open(file1, O_RDONLY)) < 0)  
  error (1, errno, "cannot open file %s for comparing", file1);  
if ((fd2 = open(file2, O_RDONLY)) < 0)  
  error (1, errno, "cannot open file %s for comparing", file2);  

/* A generic file compare routine might compare st_dev & st_ino here  
   to see if the two files being compared are actually the same file.  
   But that won't happen in CVS, so we won't bother. */  
if (sb1.st_size != sb2.st_size)  
  ret = 1;  
else if (sb1.st_size == 0)  
  ret = 0;  
else  
{  
  /* FIXME: compute the optimal buffer size by computing the least  
     common multiple of the files st_blocks field */  
  size_t buf_size = 8 * 1024;  
  size_t read1, size_t read2;  
  buf1 = xmalloc (buf_size);  
  buf2 = xmalloc (buf_size);  
  do  
  {  
    read1 = block_read (fd1, buf1, buf_size);  
    if (read1 == (size_t)-1)  
      error (1, errno, "cannot read file %s for comparing", file1);  
    read2 = block_read (fd2, buf2, buf_size);  
    if (read2 == (size_t)-1)  
      error (1, errno, "cannot read file %s for comparing", file2);  
    /* assert (read1 == read2); */  
    ret = memcmp (buf1, buf2, read1);  
  } while (ret == 0 && read1 == buf_size);  
  free (buf1);  
  free (buf2);  
}  
(void) close (fd1);  
(void) close (fd2);  
return (ret);  
}  

/* Generate a unique temporary filename. Returns a pointer to a newly  
   malloc'd string containing the name. Returns successfully or not at  
   all. */  
/* There are at least three functions for generating temporary  
   filenames. We use tempnam (SVID 3) if possible, else mktemp (BSD  
   4.3), and as last resort tmpnam (POSIX). Reason is that tempnam and  
   mktemp both allow to specify the directory in which the temporary  
   file will be created. */  
#define HAVE_TEMPNAM  
char *  
  cvttemp_name ()  
{  
  char *retval;  
  retval = tempnam (Tmpdir, "cvs");  
  if (retval == NULL)  
    error (1, errno, "cannot generate temporary filename");  
  /* tempnam returns a pointer to a newly malloc'd string, so there's  
     no need for a xstrdup */  
  return retval;  
}  
#endif  
#define HAVE_MKTEMP  
char *  
  cvttemp_name ()  
{  
  char *retval;  
  value = xmalloc (strlen (Tmpdir) + 40);  
  sprintf (value, "%s/XXXXX", Tmpdir, "cvsXXXXX");  
  retval = mktemp (value);  
  if (retval == NULL)  
    error (1, errno, "cannot generate temporary filename");
return value;

720  # else
721         char value[14 + tmpnam + 1];
722         char *retval;
723         retval = tmpnam (value);
724         if (retval == NULL)
725             error (1, errno, "cannot generate temporary filename");
726         return xstrdup (value);
727     # endif
728 }
729
730 #endif
731
732 /* Return non-zero iff FILENAME is absolute,
733    Toplevel under Unix, but more complicated under other systems. */
734 int
735 isabsolute (filename)
736     const char *filename;
737 {
738     return filename[0] == '/';
739 }
740
741 /*
742 * Return a string (dynamically allocated) with the name of the file to which
743 * LINK is symlinked.
744 */
745 char *
746 xreadlink (link)
747     const char *link;
748 {
749     char *file = NULL;
750     char *tfile;
751     int buflen = 128;
752     int link_name_len;
753     if (!islink (link))
754         return NULL;
755     do
756         {
757             file = xrealloc (file, buflen);
758             link_name_len = readlink (link, file, buflen - 1);
759             buflen *= 2;
760         } while (link_name_len < 0 && errno == ENAMETOOLONG);
761     if (link_name_len < 0)
762         error (1, errno, "cannot readlink %s", link);
763     file[link_name_len] = '\0';
764     tfile = xstrdup (file);
765     free (file);
766     return tfile;
767 }
768
769 /* Return a pointer into PATH’s last component. */
770 char *
771 last_component (path)
772     char *path;
773 {
774     char *last = strrchr (path, '/');
775     if (last && (last != path))
776         return last + 1;
777     else
778         return path;
779 }
780
781 /* Return the home directory. Returns a pointer to storage
782  managed by this function or its callees (currently getenv).
783  This function will return the same thing every time it is
784  called. Returns NULL if there is no home directory.
785 
786  Note that for a server server, this may return root’s home
787  directory. What typically happens is that upon being started from
788  inetd, before switching users, the code in cvsrc.c calls
789  gethomedir which remembers root’s home directory in the static
790  variable. Then the switch happens and gethomedir might return a
791  directory that we don’t even have read or execute permissions for
792  (which is bad, when various parts of CVS try to read there). One
793  fix would be to make the value returned by gethomedir only good
794  until the next call (which would free the old value). Another fix
795  would be to just always malloc our answer, and let the caller free
796  it (that is best, because someday we may need to be reentrant).
The workaround is to put `-f` in `inetd.conf` which means that `gethomedir` won't get called until after the switch in user ID.

The whole concept of a “home directory” on the server is pretty iffy, although I suppose some people probably are relying on it for `.cvs` and such, in the cases where it works.

```c
char *gethomedir()
{
    static char *home = NULL;
    char *env = getenv("HOME");
    struct passwd *pw;
    if (home != NULL)
        return home;
    if (env)
        home = env;
    else if ((pw = (struct passwd*) getpwnam(getuid())))
        home = xstrdup(pw->pw_dir);
    else
        return 0;
    return home;
}
```

*/ See cvs.h for description. On unix this does nothing, because the shell expands the wildcards. */

```c
void expand_wild(argc, argv, pargc, pargv)
{
    int argc;
    char **argv;
    int **pargc;
    char ***pargv;
    { int i;
        *pargc = argc;
        *argv = argv;
        *pargv = (char**) xmalloc(argc * sizeof(char*));
        for (i = 0; i < argc; ++i)
            (*pargv)[i] = xstrdup(argv[i]);
    }
}
```

```c
#define SERVER_SUPPORT
/*@ Case-insensitive string compare. I know that some systems have such a routine, but I'm not sure I see any reasons for dealing with the hair of figuring out whether they do (I haven't looked into whether this is a performance bottleneck; I would guess not). */

```c
int cvscasecmp(str1, str2)
{ char *str1;
  char *str2;
  { char *p;
    char *q;
    int pqdiff;
    p = str1;
    q = str2;
    while (pqdiff = tolower(*p) - tolower(*q)) == 0)
    { if (*p == '\0')
        return 0;
        ++p;
        ++q;
    } return pqdiff;
}
```

```c
int fopen_case(name, mode, fp, pathp)
{ char *name;
  char *mode;
  FILE **fp;
  char **pathp;
  { struct dirent *dp;
```
DIR *dirp;
char *dir;
char *fname;
char *found_name;
int retval;

/* Separate NAME into directory DIR and filename within the directory FNAME. */
dir = xstrdup (name);
fname = strrchr (dir, '/');
if (fname == NULL)
    error (1, 0, "internal error: relative pathname in fopen_case");
*fname++ = '\0';
found_name = NULL;
dirp = CVS_OPENDIR (dir);
if (dirp == NULL)
{
    if (existence_error (errno))
    {
        /* This can happen if we are looking in the Attic and the Attic
directory does not exist. Return the error to the caller;
they know what to do with it. */
        retval = errno;
        goto out;
    }
    else
    {
        /* Give a fatal error; that way the error message can be
more specific than if we returned the error to the caller. */
        error (1, errno, "cannot read directory %s", dir);
    }
}
errno = 0;
while ((dp = readdir (dirp)) != NULL)
{
    if (cvs_casecmp (dp->d_name, fname) == 0)
    {
        if (found_name == NULL)
            error (1, 0, "%s is ambiguous; could mean %s or %s",fname, dp->d_name, found_name);
        found_name = xstrdup (dp->d_name);
    }
    if (errno != 0)
        error (1, errno, "cannot read directory %s", dir);
closedir (dirp);
    if (found_name == NULL)
    {
        *fp = NULL;
        retval = ENOENT;
    }
    else
    {
        char *p;
        /* Copy the found name back into DIR. We are assuming that
found_name is the same length as fname, which is true as
long as the above code is just ignoring case and not other
aspects of filename syntax. */
        p = dir + strlen (dir);
        *p++ = '/';
        strcpy (p, found_name);
        *fp = fopen (dir, mode);
        if (*fp == NULL)
            retval = errno;
        else
            retval = 0;
    }
}
if (pathp == NULL)
    free (dir);
else if (retval != 0)
    free (dir);
else
    *pathp = dir;
free (found_name);
out:
return retval;

sendif /* SERVER_SUPPORT */
A.24  find_names.c

#include "cvs.h"

static int find_dirsPROTO((char *dir, List *list, int checkadm),
static int find_rcsPROTO((char *dir, List *list));
static int add_subdir_procPROTO(Node *, void *);
static int register_subdir_procPROTO(Node *, void *);
static List *filelist;

add the key from entry on entries list to the files list
static int add_entriesprocPROTO((node, closure))
Node *node;
void *closure;
{
Entnode *entnode = (Entnode *)node->data;
if (entnode->type != ENT_FILE) return (0);

fnode = getnode();
fnode->type = FILES;
fnode->key = xstrdup (node->key);
if (addnode (filelist, fnode) != 0) freenode (fnode);
return (0);
}

List *FindNames (repository, which, aflag, optentries)
char *repository;
int which;
int aflag;
List **optentries;
{
List *entries;
List *files;
/* make a list for the files */
files = filelist = getlist ();
/* look at entries (if necessary) */
if (which & W_LOCAL) {
    /* parse the entries file (if it exists) */
    entries = Entries_Open (aflag, NULL);
    if (entries != NULL) {
        /* walk the entries file adding elements to the files list */
        (void) walklist (entries, add_entriesproc, NULL);
        /* if our caller wanted the entries list, return it; else free it */
        if (optentries != NULL) *optentries = entries;
        else
            Entries_Close (entries);
    }
}

if ((which & W_REPOS) && readable (CVSADM_ENTSTAT)) {
    /* search the repository */
    if (find_rcs (repository, files) != 0)
        error (1, errno, "cannot open directory %s", repository);
}
/∗ search the attic too ∗/
if (which & W_ATTIC)
{
    char *dir;
    dir = xmalloc (strlen (repository) + sizeof (CVSATTIC) + 10);
    (void) sprintf (dir, "%s", repository, CVSATTIC);
    (void) findw (dir, files);
    free (dir);
}

/* sort the list into alphabetical order and return it */
sortlist (files, fsortcmp);
return (files);

/* Add an entry from the subdirs list to the directories list. This
* is called via walklist. */
static int
add_subdir_proc (p, closure)
    Node *p;
    void *closure;
{
    List *dirlist = (List *) closure;
    Entnode *entnode = (Entnode *) p->data;
    if (entnode->type != ENT_SUBDIR)
        return 0;
    dnode = getnode ();
    dnode->type = DIRS;
    dnode->key = xstrdup (entnode->user);
    if (addnode (dirlist, dnode) != 0 )
        freenode (dnode);
    return 0;
}

/* Register a subdirectory. This is called via walklist. */
#define ArgsUsed
static int
register_subdir_proc (p, closure)
    Node *p;
    void *closure;
{
    List *entries = (List *) closure;
    SubdirRegister (entries, (char *) NULL, p->key);
    return 0;
}

/* create a list of directories to traverse from the current directory */
List *
FindDirectories (repository, which, entries)
    char *repository;
    int which;
    List *entries;
{
    List *dirlist;
    /* make a list for the directories */
    dirlist = getlist ();
    /* find the local ones */
    if (which & W_LOCAL)
    {
        List *tmpentries;
        struct stickydirtag *sdtp;
        /* Look through the Entries file. */
        if (entries != NULL)
            tmpentries = entries;
        else if (isfile (CVSADM_ENT))
            tmpentries = EntriesOpen (0, NULL);
        else
            tmpentries = NULL;
        if (tmpentries != NULL)
            sdtp = (struct stickydirtag *) tmpentries->list->data;
/* If we do have an entries list, then if sdtp is NULL, or if
sdtp->subdirs is non-zero, all subdirectory information is
recorded in the entries list. */
if (tmpentries != NULL && (sdtp == NULL || sdtp->subdirs))
    walklist (tmpentries, add_subdir, proc, (void *) dirlist);
else
    {
    /* This is an old working directory, in which subdirectory
    information is not recorded in the Entries file. Find
    the subdirectories the hard way, and, if possible, add
    it to the Entries file for next time. */
    */
    /* FIXME-maybe: find_subdirs is hope for this usage because
    it skips CVSATTIC and CVSLCK directories-those names
    should be special only in the repository. However, in
    the interests of not perturbing this code, we probably
    should leave well enough alone unless we want to write
    a sanity.sh test case (which would operate by manually
    hacking on the CVS/Entries file). */
    if (find_wdirs (*, 1), dirlist, 1, tmpentries) != 0)
        error (1,errno, "cannot open current directory");
    if (tmpentries != NULL)
        {
        if (! (list_isempty (dirlist))
            walklist (dirlist, register_subdir, proc,
                        (void *) tmpentries);
        else
            Subdirs_Known (tmpentries);
        }
    }
    if (entries == NULL && tmpentries != NULL)
        Entries_Close (tmpentries);
    } /* look for sub-dirs in the repository */
    if (((which & W_REPO) && repository) &&
    /* search the repository */
    if (find_wdirs (repository, dirlist, 0, entries) != 0)
        error (1,errno, "cannot open directory", repository);
    /* We don't need to look in the attic because directories
    never go in the attic. In the future, there hopefully will
    be a better mechanism for detecting whether a directory in
    the repository is alive or dead; it may or may not involve
    moving directories to the attic. */
    }
    } /* sort the list into alphabetical order and return st */
    sortlist (dirlist, fsortcmp);
    return (dirlist);
}

/* Finds all .v files in the argument directory, and adds them to the
* files list. Returns 0 for success and non-zero if the argument directory
* cannot be opened.
* */
static int
find_wses (dir, list)
    char *dir;
    List *list;
{
    Node *p;
    struct dirent *dp;
    DIR *dirp;
    /* set up to read the dir */
    if ((dirp = CVS_OPENDIR (dir)) == NULL)
        return (1);
    /* read the dir, grabbing the .v files */
    while ((dp = readdir (dirp)) != NULL)
    {
        if (CVS_FNMATCH (RCSPAT, dp->d_name, 0) == 0)
            {
            char *comma;
            comma = strchr (dp->d_name, ','); /* strip the .v */
            *comma = '\0';
            p = getnode (
                        dp->d_name, NULL);
            p->type = FILES;
            p->key = xstrdup (dp->d_name);
            if (addnode (list, p) != 0)
                freenode (p);
            }
        }
(void) closedir (dirp);
return (0);
}

/*
 * Finds all the subdirectories of the argument dir and adds them to
 * the specified list. Subdirectories without a CVS administration
 * directory are optionally ignored. If ENTRIES is not NULL, all
 * files on the list are ignored. Returns 0 for success or 1 on
 * error.
 */

static int
finddirs (dir, list, checkadm, entries)
char *dir;
List *list;
int checkadm;
List *entries;
{
Node *p;
char *tmp = NULL;
size_t tmp_size = 0;

struct dirent *dp;
DIR *dirp;
int skip_emptydir = 0;

/* First figure out whether we need to skip directories named
 Emptydir. Except in the CVSNULLREPOS case, Emptydir is just
 a normal directory name. */
if (isabsolute (dir))
  & & strncmp (dir, CVSroot_directory, strlen (CVSroot_directory)) == 0
  & & ISDIRSEP (dir[strlen (CVSroot_directory)])
  & & strcmp (dir + strlen (CVSroot_directory) + 1, CVSROOTADM) == 0
    skip_emptydir = 1;

/* set up to read the dir */
if ((dirp = CVS_deopen (dir)) == NULL)
  return (1);

/* read the dir, grabbing sub-dirs */
while ((dp = readdir (dirp)) != NULL)
{
  if (strcmp (dp->d_name, ".") == 0 ||
      strcmp (dp->d_name, ".") == 0 ||
      strcmp (dp->d_name, CVSIXTTIC) == 0 ||
      strcmp (dp->d_name, CVSLOCK) == 0 ||
      strcmp (dp->d_name, CVSREP) == 0)
    continue;

  /* findnode() is going to be significantly faster than stat()
   because it involves no system calls. That is why we bother
   with the entries argument, and why we check this first. */
  if (entries != NULL & & findnode (entries, dp->d_name) != NULL)
    continue;

  if (skip_emptydir
    & & strcmp (dp->d_name, CVSNULLREPOS) == 0)
    continue;

  ifdef DT_DIR
    if (dp->d_type != DT_DIR)
      continue;
  endif

  ifdef DT_LNK
    if (dp->d_type != DT_LNK)
      continue;
  endif

  ifdef DT_UNS
    if (dp->d_type != DT_UNS)
      continue;
  endif

  printf (tmp, "/%s/%s", dir, dp->d_name);
}

/* check for administration directories (if needed) */
if (checkadm)
{
  /* blow off symbolic links to dirs in local dir */
  ifdef DT_DIR
    if (dp->d_type != DT_DIR)
      continue;
  endif

  ifdef DT_LNK
    if (dp->d_type != DT_LNK)
      continue;
  endif

  ifdef DT_UNS
    if (dp->d_type != DT_UNS)
      continue;
  endif
Note that we only get here if we already set tmp above. */
if (islink (tmp)) continue;
#endif DT_DIR}
#endif

/* check for new style */
expand_string (&tmp, &tmp->size, 
(strlen (dir) + strlen (dp->d_name) + sizeof (CVSADM) + 10));
(void) sprintf (tmp, "%s/%s/%s", dir, dp->d_name, CVSADM);
if (!isdir (tmp)) continue;

/* put it in the list */
p = getnode (1);
p->type = DT_DIR;
p->key = xstrdup (dp->d_name);
if (addnode (list, p) != 0)
free (p);
(void) closedir (dirp);
if (tmp != NULL)
free (tmp);
return (0);
}
A.25 hardlink.c

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

1. Collect and manage hardlink info associated with a particular file.

```c
#include "cvs.h"
#include "hardlink.h"

/* The structure currently used to manage hardlink info is a list. Therefore, most of the functions which manipulate hardlink data are walklist procedures. This is not a very efficient implementation; if someone decides to use a real hash table (for instance), then much of this code can be rewritten to be a little less arcane.

Each element of 'hardlist' represents an inode. It is keyed on the inode number, and points to a list of files. This is to make it easy to find out what files are linked to a given file FOO: find FOO's inode, look it up in hardlist, and retrieve the list of files associated with that inode.

Each file node, in turn, is represented by a 'hardlink info' struct, which includes 'status' and 'links' fields. The 'status' field should be used by a procedure like commit_fileproc or update_fileproc to record each file's status; that way, after all file links have been recorded, CVS can check the linkage of files which are in doubt (i.e. NEEDS_MERGE files).

TODO: a diagram of an example hardlist would help here. */

List *hardlist;  /* Record hardlink information for working files */
char *working_dir;  /* The top-level working directory, used for constructing full pathnames. */

/* Return a pointer to FILEPATH's node in the hardlist. This means looking up its inode, retrieving the list of files linked to that inode, and then looking up FILE in that list. If the file doesn't seem to exist, return NULL. */
Node *lookup_file_by_inode (filepath)
{
  const char *filepath;
  struct stat sb;
  Node *hp, *p;

  /* Get file's basename, so that we can stat it. */
  file = strrchr (filepath, '/');
  if (file)
    ++file;
  else
    file = (char *)filepath;

  /* inodestr contains the hexadecimal representation of an inode, so it requires two bytes of text to represent each byte of the inode number. */
  inodestr = (char *)xmalloc (2*sizeof(unsigned long)+sizeof(char)+1);
  if (stat (file, &sb) < 0)
    {
      if (errno == ENOENT)
        {
          /* The file doesn't exist; we may be doing an update on a file that's been removed. A non-existent file has no link information, so return without changing hardlist. */
          free (inodestr);
          return NULL;
        }
      error (1, errno, "cannot stat file", file);
    }

  sprintf (inodestr, "0x%lx", (unsigned long)sb.st_ino);

  /* Find out if this inode is already in the hardlist, adding a new entry to the list if not. */
  hp = findnode (hardlist, inodestr);
  if (hp == NULL)
    {
      hp = getnode ();
      hp->type = UNKNOWN;
```
hp->key = inodestr;
hp->data = (char *) getlist();
hp->delproc = dellist;
    (void) addnode (hardlist, hp);
} else
    { free (inodestr);
    }

p = findnode ((List *) hp->data, filepath);

if (p == NULL)
    { p = getnode();
      p->type = UNKNOWN;
      p->key = xstrdup (filepath);
      p->data = NULL;
      (void) addnode ((List *) hp->data, p);
    }

return p;

/* After a file has been checked out, add a node for it to the hardlist
    (if necessary) and mark it as checked out. */

void update_hardlink_info (file)
    const char *file;
{
    char *path;
    Node *n;

    struct hardlink_info *hlinfo;

    if (file[0] == '/')
        { path = xstrdup (file);
        } else
        {
            /* file is a relative pathname; assume it's from the current
                working directory. */

    char *dir = xgetwd();
    path = xmalloc (sizeof(char) * (strlen(dir) + strlen(file) + 2));
    sprintf (path, "%s/%s", dir, file);
    free (dir);
    }

    n = lookup_file_by_inode (path);
    if (n == NULL)
        { /* Something is really wrong if the file doesn't exist here;
            update_hardlink_info should be called only when a file has
            just been checked out to a working directory. */
            error (1, 0, "lost hardlink info for %s", file);
        }

    if (n->data == NULL)
        n->data = (char *) xmalloc (sizeof (struct hardlink_info));
    hlinfo = (struct hardlink_info *) n->data;
    hlinfo->status = T_UNDEF;
    hlinfo->checked_out = 1;

    return;
}

List *linked_files_on_disk (file)
    char *file;
{
    char *inodestr, *path;
    struct stat sb;
    Node *n;

    /* If hardlist is NULL, we have not been doing an operation that
       would permit us to know anything about the file's hardlinks
       (cvs update, cvs commit, etc). Return an empty list. */

    if (hardlist == NULL)
        return getlist();

    /* Get the full pathname of file (assuming the working directory) */
    if (file[0] == '/')
        path = xstrdup (file);
    else
        { char *dir = xgetwd();
        }
path = (char *) xmalloc (strlen(dir) * 3);  
strncpy (path, "\%s\%s", dir, file);  
free (dir);  
}  
/* We do an extra lookup file here just to make sure that there  
is a node for 'path' in the hardlist. If that were not so,  
comparing the working directory linkage against the repository  
linkage for a file would always fail. */  
(void) lookupinode (inode, path);  
}  
if (stat (path, &nb) < 0)  
error (1, errno, "cannot stat \%s", file);  
/* inodestr contains the hexadecimal representation of an  
ode, so it requires two bytes of text to represent  
each byte of the inode number. */  
inodestr = (char *) xmalloc ((2*strlen((ino_t)=sizeof(char)) + 1));  
strftime (inodestr, "\%lx", sb1,ino);  
/* Make sure the files linked to this inode are sorted. */  
n = findnode (hardlist, inodestr);  
sortlist ((List *) n)->data, fsortcmp);  
free (inodestr);  
return (list *) n->data;  
}  
/* Compare the files in the 'key' fields of two lists, returning 1 if  
the lists are equivalent and 0 otherwise. */  
only the basenames of each file are compared. This is an awful hack  
that exists because list_linked files in hardlink returns full paths  
and the 'hardlinks' structure of an RCS Yers node contains only  
basenames. That in turn is a result of the awful hack that only  
basenames are stored in the RCS file. If anyone ever solves the  
problem of correctly managing cross-directory hardlinks, this  
function (along with most functions in this file) must be fixed. */  
int  
comparelist (links1, links2)  
List *links1,  
List *links2;  
{  
Node *n1, *n2;  
char *p1, *p2;  
sortlist (links1, fsortcmp);  
sortlist (links2, fsortcmp);  
n1 = links1->list->next;  
n2 = links2->list->next;  
while (n1 != links1->list && n2 != links2->list)  
{  
/* Get the basenames of both files. */  
p1 = strchr (n1->key, '/');  
if (p1 == NULL)  
p1 = n1->key;  
else  
++p1;  
P2 = strchr (n2->key, '/');  
if (p2 == NULL)  
p2 = n2->key;  
else  
++p2;  
/* Compare the files' basenames. */  
if (strcmp (p1, p2) != 0)  
return 0;  
n1 = n1->next;  
n2 = n2->next;  
}  
/* At this point we should be at the end of both lists; if not,  
one file has more links than the other, and return 1. */  
return (n1 == links1->list && n2 == links2->list);  
}  
/* Find a checked-out file in a list of filenames. Used by RCS_checkout  
when checking out a new hardlinked file, to decide whether this file  
can be linked to any others that already exist. The return value  
is not currently used. */  
int  
findcheckedout (proc (node, data)  
Node *node;  
void *data;  
...
```c
{  
    Node **uptodate = (Node **) data;
    Node *link;
    char *dir = xgetwd();
    char *path;
    struct hardlink_info *hlinfo;

    /* If we have already found a file, don’t do anything. */
    if (uptodate != NULL)
        return 0;

    /* Look at this file in the hardlist and see whether the checked_out
     field is 1, meaning that it has been checked out during this CVS run. */
    path = (char *) xmalloc(sizeof(char) * (strlen(dir) + strlen(node->key) + 2));
    sprintf(path, "%s/%s", dir, node->key);
    link = lookup_file_by_inode(path);
    free(path);
    free(dir);

    if (link == NULL)
    {
        /* We haven’t seen this file – maybe it hasn’t been checked
         out yet at all. */
        return 0;
    }

    hlinfo = (struct hardlink_info *) link->data;
    if (hlinfo->checked_out)
    {
        /* This file has been checked out recently, so it’s safe to
         link to it. */
        *uptodate = link;
    }

    return 0;
}
```
A.26 hardlink.h

/* This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

/* Data type definitions and declarations for hardlink management. */

/* This file should be #included in CVS source files after cvs.h
since it relies on types and macros defined there. */

/* The ‘checked_out’ member of a hardlink_info struct is used only
when files are being checked out or updated. It is used only when
hardlinked files are being checked out. */

struct hardlink_info
{
    Ctype status;          /* as returned from Classify_File() */
    int checked_out;       /* has this file been checked out lately? */
};

extern List *hardlist;
extern char *working_dir;

Node *lookup_file_by_inode PROTO ((const char *));
void update_hardlink_info PROTO ((const char *));
List *list_linked_files PROTO ((char *));
int compare_linkage_lists PROTO ((List *, List *));
int find_checkedout_proc PROTO ((Node *, void *));
A.27 hash.c

#include "cvs.h"
#include <assert.h>

/*
* Copyright (c) 1992, Brian Berliner and Jeff Polk
* You may distribute under the terms of the GNU General Public License as
* specified in the README file that comes with the CVS source distribution.
* Polk's hash list manager. So cool.
*/

static List *listcache = NULL;
static Node *nodecache = NULL;

static void freenode(memPROTO(Node *p));

static void dellist(List *listp)
{
  int i;
  List *list;
  Node *node;

  if (listcache != NULL)
  {
    /∗ get a list from the cache and clear it ∗/
    list = listcache;
    listcache = listcache->next;
    list->next = (List *) NULL;
  }

  else
  {
    /∗ make a new list from scratch ∗/
    list = (List *) xmalloc (sizeof (List));
    memset ((char *) list, 0, sizeof (List));
    node = getnode();
    list->list = node;
    node->type = HEADER;
    node->next = node->prev = node;
    return (list);
  }

  /∗ free up a list ∗/
}

/*
* create a new list (or get an old one from the cache)
*/
List *getlist()
{
  int i;
  List *list;
  Node *node;

  if (listcache != NULL)
  {
    /∗ get a list from the cache and clear it ∗/
    list = listcache;
    listcache = listcache->next;
    list->next = (List *) NULL;
  }

  else
  {
    /∗ make a new list from scratch ∗/
    list = (List *) xmalloc (sizeof (List));
    memset ((char *) list, 0, sizeof (List));
    node = getnode();
    list->list = node;
    node->type = HEADER;
    node->next = node->prev = node;
    return (list);
  }

  /∗ free up a list ∗/
}

/*
* hash function
*/
static int hashp(const char *key);

{ unsigned int h = 0;
  unsigned int g;

  assert(key != NULL);

  while (*key != 0)
  {
    unsigned int c = *key++;
    /* The FOLD_FN_C CHAR is so that findnode_fn works. */
    h = (h << 4) + FOLD_FN_C CHAR (c);
    if ((g = h & 0xf0000000) != 0)
      h = (h ^ (g >> 24)) ^ g;
  }

  return (h % HASHSIZE);
}

/*
* Global caches. The idea is that we maintain a linked list of "free"d
* nodes or lists, and get new items from there. It has been suggested to
* use an obstack instead, but off the top of my head, I’m not sure
* that would gain enough to be worth worrying about. ∗/
*/
static List *listcache = NULL;
static Node *nodecache = NULL;

static void freenode(memPROTO(Node *p));

int hashp(const char *key);

{ unsigned int h = 0;
  unsigned int g;

  assert(key != NULL);

  while (*key != 0)
  {
    unsigned int c = *key++;
    /* The FOLD_FN_C CHAR is so that findnode_fn works. */
    h = (h << 4) + FOLD_FN_C CHAR (c);
    if ((g = h & 0xf0000000) != 0)
      h = (h ^ (g >> 24)) ^ g;
  }

  return (h % HASHSIZE);
}

/*
* create a new list (or get an old one from the cache)
*/
List *getlist()
{
  int i;
  List *list;
  Node *node;

  if (listcache != NULL)
  {
    /∗ get a list from the cache and clear it ∗/
    list = listcache;
    listcache = listcache->next;
    list->next = (List *) NULL;
  }

  else
  {
    /∗ make a new list from scratch ∗/
    list = (List *) xmalloc (sizeof (List));
    memset ((char *) list, 0, sizeof (List));
    node = getnode();
    list->list = node;
    node->type = HEADER;
    node->next = node->prev = node;
    return (list);
  }

  /∗ free up a list ∗/
}

/*
* hash function
*/
static int hashp(const char *key);

{ unsigned int h = 0;
  unsigned int g;

  assert(key != NULL);

  while (*key != 0)
  {
    unsigned int c = *key++;
    /* The FOLD_FN_C CHAR is so that findnode_fn works. */
    h = (h << 4) + FOLD_FN_C CHAR (c);
    if ((g = h & 0xf0000000) != 0)
      h = (h ^ (g >> 24)) ^ g;
  }

  return (h % HASHSIZE);
}
p = (*listp)→list;

/* free each node in the list (except header) */
while (p→next != p)
delnodex (p→next);

/* free any list-private data, without freeing the actual header */
freeopmem (p);

/* free up the header nodes for hash lists (if any) */
for (i = 0; i < HASHSIZE; i++)
{
    if ((p = (*listp)→hasharray[i]) != (Node *) NULL)
    {
        /* put the nodes into the cache */
ifndef NOCACHE
        p→type = UNKNOWN;
p→next = nodecache;
nodecache = p;
#else
        /* If NOCACHE is defined we turn off the cache. This can make
         * get easier to tools to determine where items were allocated
         * and freed, for tracking down memory leaks and the like. */
        free (p);
#endif
    }
}

/* put it on the cache */
ifndef NOCACHE
(*listp)→next = listcache;
#else
free ((*listp)→list);
free (*listp);
#endif
*listp = (List *) NULL;

/* */
/* get a new list node */
Node *
getnode ()
{
    Node *p;
    if (nodecache != (Node *) NULL)
    {
        /* get one from the cache */
p = nodecache;
        nodecache = p→next;
    }
    else
    {
        /* make a new one */
p = (Node *) xmalloc (sizeof (Node));
    }
    /* always make it clean */
    memset ((char *) p, 0, sizeof (Node));
p→type = UNKNOWN;
    return (p);
}

/* */
/* remove a node from it's list (maybe hash list too) and free it */
void
delnodex (p)
{
    Node *p;
    if (p == (Node *) NULL)
    return;
    /* take it out of the list */
p→next→prev = p→prev;
p→prev→next = p→next;
    /* if it was hashed, remove it from there too */
if (p→hashnext != (Node *) NULL)
    {p→hashnext→hashprev = p→hashprev;
p→hashprev→hashnext = p→hashnext;
}
    /* free up the storage */
freeopmem (p);
}
static void freenode_mem(Node *p) {
  if (p->delproc != (void (*)(void)) NULL) 
    p->delproc(p);            /* call the specified delproc */
  else {
    if (p->data != NULL)     /* otherwise free() it if necessary */
      free(p->data);
    if (p->key != NULL)      /* free the key if necessary */
      free(p->key);
    /* to be safe, re-initialize these */
    p->key = p->data = (char *) NULL;
    p->delproc = (void (*)(void)) NULL;
  }
}

void freenode(Node *p) {
  /* first free the memory */
  freenode_mem(p);
  /* then put it in the cache */

#ifndef NOCACHE
  p->type = UNKNOWN;
  p->next = nodecache;
  nodecache = p;
#else
  free(p);
#endif
}

int insert_before(List *list, Node *marker, Node *p) {
  if (p->key != NULL)        /* hash it too? */
    hashval = hashp (p->key);
  if (list->hasharray[hashval] == NULL) /* make a header for list? */
    { q = getnode();
      q->type = HEADER;
      list->hasharray[hashval] = q->hashnext = q->hashprev = q;
    }
  /* put it into the hash list if it’s not already there */
  for (q = list->hasharray[hashval]; q != p->hashnext;
    if ((strcmp (p->key, q->key)) == 0)
      return (-1);
  q->hashprev = q->hashnext = p;
  p->hashprev = q;
  p->hashnext = q;
  /* to be safe, re-initialize these */
  p->hashprev = p->hashnext = NULL;
  list->hashprev = list->hashnext = NULL;
  return (0);
}

/∗
  * Insert item p at end of list “list” (maybe hash it too) if hashing and it
  * already exists, return -1 and don’t actually put it in the list
  * ∗ return 0 on success
∗/
int addnode (list, p)
 List ∗list;
 Node ∗p;
{
    return insert before (list, list−>list, p);
}

/∗ Like addnode, but insert p at the front of ‘list’. This boggosity is
∗ necessary to preserve last-to-first output order for some RCS functions.
∗/
int addnode at front (list, p)
{
    return insert before (list, list−>list−>next, p);
}

/∗ Look up an entry in hash list table and return a pointer to the
node. Return NULL if not found. Abort with a fatal error for
errors. ∗/
Node ∗findnode (list, key)
 List ∗list;
 const char ∗key;
{
    Node ∗head, ∗p;

    /∗ This probably should be “assert (list != NULL)” (or if not we
    should document the current behavior), but only if we check all
    the callers to see if any are relying on this behavior. ∗/
    if (!list == (List ∗) NULL)
        return ((Node ∗) NULL);

    assert (key != NULL);

    head = list−>hasharray[hashp (key)];
    if (head == (Node ∗) NULL)
        return ((Node ∗) NULL);

    for (p = head−>hashnext; p != head; p = p−>hashnext)
        if (strcmp (p−>key, key) == 0)
            return (p);

    return ((Node ∗) NULL);
}

/∗ Like addnode, but for a filename.
∗/
Node ∗findnode fn (list, key)
{
    Node ∗head, ∗p;

    /∗ This probably should be “assert (list != NULL)” (or if not we
    should document the current behavior), but only if we check all
    the callers to see if any are relying on this behavior. ∗/
    if (!list == (List ∗) NULL)
        return ((Node ∗) NULL);

    assert (key != NULL);

    head = list−>hasharray[hashp (key)];
    if (head == (Node ∗) NULL)
        return ((Node ∗) NULL);

    for (p = head−>hashnext; p != head; p = p−>hashnext)
        if (fncmp (p−>key, key) == 0)
            return (p);

    return ((Node ∗) NULL);
}

/* walk a list with a specific proc */
int walklist (list, proc, closure)
 List ∗list;
308 • Source code

```c
int (*proc) PROTO ((Node *, void *));
void *closure;
{
    Node *head, *p;
    int err = 0;
    if (list == NULL)
        return (0);
    head = list->list;
    for (p = head->next; p != head; p = p->next)
        err += proc (p, closure);
    return (err);
}
int list_empty (list)
List *list;
{
    return list == NULL || list->list->next == list->list;
}
static int (*client_comp) PROTO ((const Node *, const Node *));
static int qsort_comp PROTO ((const void *, const void *));
static int qsort_comp (elem1, elem2)
    const void *elem1;
    const void *elem2;
{
    Node **node1 = (Node **) elem1;
    Node **node2 = (Node **) elem2;
    return client_comp (*node1, *node2);
}
/*
 * sort the elements of a list (in place)
 */
void sortlist (list, comp)
List *list;
{
    Node *head, *remain, *p, **array;
    int i, n;
    /* save the old first element of the list */
    head = list->list;
    remain = head->next;
    /* count the number of nodes in the list */
    n = 0;
    for (p = remain; p != head; p = p->next)
        n++;
    /* allocate an array of nodes and populate it */
    array = (Node **) xmalloc (sizeof(Node *) * n);;
    i = 0;
    for (p = remain; p != head; p = p->next)
        array[i++] = p;
    /* sort the array of nodes */
    client_comp = comp;
    qsort (array, n, sizeof(Node *), qsort_comp);
    /* rebuild the list from beginning to end */
    head->next = head->prev = head;
    for (i = 0; i < n; i++)
    {
        p = array[i];
        p->next = head;
        p->prev = head->prev;
        p->next->prev = p;
    }
    /* release the array of nodes */
    free (array);
}
/*
 * compare two files list node (for sort)
 */
int fsortcmp (p, q)
    const Node *p;
    const Node *q;
{
    return (strcmp (p->key, q->key));
}
```
/ * Debugging functions. Quite useful to call from within gdb. */

static char *nodetypestring PROTO ((Ntype));

static char *

nodetypestring (type)
Ntype type;
{
    switch (type) {
        case UNKNOWN: return("UNKNOWN");
        case HEADER: return("HEADER");
        case ENTRIES: return("ENTRIES");
        case FILES: return("FILES");
        case LIST: return("LIST");
        case RCSNODE: return("RCSNODE");
        case RCSVERS: return("RCSVERS");
        case DIRS: return("DIRS");
        case UPDATE: return("UPDATE");
        case LOCK: return("LOCK");
        case NDBMNODE: return("NDBMNODE");
        case FILEATTR: return("FILEATTR");
        case VARIABLE: return("VARIABLE");
        case RCSFIELD: return("RCSFIELD");
    }
    return("<trash>");
}

static int printnode PROTO ((Node *, void *));

static int

printnode (node, closure)
Node *node;
void *closure;
{
    if (node == NULL) {
        (void) printf("NULL node.
");
        return(0);
    }
    (void) printf("Node at 0x%p: type = %s, key = 0x%p = \%s\", data = 0x%p, prev = 0x%p\n", node, nodetypestring(node->type), node->key, node->data, node->next, node->prev);
    return(0);
}

void printlist PROTO ((List *));

void

printlist (list)
List *list;
{
    if (list == NULL) {
        (void) printf("NULL list.
");
        return;
    }
    (void) printf("List at 0x%p: list = 0x%p, HASHSIZE = %d, next = 0x%p\n", list, list->list, HASHSIZE, list->next);
    (void) walklist(list, printnode, NULL);
    return;
}
A.28 hash.h

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

/* The number of buckets for the hash table contained in each list. This
 * should probably be prime.
 */
#define HASHSIZE 151

/* Types of nodes */
enum ntype
{
    UNKNOWN, HEADER, ENTRIES, FILES, LIST, RCSNODE,
    RCSVERS, DIRS, UPDATE, LOCK, NDBMNODE, FILEATTR,
    VARIABLE, RCSFIELD
};
typedef enum ntype Ntype;

struct node
{
    Ntype type;
    struct node *next;
    struct node *prev;
    struct node *hashnext;
    struct node *hashprev;
    char *key;
    char *data;
    void (*delproc)();
};
typedef struct node Node;

struct list
{
    Node *list;
    Node *hasharray[HASHSIZE];
    struct list *next;
};
typedef struct list List;

List *getlistPROTO(void);
Node *findnodePROTO(List *list, const char *key));
Node *findnodefnPROTO(List *list, const char *key));
Node *getnode PROTO(void);
int insert_beforePROTO(List *list, Node *marker, Node *p));
int addnodePROTO(List *list, Node *p));
int addnode_at_front PROTO(List *list, Node *p));
int walklist PROTO(List *list, int (*) (Node *n, void *closure), void *closure));
int list_isempty PROTO ((List *list));
void dellist PROTO(List **listp);
void delnode PROTO(Node *p));
void freenode PROTO((Node * p));
void sortlist PROTO((List * list, int (*)(const Node * const Node *, const Node *)));
int fsortcmp PROTO((const Node * p, const Node * q));
A.29 history.c

/*
 You may distribute under the terms of the GNU General Public License
 as specified in the README file that comes with the CVS 1.0 kit.

*************** History of Users and Module ***************

LOGGING: Append record to "$CVSROOT/ADMM/HISTORY".

10 On For each Tag, Add, Checkout, Commit, Update or Release command,
one line of text is written to a History log.

X date | user | CurDir | special | rev(s) | argument 'n'

where: [The spaces in the example line above are not in the history file.]

X is a single character showing the type of event:
T "Tag" cmd.
O "Checkout" cmd.
P "Report" cmd.
W "Update" cmd - No User file, Remove from Entries file.
U "Update" cmd - File was checked out over User file.
G "Update" cmd - File was merged successfully.
C "Update" cmd - File was merged and shows overlaps.
M "Commit" cmd - "Modified" file.
R "Commit" cmd - "Removed" file.

20 date is a fixed length 8-char hex representation of a Unix time.

user is the username of the person who typed the command.

CurDir The directory where the action occurred. This should be the
absolute path of the directory which is at the same level as

Repository For record types [W,U,G,C,M,A,R] this field holds the
repository read from the administrative data where the
command was typed.
T "A" -> New Tag, "D" -> Delete Tag
Otherwise it is the Tag or Date to modify.
O,F,E A "" (null field)

30 rev(s) Revision number or tag.
T The Tag to apply.
O,E The Tag or Date, if specified, else "" (null field).
P "" (null field)
W The Tag or Date, if specified, else "" (null field).
U The Revision checked out over the User file.
G,C The Revision(s) involved in merge.
M,A,R RCS Revision affected.

argument The module (for [TOEUF]) or file (for [WUGCMAR]) affected.

Report categories: "User" and "Since" modifiers apply to all reports.
[For "sort" ordering see the "sort" routine.]

for record types

40 -e, -x [TOEFWUGCMAR]

Extracted records are simply printed, No analysis is performed.
All "field" modifiers apply. -e chooses all types.

Checked 'O'ut modules

50 -o, -w

Checked out modules. "F" and "O" records are examined and if
the last record for a repository/file is an 'O', a line is
printed. "-w" forces the "working dir" to be used in the
comparison instead of the repository.

Committed (Modified) files

60 -c, -l, -w

All 'M'odified, 'A'dded and 'R'emoved records are examined.
"Field" modifiers apply. -l forces a sort by file within user
and shows only the last modifier. -w works as in Checkout.

Warning: Be careful with what you infer from the output of
"cvs hi -c -l". It means the last time anyone
changed the file, not the list of files for which
you were the last changer!!!

Module history for named modules.
-m module, -l

This is special. If one or more modules are specified, the module names are remembered and the files making up the modules are remembered. Only records matching exactly those files and repositories are shown. Sorting by "module", then filename, is implied. If -l ("last modified") is specified, then "update" records (types WUGQ), tag and release records are ignored and the last (by date) "modified" record.

TAG history

-T All Tag records are displayed.

*** Modifiers.

Since . . . [All records contain a timestamp, so any report category can be limited by date.]
-D date - The "date" is parsed into a Unix "time_t" and records with an earlier time stamp are ignored.
-r rev/tag - A "rev" begins with a digit. A "tag" does not. If you use this option, every file is searched for the indicated rev/tag.
-t tag - The "tag" is searched for in the history file and no record is displayed before the tag is found. An error is printed if the tag is never found.
-b string - Records are printed only back to the last reference to the string in the "module", "file" or "repository" fields.

Field Selections [Simple comparisons on existing fields. All field selections are repeatable.]
-a - All users.
-u user - If no user is given and '-a' is not given, only records for the user typing the command are shown.
-f filematch - Only records in which the "file" field contains the string "filematch" are considered.
-p repository - Only records in which the "repository" string is a prefix of the "repos" field are considered.
-m modulename - Only records which contain "modulename" in the "module" field are considered.

EXAMPLES: ("cvs history", "cvs his" or "cvs hi")

*** Checked out files for username. (default self, e.g. "dgg")
cvs hi [equivalent to: "cvs hi -o -u dgg"]
cvs hi -u user [equivalent to: "cvs hi -o -u user"]
cvs hi -o [equivalent to: "cvs hi -o -u dgg"]

*** Committed (modified) files from the beginning of the file.
cvs hi -c [-u user]

*** Committed (modified) files since Midnight, January 1, 1990:
cvs hi -c -D 'Jan 1 1990' [-u user]

*** Committed (modified) files since tag "TAG" was stored in the history file:
cvs hi -c -t TAG [-u user]

*** Committed (modified) files since tag "TAG" was placed on the files:
cvs hi -c -r TAG [-u user]

*** Who last committed file/repository X?
cvs hi -c -l [-fp] X

*** Modified files since tag/date/file/repos?
cvs hi -c (-r TAG | -D Date | -b string)

*** Tag history

cvs hi -T

*** History of file/repository/module X.
cvs hi [-fp] X

*** History of user "user".
cvs hi -e -u user

*** Dump (eXtract) specified record types

cvs hi -x [TOPWUGCMAR]

FUTURE: J[Join], I[Import] (Not currently implemented.)
#include "cvs.h"
#include "saved.h"

static struct hrec {
    char type;    /* Type of record (In history record) */
    char user;    /* Username (In history record) */
    char dir;     /* "Compressed" Working dir (In history record) */
    char repos;   /* Tag is special.) Repository (In history record) */
    char rev;     /* Revision affected (In history record) */
    char file;    /* Filename (In history record) */
    char end;     /* Ptr into repository to copy at end of workdir */
    char smod;    /* The module within which the file is contained */
    time_t date;  /* Calculated from date stored in record */
    int idx;      /* Index of record, for "stable" sort. */
} *hrec;  /* hrec deals;*/

static char *fill_hrec PROTO((char *line, struct hrec *hrec + hr));
static int accept_hrec PROTO((struct hrec *hrec + hr));
static int select_hrec PROTO((struct hrec *hrec + hr));
static int sort_order PROTO((const PTR l, const PTR r));
static int within PROTO((char *find, char *string));
static void expand Modules PROTO((void));
static void read_hrecs PROTO((char *fname));
static void report_hrecs PROTO((void));
static void save_file PROTO((char *dir, char *name, char *module));
static void save_module PROTO((char *module));
static void save_user PROTO((char *name));

#define ALL_REC_TYPES "TREPWTCH"  
#define USER_INCREMENT 2
#define FILE_INCREMENT 128
#define MODULE_INCREMENT 5
#define HREC_INCREMENT 128

static short report_count;
static short extract;
static short checkout;
static short modified;
static short tagreport;
static short module_report;
static short working;
static short last_entry;
static short all_users;
static short usersort;
static short reposort;
static short filesort;
static short modulesort;
static short tz_local;
static time_t tz_seconds_local_GMT;
static char *tz_name = "+0000";

/* -r, -t, or -b options, malloc'd. These are "" if the option in
question is not specified or is overridden by another option. The
main reason for using "" rather than NULL is historical. Together
with since_date, these are a mutually exclusive set; one overrides the
others. */
static char *since_yy;
static char *since_mm;
static char *backup;
/* -D option, or 0 if not specified. RCS format. */
static char *since_date;

static struct hrec *last_since_MM;
static struct hrec *last_backup;

/* Record types to look for, malloc'd. Probably could be statically
allocated, but only if we wanted to check for duplicates more than
we do. */
static char *rec_types;
static int hrec_count;
static int hrec_max;
static char *user_list;    /* Pts to array of pts to user names */
static int user_max;       /* Number of elements allocated */
static int user_count;     /* Number of elements used */

static struct file_list_str {
    char *file;
    char *module;
} *file_list;  /* Pts to array file name structs */
static int filelist_max;  /* Number of elements allocated */
static int filelist_count;  /* Number of elements used */
static char *mod_list; /* Ptr to array of ptrs to module names */
static int mod_max; /* Number of elements allocated */
static int mod_count; /* Number of elements used */
static char *histfile; /* Ptr to the history file name */

/* This is pretty unclear. First of all, separating "flags" vs.
   "options" (I think the distinction is that "options" take arguments)
   is nonstandard, and not something we do elsewhere in CVS. Second of
   all, what does "reports" mean? I think it means that you can only
   supply one of those options, but "reports" hardly has that meaning in
   a self-explanatory way. */

static const char *const history_usage[] = {
    "Usage: %s [report] [-flags] [-options args] [files...]

    -r <rev/tag> Since rev or tag (looks inside RCS files!)
    -t <tag> Since tag record placed in history file (by anyone).
    -x [TOEFWUCGMAR] Extract by record type
    -m <module> Look for specified module (repeatable)
    -c Committed (Modified) files
    -T Produce report on all TAGs
    Reports:
    l
    -n <modulename> In module (repeatable)
    -p <repos> In repository (repeatable)
    -z <tz> Output for time zone <tz> (e.g. -z -0700)
    -b <str> Back to record with str in module/file/repos field
    -u <user> For user name (repeatable)
    -D <date> Since date (Many formats)
    -l Last modified (committed or modified report)
    -w Working directory must match
    -E Everything (same as -x, but all record types)
    -f <file> Specified file (same as command line) (repeatable)
    -g <user> For user (repeatable)
    -s <repos> In repository (repeatable)
    Flags:
    -F If a user is selected at all, sort it first. User-within-file is useless.
    -S - If a module was selected explicitly, sort next on module.
       Then sort by file. "File" is "repository/file" unless "working" is set,
       then it is "workdir/file". (Revision order should always track date.)
       Always sort timestamp last.

    Sort routine for qsort:
    - If a user is selected at all, sort it first. User-within-file is useless.
    - If a module was selected explicitly, sort next on module.
    - Then sort by file. "File" is "repository/file" unless "working" is set,
      then it is "workdir/file". (Revision order should always track date.)
      Always sort timestamp last.

    */

    static int sort_order (l, r)
    
    const PTR l;
    const PTR r;
    
    int i;
    const struct hrec *left = (const struct hrec *) l;
    const struct hrec *right = (const struct hrec *) r;
    
    if (user_sort) /* If Sort by username, compare users */
    
    if ((i = strcmp (left->user, right->user)) != 0)
    return (i);
    
    if (module_sort) /* If sort by modules, compare module names */
    
    if ((left->mod & right->mod)) != 0)
    return (i);
    
    if (repos_sort) /* If sort by repository, compare them */
    
    if ((i = strcmp (left->repos, right->repos)) != 0)
    return (i);
    
    if (file_sort) /* If sort by filename, compare files, NOT dirs */
    
    if ((i = strcmp (left->file, right->file)) != 0)
    return (i);
    
    if (working)
    
    if ((i = strcmp (left->dir, right->dir)) != 0)
    return (i);
    
    if ((i = strcmp (left->end, right->end)) != 0)
    return (i);
    
    /* By default, sort by date, time 
    XXX: This fails after 2030 when date slides into sign bit */
if ((i = ((long) (left->date) - (long) (right->date))) != 0)
    return (i);

    /* For matching dates, keep the sort stable by using record index */
    return (left->idx = right->idx);
}

int
history (argc, argv)
int argc;
char **argv;
{
    int i, c;
    char *name;
    if (argc == -1)
        usage (history.c);

    since_rev = xstrdup (**);
since_tag = xstrdup (**);
backto = xstrdup (**);

rec_types = xstrdup (**);
optind = 0;
while ((c = getopt (argc, argv, "T:dl:X:r:t:u:x:z:")) != -1)
{
    switch (c)
    {
    case 'T':
        /* Tag list */
        report_count++;
        tag_report++;
        break;
    case 'a':
        /* For all usernames */
        all_users++;
        break;
    case 'c':
        report_count++;
        modified = 1;
        break;
    case 'e':
        report_count++;
        extract++;
        free (rec_types);
        rec_types = xstrdup (ALL_REC_TYPES);
        break;
    case 'l':
        /* Find Last file record */
        last_entry = 1;
        break;
    case 'o':
        report_count++;
        v_checkout = 1;
        break;
    case 'y':
        /* Match Working Dir (CurDir) fields */
        working = 1;
        break;
    case 'X':
        /* Undocumented debugging flag */
        histfile = optarg;
        break;
    case 'D':
        /* Since specified date */
        if (since_rev || since_tag || backto)
            {
                error (0, 0, "date overriding rev/tag/backto");
                since_rev = since_tag = backto = "\0";
            }
        since_date = MakeDate (optarg);
        break;
    case 'b':
        /* Since specified file/Repos */
        if (since_file || since_rev || since_tag)
            {
                error (0, 0, "backto overriding date/rev/tag");
                since_rev = since_tag = backto = "\0";
                if (since_date != NULL)
                    free (since_file);
                since_file = NULL;
            }
        free (backto);
        backto = xstrdup (optarg);
        break;
    case 'f':
        /* For specified file */
        save_file (**, optarg, (char *) NULL);
        break;
    case 'm':
        /* Full module report */
        report_count++;
        module_report++;
        break;
    case 'a':
        /* Look for specified module */
        save_module (optarg);
        break;
    case 'p':
        /* For specified directory */
        save_file (optarg, **, (char *) NULL);
        break;
    case 'r':
        /* Since specified Tag/Rev */
if (since\_late || since\_tag || *backto*)
{
    error (5, 0, "rev overriding date/tag/backto");
    since\_tag = *backto = '\0';
    if (since\_late != NULL)
        free (since\_late);
    since\_late = NULL;
}
free (since\_ev);
since\_ev = xstrdup (optarg);
break;
case 't':  /* Since specified Tag/Rev */
    if (since\_late || since\_ev || *backto*)
    {
        error (5, 0, "tag overriding date/marker/file/repos");
        since\_ev = *backto = '\0';
        if (since\_late != NULL)
            free (since\_late);
        since\_late = NULL;
    }
    free (since\_tag);
since\_tag = xstrdup (optarg);
break;
case 'u':  /* For specified username */
    save\_user (optarg);
    break;
case 'x':
    report\_count++;  
    extract++;  
    {
        char *cp;
        for (cp = optarg; cp; cp++)
            if (strchr (ALL\_REC\_TYPES, *cp))
                error (1, 0, "Gc is not a valid report type", *cp);
    }
    free (rec\_types);
    rec\_types = xstrdup (optarg);
    break;
case 't_':
    tz\_local =
        (optarg[0] == '1' || optarg[0] == 'L')
        && optarg[1] == 't' || optarg[1] == 'T')
        && optarg[2];
    if (tz\_local)
        tz\_name = optarg;
    else
    {
        /* Convert a known time with the given timezone to time_t.
         * Use the epoch + 23 hours, so timezones east of GMT work.
         */
        static char f[] = "1/1/1970 23:00 Zs";
        char *buf = xmalloc (sizeof (f) - 2 + strlen (optarg));
        time\_t t;
        sprintf (buf, f, optarg);
        t = get\_late (buf, (struct timeb *) NULL);
        free (buf);
        if (t == (time\_t) -1)
            error (0, 0, "Es is not a known time zone", optarg);
        else
        {
            /* Convert to seconds east of GMT, removing the
             * 23-hour offset mentioned above.
             */
            time\_seconds\_east\_of\_GMT = (time\_t)23 * 60 + t;
            tz\_name = optarg;
        }
    }  
    break;
case 'r_':
default:
    usage (history\_usg);
    break;
}
=c = optind;  /* Save the handled option count */

رعا = ------------ Now analyze the arguments a bit */
if (report\_count)
    v\_checkout++;  
else if (report\_count > 1)
    error (1, 0, "Only one report type allowed from: \"-Tcomx\", ");

#define CLIENT\_SUPPORT  
if (client\_active)
    {  
        struct filelist\_str *f;
        char **mod;
/+ We're the client side. Fire up the remote server. */

start_server();

ign_setup();

if (tag_report)
    send_arg("-T");
if (all_users)
    send_arg("-a");
if (modified)
    send_arg("-c");
if (lastentry)
    send_arg("-l");
if (recheckout)
    send_arg("-o");
if (working)
    send_arg("-w");
if (histfile)
    send_arg("-x");
if (since_date)
    client_senddate(since_date);
if (backto0) := 'V'
    option_with_arg (*'V', backto);
for (f1 = file_list; f1 < &file_list[file_count]; ++f1)
{
    if (f1->>file[0] == 'x')
        option_with_arg (*'x', f1->>file + 1);
    else
        option_with_arg (*'x', f1->>file);
}

if (module_report)
    send_arg("-n");
for (mod = module_list; mod < &module_list[module_count]; ++mod)
    option_with_arg (*'n', *mod);
if (*since_ev)
    option_with_arg (*'r', since_ev);
if (*since_tag)
    option_with_arg (*'r', since_tag);
for (mod = user_list; mod < &user_list[user_count]; ++mod)
    option_with_arg (*'t', *mod);
if (extract)
    option_with_arg (*'t', rec_types);
    option_with_arg (*'x', timezone);
    send_to_server("history\012", 0); return get_responses_and_close();

sendif

if (all_users)
    save_user (**);
if (module_list)
    expand_modules();
if (tag_report)
{
    if (strchr(rec_types, 'T'))
    {
        rec_types = xrealloc(rec_types, strlen(rec_types) + 5);
        (void) strcat(rec_types, "T");
    }
} else if (extract)
{
    if (user_list)
        user_sort++; 
    } else if (modified)
{

free(rec_types);
rec_types = xstrdup("MKB");

/* If the user has not specified a date oriented flag ("Since"), sort
 * by Repository/file before date. Default is "just" date.
 */
if (since_date & *since_ev & *since_tag & !backto)
    report_sort++; 
    file_sort++; 

/* If we are not looking for last_modified and the user specified
 * one or more users to look at, sort by user before filename.
 */
if (lastentry & user_list)
    user_sort++;
} else if (module_report)
{ free/rec_types);
  rec_types = xstrdup(last_entry ? "U" : ALL_REC_TYPES);
  module_sort_i++; 
  repo_sort_i++; 
  file_sort_i++; 
  working = 0; /* User's workdir doesn't count here */
} else  
  /* Must be "checkout" or default */
  { free/rec_types);
  rec_types = xstrdup("P"); /* See comments in "modified" above */
  if(last_entry && user_list) 
    user_sort_i++; 
    if((since_date && !since_rev && !since_tag && !backto) 
      file_sort_i++;
  }
  /* If no users were specified, use self (-a saves a universal ("") user */
  if(user_list)
    save_user(getcaller()); 
    /* If we're looking back to a Tag value, must consider "Tag" records */
    if(!since_tag && !hrech 
      memrecs = xmalloc(rec_types, strlen(rec_types) + 5); 
      (void)sprintf(rec_types, "T"); 
    }
  
  argc -= c;
  argv += c;
  for (i = 0; i < argc; i++) 
    save_file("", argv[i].char + NULL);
  }
  if (histfile) 
    fname = xstrdup(histfile);
  else  
    { 
      fname = xmalloc(strlen(CVSroot)directory) + sizeof(CVSROOTADM) 
        + sizeof(CVSROOTADM_HISTORY) + 10;
        (void)sprintf((fname, "Ta/Ta/Ta", CVSroot, directory, 
          CVSROOTADM, CVSROOTADM_HISTORY); 
      }
    read_hrecs(fname);
    qsort((PTR)hrec, hrec_count, sizeof(struct hrec), sort_order);
    report_hrecs();
    free (fname);
    if (since_date != NULL)
      free(since_date);
    free (since_rev);
    free (since_tag);
    free (backto);
    free (rec_types);
    return (0);
} 

void
history_write(type, update_dir, revs, name, repository)
  int type;
  char *update_dir;
  char *revs;
  char *name;
  char *repository;
  { char *fname;
    char *workdir;
    char *username = getcaller();
    int fd;
    char *tilde = "+", *cp, *cp2, *repos;
    int i;
    static char *tildes = "+";
    static char *PrCurDir = NULL;
    if (logoff) /* History is turned off by cmd line switch */
      return;
    fname = xmalloc(strlen(CVSroot)directory) + sizeof(CVSROOTADM) 
      + sizeof(CVSROOTADM_HISTORY) + 10;
      (void)sprintf((fname, "Ta/Ta/Ta", CVSroot, directory, 
        CVSROOTADM, CVSROOTADM_HISTORY); 
    /* turn off history logging if the history file does not exist */
    if (isfile (fname))
      { logoff = 1; 
        goto out; 
      }
if (trace)
    SERVER_WRITE("CHDIR: cwd %s
", (server_active) ? "$" : "");
else
    fprintf(stderr, "%s" fname);
#endif

int fd = CVS_OPEN(fname, O_WRONLY | O_APPEND | O_CREAT | OPEN_BINARY, 0666);
if (fd < 0)
    error (1, errno, "cannot open history file: %s", fname);
repos = ShortRepository (repository);
if (PrCurDir)
    char *pwdir;
pwdir = gethomdir ();
PrCurDir = CurDir;
if (pwdir != NULL)
    { /* Assumes neither CurDir nor pwdir ends in */ */
        i = strlen(pwdir);
        if (strncmp (CurDir, pwdir, i))
            { PrCurDir += i; /* Point to */ separator */
                tilda = "~";
            }
else
    { /* Try harder to find a *homedir* */
        struct saved_cwd cwd;
        char *homedir;
        if (saved_cwd (cwd))
            error (1, NULL);
        if (CVS_getdirdir (pwdir) < 0)
            error (1, errno, "can't getdirdir(%s)", pwdir);
        homedir = xgetwd ();
        if (homedir == NULL)
            error (1, errno, "can't getwd in %s", pwdir);
        if (restored_cwd (&cwd, NULL))
            error (1, NULL);
        free_cwd (&cwd);
        i = strlen(homedir);
        if (strncmp (CurDir, homedir, i))
            { PrCurDir += i; /* Point to */ separator */
                tilda = "~";
            }
        free (homedir);
    }
}
if (type == 'T')
    { repos = update_dir;
        update_dir = "";
    }
else if (update_dir && *update_dir)
    slash = "/";
else
    update_dir = "";
workdir = xmalloc (strlen (tilda) + strlen (PrCurDir) + strlen (slash)
    + strlen (update_dir) + 10);
(void) sprintf (workdir, "%s$\%s", tilda, PrCurDir, slash, update_dir);

/* "workdir" is the directory where the file "name" is. ("~\%s" == $HOME)
 * "repos" is the Repository, relative to $CVSROOT where the RCS file is.
 * "$Workdir/$name" is the working file name.
 * "$CVSROOT$/repos/$name,w" is the RCS file in the Repository.
 * First, note that the history format was intended to save space, not
 * to be human readable.
 * Note:
 * The working file directory ("workdir") and the Repository ("repos")
 * usually end with the same one or more directory elements. To avoid
 * duplication (and save space), the "workdir" field ends with
 * an integer offset into the "repos" field. This offset indicates

beginning of the "tail" of "repos", after which all characters are
duplicates.

In other words, if the "workdir" field has a ' /' (a very stupid thing
to put in a filename) in it, then every thing following the last '/'
is a hex offset into "repos" of the first character from "repos" to
append to "workdir" to finish the pathname.

It might be easier to look at an example:

M$7b346[3]dpq\ [/work\ ]usr\ /local\ /cvs\ /examples\ ]1.2\login

Indicates that the workdir is really " /work/cvs/examples", saving
10 characters, where " /work/cvs" would save 6 characters and mean that
the workdir is really " /work/examples". It will mean more on
directories like: war/local/gnu/emacs/dist-19.17/bsp/term

"workdir" is always an absolute pathname ( /xxx is an absolute path)
"repos" is always a relative pathname. So we can assume that we will
never run into the top of "workdir" – there will always be a " /" or
a " /" at the head of "workdir" that is not matched by anything in
"repos". On the other hand, we *can* run off the top of "repos".

Only "compress" if we save characters.
*/

if (repos)
repos = " /";

if (cp = workdir + strlen(workdir) - 1;
   cp2 = repos + strlen(repos) - 1;
   for (i = 0; cp2 >= repos & & cp > workdir & & *cp == *cp2--; cp--)
i += i;
}

if (i > 2)
{
    i = strlen(repos) - i;
    (void) sprintf ((cp + 1), " *\%s", i);
}

if (revs)
repos = " /"
line = xmalloc (strlen (username) + strlen (workdir) + strlen (repos)
   + strlen (revs) + strlen (name) + 120);
strftime (line, "%F%T%v\%n\%s\%t\%v\%n\%s\%t\%v\%n", time, (long) ((time_t *) NULL)),
   username, workdir, repos, revs, name);

/* Lesser some race conditions on non-Possix-compliant hosts. */
if (lseek (fd, (off_t) 0, SEEK_END) == -1)
error (1, errno, "cannot seek to end of history file: %s", fname);

if (write (fd, line, strlen (line)) < 0)
error (1, errno, "cannot write to history file: %s", fname);
free (line);
if (close (fd) != 0)
error (1, errno, "cannot close history file: %s", fname);
free (workdir);
out:
free (fname);
}

/*
 * save_user() adds a user name to the user list to select. Zero-length
 * user name ("") matches any user.
 */
static void
save_user (name)
char *name;
{
    if (user_count == user_max)
    {
        user_max += USER_INCREMENT;
        user_list = realloc (user_list + user_max * sizeof (char *));
    }
    user_list[user_count++] = xstrdup (name);
}

/* save_file() adds file name and associated module to the file list to select.

If "dir" is null, store a file name as is.
If "name" is null, store a directory name with a ' /' on the front.
Else, store concatenated "dir/\name".

Later, in the "select" stage:
- if it starts with ' /', it is prefix-matched against the repository.
- if it has a ' /' in it, it is matched against the repository/file.
- else it is matched against the file name.
static void
save_file (dir, name, module)
{
    char *dir;
    char *name;
    char *module;
    {
        char *cp;
        struct file_list *fl = &fl;
        if (file_count == file_max)
        {
            file_max += FILE_INCREMENT;
            file_list = (struct file_list *) xrealloc ((char *) file_list,
                    file_max + sizeof (*fl));
        } else
            file_list[file_count++] = xmalloc (strlen (dir) + strlen (name) + 2);
        file->module = module;
        if (dir && *dir)
        {
            (void) strpy (cp, dir);
            (void) strcat (cp, "/");
            (void) strcat (cp, name);
            if (name && *name)
                { 
                    *cp++ = '*';
                    (void) strpy (cp, dir);
                }
        } else
            if (name && *name)
                { 
                    (void) strpy (cp, name);
                }
            else
                { 
                    error (0, 0, "save_file: null dir and file name");
                }
        }
    } else
        { 
            return (0);
        } else
        { 
            return (1);
        } else
        { 
            return (-1);
        } static void
save_module (module)
    {
        char *module;
        if (mod_count == mod_max)
        {
            mod_max += MODULE_INCREMENT;
            mod_list = (char **) xrealloc ((char *) mod_list,
                    mod_max + sizeof (char *));
        } mod_list[mod_count++] = xstrdup (module);
    } static void
expand_modules ()
    {
    } /* fill_rec
* Take a ptr to 7-part history line, ending with a newline, for example:
*  \
*  "273b3463$dvq|/work+9/user/local/cvs/examples/1.2[loginfo
* * Split it into 7 parts and drop the parts into a "struct hrec".
* Return a pointer to the character following the newline.
* */
define NEXT_BAR(here) do { while (isspace(*line)) line++; hr->here = line;
        while ((c = *line++) && c != '1') ; if (lo) return(sta); 
        (*line - 1) = '\0';} while (0) static char *
fill_rec (line, hr)
    {
        char *cp, *rtn;
        struct hrec *hr;
        { 
            char *cp, *rtn;
            int c;
            int off;
            static int idx = 0;
            unsigned long date;
memset ((char *) hr, 0, sizeof (*hr));
while ((isspace (*line)))
  line++;
if (!((line = strchr (line, '\n'))))
  return (*);
  *rtn++ = '\0';
hr->type = line++;
(void) scanf (line, "%s", &date);
hr->date = date;
while (*line && strchr (*0123456789abcdefABCDEF", *line))
  line++;
if (*line == '\n')
  return (rtn);

NEXTHREC (user);
NEXTHREC (dir);
if (((cp = strchr (hr->dir, '*')) != NULL))
  
  *cp++ = '\0';
  (void) scanf (cp, "%s", &off);
  hr->end = line + off;
else
  hr->end = line - 1;  /* A handy pointer to \0 */
NEXTHREC (repos);
NEXTHREC (rev);
hr->idx = idx + 1;
if (strchr (*REV", *(hr->type)))
  hr->mod = line;

NEXTHREC (file);  /* This returns ptr to next line or final \0 */
return (rtn);  /* If it falls through, go on to next record */

/* read_hrec's job is to read the history file and fill in all the "hrec"
 * (history record) array elements with the ones we need to print.
 * Logic:
 * - Read the whole history file into a single buffer.
 * - Walk through the buffer, parsing lines out of the buffer.
 *   1. Split line into pointer and integer fields in the "next" hrec.
 *   2. Apply tests to the hrec to see if it is wanted.
 *   3. If it is wanted, bump the hrec pointer down by one.
 */

static void
read_hrec (fname)
char *cp, *cp2;
int i, fd;
struct hrec *hr;
struct stat st_buf;
if ((fd = CVSOPEN (fname, O_RDONLY | OPEN_BINARY)) < 0)
  error (1, errno, "cannot open history file: \n", fname);
if (fstat (fd, &st_buf) < 0)
  error (1, errno, "can't stat history file");

/* Exactly enough space for lines data */
if ((i = st_buf.st_size) < 0)
  error (1, 0, "history file is empty");
cp = xmalloc (i + 2);
if (read (fd, cp, i) != i)
  error (1, errno, "cannot read log file");
(void) close (fd);
if (*cp + i - 1 != '\n')
  
  *(cp + i) = '\n';  /* Make sure last line ends in \n */
  
  i++;
/*cp += i;*/
for (cp2 = cp; cp2 - cp < i; cp2++)
  
  if (*cp2 != '\n' & & isspace (*cp2))
    cp2 = ' ';

hrec_max = HREC_INCREMENT;
hrec_head = (struct hrec *) xmalloc (hrec_max * sizeof (struct hrec));
while (*cp)
  
  if (hrec_count == hrec_max)
    
    struct hrec *old_head = hrec_head;

hrec_max += HREC_INCREMENT;
  hrec_head = (struct hrec *) realloc ((char *) hrec_head,
         hrec_max + sizeof (struct hrec));

  if (hrec_head != old_head)
    {
    if (last_since_tag)
      last_since_tag = hrec_head + (last_since_tag - old_head);
    if (last_backto)
      last_backto = hrec_head + (last_backto - old_head);
    }

  hr = hrec_head + hrec_count;
  cp = fill_rec (cp, hr); /* cp == next line or \0 at end of buffer */

  if (select_hrec (hr))
    hrec_count++;}

  /* Special selection problem: If “since_tag” is set, we have saved every
   * record from the 1st occurrence of “since_tag”, when we want to save
   * records since the last occurrence of “since_tag”. So what we have
   * to do is bump hrec_head forward and reduce hrec_count accordingly.
   */
  if (last_since_tag)
    {
    hrec_count -= (last_since_tag - hrec_head);
    hrec_head = last_since_tag;
    }

  /* Much the same thing is necessary for the “backto” option. */
  if (last_backto)
    {
    hrec_count -= (last_backto - hrec_head);
    hrec_head = last_backto;
    }

  /* Utility program for determining whether “find” is inside “string” */
static int
within (find, string)
char  *find, *string;

  { int c, len;
      if (!find || !string)
        return (0);
      c = *find++;
      len = strlen (find);

      while (*string)
        { if ((string = strchr (string, c)))
            return (0);
              string++;}
      return (1);
    }

  /* The purpose of “select_hrec” is to apply the selection criteria based on
   * the command arguments and defaults and return a flag indicating whether
   * this record should be remembered for printing.
   */
static int
select_hrec (hr)
struct hrec  *hr;

  { struct file_list  *fl;

    /* “Since” checking. The argument parser guarantees that only one of the
     * following four choices is set:
     * 1. If “since_date” is set, it contains the date specified on the
        command line, hr->date fields earlier than “since_date” are ignored.
     * 2. If “since_rev” is set, it contains either an RCS “-dotted” revision
        number (which is of limited use) or a symbolic TAG. Each RCS file
        is examined and the date on the specified revision (or the revision
        corresponding to the TAG) in the RCS file (CVSROOT/repos/file) is
        compared against hr->date as in 1. above.
     * 3. If “since_tag” is set, matching tag records are saved. The field
        “last_since_tag” is set to the last one of these. Since we don’t
        know where the last one will be, all records are saved from the
        first occurrence of the TAG. Later, at the end of “select_hrec”
        records before the last occurrence of “since_tag” are skipped.
     */

    int count;
    struct file_list  *fl;

    /* Since checking: The argument parser guarantees that only one of the
     * following four choices is set:
     */
    /* 1. If “since_date” is set, it contains the date specified on the
        command line, hr->date fields earlier than “since_date” are ignored.
     */
    /* 2. If “since_rev” is set, it contains either an RCS “-dotted” revision
        number (which is of limited use) or a symbolic TAG. Each RCS file
        is examined and the date on the specified revision (or the revision
        corresponding to the TAG) in the RCS file (CVSROOT/repos/file) is
        compared against hr->date as in 1. above.
     */
    /* 3. If “since_tag” is set, matching tag records are saved. The field
        “last_since_tag” is set to the last one of these. Since we don’t
        know where the last one will be, all records are saved from the
        first occurrence of the TAG. Later, at the end of “select_hrec”
        records before the last occurrence of “since_tag” are skipped.
     */
4. If "backto" is set, all records with a module name or file name
   matching "backto" are saved. In addition, all records with a
   repository field with a "prefix" matching "backto" are saved.
   The field "lastrev" is set to the last one of these. As in
   3. above, "select" adjusts to include the last one later on.

if (since_last)
{
    char *ourdate = date_from_time (hr->date);
    if (RCS_datecmp (ourdate, since_last) < 0)
        return (0);
    free (ourdate);
} else if (*since_rev)
{
    Vers_TS vers;
    struct file_info finfo;

    memset (&finfo, 0, sizeof finfo);
    finfo.file = hr->file;
    /* Not used, so don't worry about it. */
    finfo.update_dir = NULL;
    finfo.fullname = info_file;
    finfo.repository = hr->repos;
    finfo.entries = NULL;
    finfo.rcs = NULL;
    vers = Version_TS (&finfo, (char *) NULL, since_rev, (char *) NULL,
        1, 0);
    if (vers->vn_rcs)
        if (t = RCS_getmtime (vers->srcfile, vers->vn_rcs, (char *) 0, 0))
            /* t is 0 if not this user */
            if (hr->date < t)
                freevers_ts (&vers);
            return (0);
    } else if (*since_tag)
    { if (*hr->type == 'T')
        { /* A "T"ag record, the "rev" field holds the tag to be set,
           while the "repos" field holds "D"elete, "A"dd or a rev.
           */
                if (within (since_tag, hr->rev))
                { last_since_tag = hr;
                    return (1);
                } else
                    return (0);
    } else if (*backto)
    { if (within (backto, hr->file) || within (backto, hr->mod) ||
        within (backto, hr->repos))
        last_backto = hr;
    else
        return (0);
    } else if (*user_list & hr->user)
    { for (cpp = user_list, count = user_count; count; cpp++, count--)
        if (*cpp)
            break; /* null user == accept */
        if (strcmp (hr->user, *cpp)) /* found listed user */
            break;
    } if (count)
        return (0); /* Not this user */
/* Record type checking:
 * 1. If Record type is not in rec_types field, skip it.
 * 2. If mod_list is null, keep everything. Otherwise keep only modules
 *    on mod_list.
 * 3. If neither a 'T', 'F' nor 'O' record, run through "file_list". If
 *    file_list is null, keep everything. Otherwise, keep only files on
 *    file_list, matched appropriately.
 */

1270 if (!strcmp (rec_types, *(hr->type)))
    return (0);
if (!strcmp (*hr->type))  /* Don't bother with "file" if "TFOE" */
    {if (file_list)        /* If file_list is null, accept all */
         {
 /* 1. If file_list entry starts with 's', skip the 's' and
    compare it against the repository in the hrec.
 * 2. If file_list entry has a '/' in it, compare it against
 *    the concatenation of the repository and file from hrec.
 * 3. Else compare the file_list entry against the hrec file.
 */
         char *cmpfile = NULL;
         if (*cp = file->file == '*')
             {
             cp++;
             /* if argument to -p is a prefix of repository */
             if (strstr (cp, hr->repos, strlen (cp)))
                 {
                 hr->mod = fl->module;
                 break;
                 }
             else
                 {
                 if (chrchr (cp, '/'))
                     {
                     cmpfile = xmalloc (strlen (hr->repos)
                                   + strlen (hr->file)
                                   + 10);
                     sprintf (cmpfile, "%s/%s",
                                    hr->repos, hr->file);
                     cp2 = cmpfile;
                     }
                 else
                     {
                     cp2 = hr->file;
                     }
                 }
             /* if requested file is found within {repos}/file fields */
             if (within (cp, cp2))
                 {
                 hr->mod = fl->module;
                 break;
                 }
             if (cmpfile != NULL)
                 free (cmpfile);
         }
     }
    if (!count)      /* String specified and no match */
        return (0);
    }
else if (mod_list)  
    {
         for (cpp = mod_list, count = mod->count; count; cpp++, count--)
             {if (hr->mod & & strstr (hr->mod, *cpp)) /* found module */
                  break;
             }
         if (!count)  /* Module specified & this record is not one of them. */
             return (0);
    }
else
    {
         return (1);  /* Select this record unless rejected above. */
    }

1340 /* The "sort_order" routine (when handed to qsort) has arranged for the
 *  hrecs file to be in the right order for the report.
 *  Most of the "selections" are done in the select_hrec routine, but some
 *  selections are more easily done after the qsort by "accept_hrec".
 */
static void
report_hrecs ()
{
struct hrec *hr, *lr;
struct tm *tm;
int i, count, ty;
char *cp;
int user_len, file_len, rev_len, mod_len, repos_len;

if (!since_tag && !last_since_tag)
{
    (void) printf("No tag found: \%s\n", since_tag);
    return;
}
else if (!backto && !last_backto)
{
    (void) printf("No module, file or repository with: \%s\n", backto);
    return;
}
else if (hrec_count < 1)
{
    (void) printf("No records selected.\n");
    return;
}

user_len = file_len = rev_len = mod_len = repos_len = 0;

/* Run through lists and find maximum field widths */
hr = lr = hrec_head;
hr++;
for (count = hrec_count; count--; lr = hr, hr++)
{
    char *repos;

    if (!count)
        hr = NULL;
    if (!accept_hrec (lr, hr))
        continue;

    ty = *(lr->type);
    repos = xstrdup (lr->repos);
    if ((cp = strchr (repos, '\')) != NULL)
    {
        if ((lr->mod & & strrmp (++cp, lr->mod))
        {
            (void) strcpy (cp, "*");
        }
    }
    if (!islen (lr->user)) > user_len)
        user_len = i;
    if (!islen (lr->file)) > file_len)
        file_len = i;
    if (ty = 'T' && (i = islen (repos)) > repos_len)
        repos_len = i;
    if (ty = 'T' && (i = islen (lr->rev)) > rev_len)
        rev_len = i;
    if (lr->mod & & (i = islen (lr->mod)) > mod_len)
        mod_len = i;
    free (repos);
}

/* Walk through hrec array setting "lr" (Last Record) to each element.
* "hr" points to the record following "lr" - It is NULL in the last * pass.
* *
* There are two sections in the loop below:
* 1. Based on the report type (e.g. extract, checkout, tag, etc.), * decide whether the record should be printed.
* 2. Based on the record type, format and print the data.
*/
for (lr = hrec_head, hr = (lr + 1); hrec_count--; lr = hr, hr++)
{
    char *workdir;
    char *repos;

    if (!hrec_count)
        hr = NULL;
    if (!accept_hrec (lr, hr))
        continue;

    ty = *(lr->type);
    if (!hr->local)
    {
        time_t t = lr->date + tm_secondsGMT;
        tm = localtime (&t);
    }
    else
        tm = localtime (&(lr->date));

    (void) printf("%s %02d/%02d %02d:%02d %s %-*s",
        tm->tm_mon + 1, tm->tm_mday, tm->tm_hour, tm->tm_min, tm->tm_sec,
        user_len, lr->user);
workdir = xmalloc(strlen (lr->dir) + strlen (lr->end) + 10);

(void) sprintf (workdir, "%s\n", lr->dir, lr->end);
if ((cp = strchr (workdir, '?')) != NULL) {
    if (lr->mod && strcmp (++cp, lr->mod)) {
        (void) strcpy (cp, "*");
    }
}
repos = xmalloc (strlen (lr->repos) + 10);
(void) strcpy (repos, lr->repos);

if ((cp = strchr (repos, '?')) != NULL) {
    if (lr->mod && strcmp (++cp, lr->mod)) {
        (void) strcpy (cp, "*");
    }
}

switch (ty) {
    case 'T':
        /* Tag records: repository is a "tag type", rev is the tag */
        (void) printf ("%s@%s %s %s =%s= %s", ty, workdir);
        if (working) {
            (void) printf ("%s", workdir);
        }
        break;
    case 'E':
    case 'F':
    case 'O':
        if (lr->rev && ++(lr->rev)) {
            (void) printf ("%s", lr->rev);
            (void) printf ("%s\n", ty, workdir);
        }
        break;
    case 'U':
    case 'G':
    case 'C':
    case 'W':
    case 'M':
        (void) printf ("%s\n", ty, workdir);
        break;
    default:
        (void) printf ("Hey! What is this junk? RecType[0x%2.2x]", ty);
        break;
}

(void) putchar (\n);
free (workdir);
frees (repos);
}

static int
accept_rec (lr, hr)
struct hrec *hr, *lr;

int ty;
ty = *(lr->type);

if (lastwincrev & ty == 'T')
    return (1);

if (ty != 'O')
    return (0); /* Only interested in 'O' records */

/* We want to identify all the states that cause the next record */
/* ('hr') to be different from the current one ('lr') and only */
/* print a line at the allowed boundaries. */

if (!hr ||
    strcmp (hr->user, lr->user) || /* User has changed */
    strcmp (hr->mod, lr->mod) || /* Module has changed */
    (working &
     / * If we must watch 'working' */
     ! (strcp (hr->dir, lr->dir) || /* and the 1st parts or */
        ! (strcp (hr->end, lr->end)))); /* the 2nd parts differ */
    return (1);
}
else if (modified)
{

if (!last_entry || /* Don't want only last rec */
hr || /* Last entry is a "last entry" */
strcmp(hr->repos, lr->repos) || /* Repository has changed */
strcmp(hr->file, lr->file)) /* File has changed */
return (1);

if (working) {/* If must match "workdir" */
if (strcmp(hr->dir, lr->dir)) /* and the 1st parts or */
  strcmp(hr->end, lr->end)) /* the 2nd parts differ */
  return (1);
} else if (module_report)
{
  if (!last_entry || /* Don't want only last rec */
    hr || /* Last entry is a "last entry" */
    strcmp(hr->mod, lr->mod) || /* Module has changed */
    strcmp(hr->repos, lr->repos) || /* Repository has changed */
    strcmp(hr->file, lr->file)) /* File has changed */
  return (1);
}
else /* "extract" and "tag_report" always print selected records. */
{
  return (1);
}

return (0);
ignore.c

A.30 ignore.c

/* This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

#include "cvs.h"
#include "getline.h"

/* Ignore file section. 
* "!" may be included any time to reset the list (i.e. ignore nothing);
* "*" may be specified to ignore everything. It stays as the first
* element forever, unless a "!" clears it out.
*/

static char ***ignlist; /* List of files to ignore in update 
* and import */
static char ***ignlist = NULL;

static int igncount; /* Number of active entries */
static int aign_count = 0;

static int ignsize; /* This many slots available (plus 
* one for a NULL) */
static int ignhold = -1; /* Index where first “temporary” item
* is held */

const char *ign_default = "... core RCSLOG tags RCS RCS .make.state
.nse_depinfo * .* cvslog.* CVS CVS.adm .del.* .a .olb .o .obj

#define IGN_GROW 16 /* grow the list by 16 elements at a
* time */

/* Nonzero if we have encountered an -I ! directive, which means one should 
no longer ask the server about what is in CVSROOTADM_IGNORE. */
int ign_inhibit_server;

/* To the “ignore list”, add the hard-coded default ignored wildcards above,
* the wildcards found in $.CVSROOT/.CVSROOT/.cvsignore, the wildcards found in
* variable.
*/

void ign_setup ()
{
    char *home_dir;
    char *tmp;
    ign_inhibit_server = 0;

    /* Start with default list and special case */
    tmp = xstrdup (ign_default);
    ign_add (tmp, 0);
    free (tmp);
}

#define CLIENT_SUPPORT
/* The client handles another way, by (after it does its own ignore file 
processing, and only if !ign_inhibit_server), letting the server
know about the files and letting it decide whether to ignore 
them based on CVSROOTADM_IGNORE. */
if (client_active)
{
    char *file = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM) + sizeof (CVSROOTADM_IGNORE) + 10);
    /* Then add entries found in repository, if it exists */
    (void) sprintf (file, "%s/%s", CVSroot_directory, CVSROOTADM, CVSROOTADM_IGNORE);
    ign_add (file, 0);
    free (file);
}

/* Then add entries found in home dir, (if user has one) and file exists */
home_dir = get_home_dir ();
if (home_dir)
{
    char *file = xmalloc (strlen (home_dir) + sizeof (CVSDOTIGNORE) + 10);
    (void) sprintf (file, "%s/%s", home_dir, CVSDOTIGNORE);
}
```c
ign_add_file (file, 0);
free (file);
}
/* Then add entries found in CVSIGNORE environment variable. */
ign_add (getenv (IGNORE_ENV), 0);
/* Later, add ignore entries found in -I arguments */
}
/* Open a file and read lines, feeding each line to a line parser. Arrange */
/* for keeping a temporary list of wildcards at the end, if the "hold" */
/* argument is set. */
void
ign_add_file (file, hold)
char *file;
int hold;
{ FILE *fp;
char *line = NULL;
size_t line_allocated = 0;
/* restore the saved list (if any) */
if (s_ign_list != NULL)
{ int i;
  for (i = 0; i < s_ign_count; i++)
    ign_list[i] = s_ign_list[i];
  s_ign_count = s_ign_list = NULL;
  s_ign_count = 0;
  free (s_ign_list);
  s_ign_list = NULL;
}
/* is this a temporary ignore file? */
if (hold)
{ /* re-set if we had already done a temporary file */
  if (ign_hold < 0)
  { int i;
    for (i = ign_hold; i < ign_count; i++)
      free (ign_list[i]);
    ign_count = ign_hold;
    ign_list[ign_count] = NULL;
  }
  else
  { ign_hold = ign_count;
  }
}
/* load the file */
fp = CVS_OPEN (file, "r");
if (fp == NULL)
{ if (! existence)
      error (0, errno, "cannot open %s", file);
    return;
} while (getline (&line, &line_allocated, fp) >= 0)
ign_add (line, hold);
if (error (fp))
  error (0, errno, "cannot read %s", file);
if (fclose (fp) < 0)
  error (0, errno, "cannot close %s", file);
free (line);
/* Parse a line of space-separated wildcards and add them to the list. */
void
ign_add (ign, hold)
char *sign;
int hold;
{ if (ign || *(sign)
    return;
  for (sign = ign; ign++)
  { char *mark;
    char save;
    /* ignore whitespace before the token */
}
```
if (isspace (*sign))
    continue;
/*
 * if we find a single character !, we must re-set the ignore list
 * (saving it if necessary). We also catch * as a special case in a
 * global ignore file as an optimization
 */
if (((ign+1) | isize (*sign+1)) & (*sign == '!' || *sign == '*'))
{
    /* permanently reset the ignore list */
    int i;
    for (i = 0; i < igncount; i++)
        free (ignlist[i]);
    igncount = 0;
    ignlist[0] = NULL;
    /* if we are doing a '!', continue; otherwise add the '*' */
    if (*sign == '!')
    {
        ign inhibit_server = 1;
        continue;
    }
    else if (*sign == ' '*)
    {
        /* temporarily reset the ignore list */
        int i;
        if (ignhold >= 0)
        {
            for (i = ignhold; i < igncount; i++)
                free (ignlist[i]);
            ignhold = -1;
        }
        ignlist = (char **) xmalloc (igncount * sizeof (char *));
        for (i = 0; i < igncount; i++)
            ignlist[i] = ignlist[i];
        igncount = igncount;
        ignlist[0] = NULL;
        continue;
    }
    /* If we have used up all the space, add some more */
    if (igncount >= ignelse)
    {
        ign_size += IGN_GROW;
        ign_list = (char **) xrealloc ((char *) ign_list,
                (ignsize + 1) * sizeof (char *));
    }
    /* find the end of this token */
    for (mark = ign; *mark & & isspace (*mark); mark++)
        /* do nothing */ ;
    save = *mark;
    *mark = '0';
    ignlist[igncount++] = xstrdup (ign);
    ignlist[igncount] = NULL;
    *mark = save;
    if (save)
        ign = mark;
    else
        ign = mark - 1;
}
/* Set to 1 if filenames should be matched in a case-insensitive
   fashion. Note that, contrary to the name and placement in ignore.c,
   this is no longer just for ignore patterns. */
int ign case;
/* Return 1 if the given filename should be ignored by update or import. */
int
ign_name (name)
char *name,
{
    char *c = ign list;
    if (c && == NULL)
        return (0);
    if (ign case)
{ /* We do a case-insensitive match by calling fnmatch on copies of
the pattern and the name which have been converted
to lowercase. FIXME: would be much cleaner to just unify this
with the other case-insensitive fnmatch stuff (FOLD in lib/fnmatch.c; os2/fnmatch in emx/system.c). */
char *name_lower;
char *pat_lower;
char *p;
name_lower = xstrdup (name);
for (p = name_lower; *p != '\0'; ++p)
  *p = tolower (*p);
while (*cpp)
{
  pat_lower = xstrdup (++cpp);
  for (p = pat_lower; *p != '\0'; ++p)
    *p = tolower (*p);
  if (CVS_FNMATCH (pat_lower, name_lower, 0) == 0)
    goto matched;
  free (pat_lower);
}
free (name_lower);
return 0;
matched:
free (name_lower);
free (pat_lower);
return 1;
}
else
{
while (*cpp)
  if (CVS_FNMATCH (++cpp, name, 0) == 0)
    return 1;
return 0;
}

/* FIXME: This list of dirs to ignore stuff seems not to be used.
Really? send/ignoreproc and update/ignoreproc both call
ignore_directory and make sure calls ignore_add. No doubt could
use some documentation/testsuite work. */
static char *dir_ign_list = NULL;
static int dir_ign_max = 0;
static int dir_ign_current = 0;

/* Add a directory to list of dirs to ignore. */
void
ignore_add (name)
char *name;
{
  /* Make sure we've got the space for the entry. */
  if (dir_ign_current <= dir_ign_max)
  {
    dir_ign_max += IGN_GROW;
    dir_ign_list =
      (char **) realloc (dir_ign_list, (dir_ign_max + 1) * sizeof (char *));
  }
  dir_ign_list[dir_ign_current] = name;
  dir_ign_current += 1;
}

/* Return nonzero if NAME is part of the list of directories to ignore. */
int
ignore_directory (name)
char *name;
{
  int i;
  if (!dir_ign_list)
    return 0;
  i = dir_ign_current;
  while (i--)
  {
    if (strncmp (name, dir_ign_list[i], strlen (dir_ign_list[i])) == 0)
      return 1;
  }
  return 0;
}
/* 
  * Process the current directory, looking for files not in ILIST and

332 – Source code
void ignore_files (list, entries, update dir, proc)
{
    int subdirs;
    DIR *dirp;
    struct dirent *dp;
    struct stat sb;
    char *file;
    char *dirset;

    /* Set SUBDIRS if we have subdirectory information in ENTRIES. */
    if (entries == NULL)
        subdirs = 5;
    else
    {
        struct stickydirtag *sdtp;
        subdirs = (struct stickydirtag *) entries->list->data;
        subdirs = subdirs % 2 == NULL ? subdirs : subdirs;
    }

    /* If we get called with update_dir set to ".", trim the wildcard */
    if (strcmp (update_dir, CVSDOTWRAPPER) == 0)
        xdir = "";
    else
    {
        xdir = update_dir;
        dirp = CVSOPENDIR (xdir);
        if (dirp == NULL)
            return;
    }

    ignamadd_file (CVSDOTIGNORE, 1);
    wrapadd_file (CVSDOTWRAPPER, 1);

    while ((dp = readdir (dirp)) != NULL)
    {
        file = dp->d_name;
        if (strcmp (file, "") == 0 || strcmp (file, "..") == 0)
            continue;
        if (findnode_in (list, file) == NULL)
            continue;

        if (subdirs)
        {
            Node *node;
            node = findnode_in (entries, file);
            if (node != NULL
                && ((Entnode *) node->data)->type == ENT_SUBDIR)
            {
                char *p;
                int dir;

                /* For consistency with past behaviour, we only ignore
                   this directory if there is a CVS subdirectory.
                   This will normally be the case, but the user may
                   have messed up the working directory somehow. */
                p = xmalloc ((strlen (file) + sizeof CVSADM + 10);
                sprintf (p, "%.x", file, CVSADM);
                dir = idir (p);
                free (p);
                if (dir)
                    continue;
            }

            /* We could be ignoring FIFOs and other files which are neither
               regular files nor directories here. */
            if (ignore.c (file))
                continue;
        }

        if (entry->d_type != DT_DIR)
            continue;
    }

    if (entry->d_type == DT_DIR)
    {
        if (lstat (file, &sb) != -1)
        {
            if (entry->d_type == DT_DIR)
                dp->d_type == DT_DIR || dp->d_type == DT_UNKNOWN &&
            ...
        }
    }
}
endif

S_ISDIR(sb.st_mode))
{
    if (! subdirs)
        char *temp;
        temp = xmalloc(strlen(file) + sizeof(CVSADM) + 10);
        (void) sprintf(temp, "%s/%s", file, CVSADM);
        if (isdir(temp))
        {
            free(temp);
            continue;
        }
        free(temp);
    }
#endif

#define S_ISLNK
else if (
#endif

#define DT_DIR

if (dp->d_type == DT_LNK || dp->d_type == DT_UNKNOWN &&
#endif

#define S_ISLNK(sb.st_mode))
{
    continue;
}
#endif

(*proc)(file, xdir);
}
#endif
}
(void) closedir(dirp);
A.31 import.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * "import" checks in the vendor release located in the current directory into
 * the CVS source repository. The CVS vendor branch support is utilized.
 *
 * At least three arguments are expected to follow the options:
 * repository Where the source belongs relative to the CVSROOT
 * VendorTag Vendor's major tag
 * VendorReleaseTag Tag for this particular release
 *
 * Additional arguments specify more Vendor Release Tags.
 */

#include "cvs.h"
#include "saveded.h"
#include <assert.h>

static char *get_comment PROT0((char *user));
static int addrev PROT0((char *message, RCSNode *rcs, char *vfile, char *vers));
static int addtags PROT0((RCSNode *rcs, char *vfile, char *vtag, int targe, char *tagv[]));
static int importdescend PROT0((char *message, char *vtag, int targe, char *tagv[]));
static int importdescenddir PROT0((char *message, char *dir, char *vtag, int targe, char *tagv[], int inattic));
static void addlog PROT0((char *ch, char *fname));

static int reposlen;
static char *vhead;
static char *vbranch;

static FILE *logfile;
static char *repository;
static int conflicts;
static int use_file_modtime;
static char *keyw1d1d1option = NULL;

static const char *const import_usage[] =
{
    "-d	Use the file's modification time as the time of import.
"
    "-W spec	Wrappers specification line.
"
    "-m msg	Log message.
"
    "-b bra	Vendor branch id.
"
    "-k sub	Set default RCS keyword substitution mode.
"
    "\"{-f \"A.31 import.c\"}"
    "("Specify the --help global option for a list of other help options\")\n"
"
    "Usage: %s %s [-d] [-k subst] [-I ign] [-m msg] [-b branch]\
"
    int argc, argv[];
    char **argv;
    {
        char *message = NULL;
        char *tmpfile;
        char *chp;
        int i, c, msglen, err;
        List *alist;
        Node *p;
        struct logfileinfo *li;

        if (argc <= -1)
            usage (import_usage);

        ign_setup ();
        wrap_setup ();

        vbranch = xstrdup (CVSBRANCH);

        optind = 0;
        while ((c = getopt (argc, argv, "*-Wm?:k:s:1:k:V:*")) != -1)
            {
                switch (c)
                    {
                        case 'Q':
                            case 'q':

#endif SERVER_SUPPORT
*/
Global option requests, so we must ignore them. */

endif

error (1, 0, "-q or -Q must be specified before "%s", command==name);
break;
case 'd':
    usefile[modtime] = 1;
    break;
case 'b':
    free (vbranch);
    vbranch = xstrdup(optarg);
    break;
case 'n':
    ifdef FORCE_USEEDITOR
        useeditor = 1;
    else
        useeditor = 0;
    endif
    message = xstrdup(optarg);
    break;
case 'I':
    ignore[0] = optarg, 0;
    break;
case 'k':
    /* RCS_keyword_flag returns strings of the form -kxx. We
    only use it for validation, so we can free the value
    as soon as it is returned. */
    free (RCS_keyword_flag(optarg));
    keyword[opt] = optarg;
    break;
case 'v':
    vhead[val] = optarg, 0;
    break;
case 't':
    default:
        usage (import_usage);
        break;
    }
110
for (i = 1; i < argc; i++)
    /* check the tags for validity */
    { int j;
        RCS_keyword_tag [argv[i]] = 0;
        for (j = 1, j < i, j++)
            if (strcmp(argv[i], argv[j]) == 0)
                break;
        if (j < i)
            error (1, 0, "tag %s was specified more than once", argv[i]);
    }
/* XXX - this should be a module, not just a pathname */
if (!isabsolute (argv[0]) )
    { 
        if (CVSroot[0] == NULL)
            
            { error (0, 0, "missing CVSROOT environment variable\n");
                error (1, 0, "Set it or specify the -d option to \n",
                    program==name);
            }
            repository = xmalloc (strlen (CVSroot) + strlen (argv[0])
                + 10);
            (void) sprintf (repository, "%s/\n", CVSroot), argv[0]);
            repon_len = strlen (CVSroot);
        else
            repository = xmalloc (strlen (argv[0]) + 5);
            (void) strcpy (repository, argv[0]);
            repon_len = 0;
        }
/* Consistency checks on the specified vendor branch. It must be
 composed of only numbers and dots (\`). Also, for now we only
 * support branching to a single level, so the specified vendor branch
 * must only have two dots in it (like "1.1.1"). */
if (cp = vbranch; cp != '\0'; cp++)
    if (!isdigit (+cp) & & +cp != ' ')
        error (1, 0, "Is not a numeric branch", vbranch);
    if (!numdots (vbranch) != 2 )
        error (1, 0, "Only branches with two dots are supported: \n",
            vbranch);
        vhead = xstrdup (vbranch);
        cp = strrchr (vhead, '\
');
*cp = '\0';

#define CLIENT_SUPPORT
if (client_active)
{
  /* For rationale behind calling start_server before do_editor, see
   * commit.c */
  start_server();
}
#endif

if (use_editor)
{
  do_editor ((char *) NULL, &message, repository,
             (List *) NULL);
}

do_verify (message, repository);
msglen = message == NULL ? 0 : strlen (message);
if (msglen == 0 || message[msglen - 1] == '\n')
{
  char *nm = xmalloc (msglen + 2);
  *nm = '\0';
  if (message != NULL)
  {
    (void) strcpy (nm, message);
    free (message);
  }
  (void) strcat (nm + msglen, "\n");
  message = nm;
}

#define CLIENT_SUPPORT
if (client_active)
{
  int err;

  if (use_filemodtime)
    send_arg("-d");
  if (vbranch[0] == '\0')
    option_with_arg (*-b*, vbranch);
  if (message)
    option_with_arg (*-m*, message);
  if (keyword_opt != NULL)
    option_with_arg (*-k*, keyword_opt);
  /* The only ignore processing which takes place on the server side
   * is the CVSROOT/cvsignore file. But if the user specified -I, the
   * documented behavior is to not process said file. */
  if (ignore_inhibit_server)
  {
    send_arg (*-I*);
    send_arg (*"*");
  }
  wrap_send (1);

  logfp = stdin;
  client_init_setup (repository);
  err = import_dscend (message, argv[0], argc - 2, argv + 2);
  client_import_close ();
  send_server ("SUCCESS", 0);
  err += get_response_and_close ();
  return err;
}
#endif

/* Make all newly created directories writable. Should really use a more
 * sophisticated security mechanism here. */
(void) umask (cvsemask);
make_directories (repository);

/* Create the logfile that will be logged upon completion */
tmpfile = cvsimportname (1);
if ((logfp = CVSCLOSE (tmpfile, "w")) == NULL)
  error (1, errno, "cannot create temporary file \"%s\"", tmpfile);
/* On systems where we can unlink an open file, do so, so it will go
 * away no matter how we exist. FIXME-maybe: Should be checking for
 * errors but I'm not sure which error(s) we get if we are on a system
 * where one can't unlink open files. */
(void) CVSSUNLINK (tmpfile);
(void) fprintf (logfp, "\nVendor Tag:\"%s\", argv[1]);
(void) fprintf (logfp, "Release Tags:\"%s\")
for (i = 2; i < argc; i++)
(void) fprintf (logfp, "%s", argv[0]);

(void) fprintf (logfp, "%s"):

/* Just Do It. */
err = importdescend (message, argv[1], argc - 2, argv + 2);
if (conflicts)
{
  if (really_quiet)
  {
    char buf[80];
    fprintf (buf, "%d conflicts created by this import.
",
conflicts);
    cvs_output (buf, 0);
    cvs_output ("Use the following command to help the merge:
", 0);
    cvs_output ("%s\", vbranch);
    cvs_output ("checkout -j", 0);
    cvs_output (program_name, 0);
    cvs_output ("-j", 0);
    cvs_output (argv[4], 0);
    cvs_output ("--edit", 0);
    cvs_output (argv[6], 0);
  }
  (void) fprintf (logfp, "%d conflicts created by this import:
",
conflicts);
  (void) fprintf (logfp, "Use the following command to help the merge:
",
program_name, argv[1], argv[2], argv[3]);
}
else
{
  if (really_quiet)
  cvs_output ("No conflicts created by this import\n", 0);
  (void) fprintf (logfp, "No conflicts created by this import\n");
}

/* Write out the logfile and clean up. */
ulist = getlist ();
p = getnode ();
p->type = UPDATE;
p->deldelproc = update_delproc;
p->key = xstrdup ("- Imported sources");
li = (struct logfile_info *) xmalloc (sizeof (struct logfile_info));
li->type = T_TITTLE;
li->tag = xstrdup (vbranch);
li->rev_old = li->rev_new = NULL;
p->data = (char *) li;
(void) addnode (ulist, p);
Update_Logfile (repository, message, logfp, ulist); dellist (&ulist);
if (fclose (logfp) < 0)
  error (0, errno, "error closing %a", tmpfile);
/* Make sure the temporary file goes away, even on systems that don’t
let you delete a file that’s in use. */
if (CVS_UNLINK (tmpfile) < 0 && file_existence_error (errno))
  error (0, errno, "$a cannot remove \%a", tmpfile);
free (tmpfile);
if (message)
  free (message);
free (repository);
free (vbranch);
free (vhead);
return (err);
}

/* process all the files in ",", then descend into other directories. */

static int
import_descend (message, vtag, targc, targv)
char *message,
char *vtag,
int targc;
char *targv[];
{
  DIR *dirp;
  struct dirent *dp;
  int err = 0;
  List *dirlist = NULL;

  /* first, load up any per-directory ignore lists */
```c
int add_file (CVSDOTIGNORE, 1);
wrap_add_file (CVSDOTWRAPPER, 1);

if ((dirp = CVSOPENDIR (".*") == NULL)
  
err++;
  
} else
  
while ((dp = readdir (dirp)) != NULL)
  
if (strcmp (dp->d_name, ".") == 0 || strcmp (dp->d_name, "..") == 0)
    continue;

#ifdef SERVER_SUPPORT
  
/* CVS directories are created in the temp directory by server.c because it doesn't special-case import. So don't print a message about them, regardless of -I. */
  
if (server_active && ! strcmp (dp->d_name, CVSADM) == 0)
    continue;
#endif

if (ign_name (dp->d_name))
  
add_log (T1, dp->d_name);
  
continue;

} else
  
if (dp->d_type == DT_DIR
      || (dp->d_type == DT_LNK && !isdir (dp->d_name)))
  
isdir (dp->d_name)
  
& (wrap_name) has (dp->d_name, WRAP_TOCVS)
  }
  
Node *n;
  
if (dirlist == NULL)
    dirlist = getlist();
  
n = getnode();
  
n->key = xstrdup (dp->d_name);
  
addnode (dirlist, n);
  }
else if (dp->d_type == DT_LNK || dp->d_type == DT_LNK &&
#ifdef DT_DIR
  
islink (dp->d_name))
  
add_log (T1, dp->d_name);
  
err++;
  }
  
else
  
#ifdef CLIENT_SUPPORT
  
if (client_active)
    
err += client_process_add_file (message, dp->d_name, vtag, targ, targv, repository, keyword, opt != NULL &&
        keyword[opt[0]] == 'v');
  
#else
  
err += process_add_file (message, dp->d_name, vtag, targ, targv);
  
#endif
  
(void) closedir (dirp);
  }
if (dirlist != NULL)
  
{ Node *head, *p;
    
head = dirlist->list;
    for (p = head->next; p != head; p = p->next)
      
err += import_descend_dir (message, p->key, vtag, targ, targv);
  }

dellist (&dirlist);

return (err);

/*
 * Process the argument import file.
 */
```
*/

static int
process_import_file (message, vfile, vtag, targv)
{
  char *message;
  char *vfile;
  char *vtag;
  char *targv[];

  rcs = xmalloc (strlen (repository) + strlen (vfile) +
                sizeof (RCSEXT) + 5);
  (void) sprintf (rcs, "%s/%s", repository, vfile, RCSEXT);
  if (!isfile (rcs))
    {
      char *attic_name;
      attic_name = xmalloc (strlen (repository) + strlen (vfile) +
                            sizeof (CVSATTIC) + sizeof (RCSEXT) + 10);
      (void) sprintf (attic_name, "%s/%s", repository, CVSATTIC,
                      vfile, RCSEXT);
      if (!isfile (attic_name))
        {
          int retval;
          char *free_opt = NULL;
          char *our_opt = keyword_opt;

          free (attic_name);
          /*
           * A new import source file; it doesn't exist as a .v within the
           * repository nor in the Attic – create it anew.
           */
          add_log ('V', vfile);
        }

        #ifdef SERVER
        /* The most reliable information on whether the file is binary
         * is what the client told us. That is because if the client had
         * the wrong idea about binaryness, it corrupted the file, so
         * we might as well believe the client. */
        if (server_active)
          {
            Node *node;
            List *entries;

            /* Reading all the entries for each file is fairly silly, and
             * probably slow. But I am too lazy at the moment to do
             * anything else. */
            entries = Entries_Open (0, NULL);
            node = findnode_in (entries, vfile);

            if (node != NULL)
              {
                Entnode *entdata = (Entnode *) node->data;
                if (entdata->type == ENT_FILE)
                  {
                    assert (entdata->options[0] == '-v');
                    &entdata->options[1] = 'A';
                    our_opt = xstrdup (entdata->options + 2);
                    free_opt = our_opt;
                  }

                Entries_Close (entries);
              }
        #endif

        retval = addrcsfile (message, rcs, vfile, vhead, our_opt,
                             vbranch, vtag, targv, NULL, 0, NULL, log(p));
        free (free_opt);
        free (rcs);
        return (retval);
    }

    free (attic_name);
    inattic = 1;
}
free (rcs);
/*
 * an rcs file exists. have to do things the official, slow, way.
 */
return (update_rcs_file (message, vfile, vtag, targv, inattic));
}

/*
 * The RCS file exists; update it by adding the new import file to the
 * (possibly already existing) vendor branch.
 */

static int
540 update\-vfile (message, vfile, vtag, targc, targv, inattic)
541 char *message;
542 char *vfile;
543 char vtag;
544 int targc;
545 char *targv[];
546 int inattic;
547 {
548 VersionTS *vers;
549 int letter;
550 char *tocvsPath;
551
552 struct fileinfo finfo;
553
554 memset (&finfo, 0, sizeof finfo);
555 finfo.file = vfile;
556 /* Not used, so don't worry about it. */
557 finfo.upsdir = NULL;
558 finfo.fullname = finfo.file;
559 finfo.repository = repository;
560 finfo.entries = NULL;
561 finfo.rcs = NULL;
562
563 vers = VersionTS (finfo, (char *) NULL, vbranch, (char *) NULL, 1, 0);
564 if (vers->vn_rcs != NULL
565 & RCS_inplace(vers->srcfile, vers->vn_rcs))
566 {
567 int different;
568
569 /* The rcs file does have a revision on the vendor branch. Compare
570 * this revision with the import file; if they match exactly, there
571 * is no need to install the new import file as a new revision to the
572 * branch. Just tag the revision with the new import tags.
573 * 
574 * This is to try to cut down the number of "C" conflict messages for
575 * locally modified import source files.
576 */
577 tocvsPath = wraptocvsprocessfile (vfile);
578 /*FIXME: Why don't we pass tocvsPath to RCS\-semble if it is
579 * different - RCS\-semble (vers->srcfile, vers->vn_rcs, "ko", vfile).
580 if (tocvsPath)
581 if (unlink\-file\-dir (tocvsPath) < 0)
582 error (1, errno, "cannot remove \"", tocvsPath);
583 if (different)
584 {
585 int retval = 0;
586
587 /* The two files are identical. Just update the tags, print the
588 * "U", signifying that the file has changed, but needs no
589 * attention, and we're done.
590 */
591 if (add\-tags (vers->srcfile, vfile, vtag, targv))
592 retval = 1;
593 add\-log ("U", vfile);
594 freesvers\-ts (&vers);
595 return (retval);
596 }
597
598 /* We may have failed to parse the RCS file; check just in case */
599 if (vers->srcfile == NULL ||
600 add\-rev (message, vers->srcfile, vfile, vers->vn_rcs) ||
601 add\-tags (vers->srcfile, vfile, vtag, targv))
602 {
603 freesvers\-ts (&vers);
604 return (1);
605 }
606
607 if (vers->srcfile->branch == NULL || inattic ||
608 strcmp (vers->srcfile->branch, vbranch) != 0)
609 {
610 conflicts++;
611 letter = 'C';
612 }
613 else
614 letter = 'U';
615 add\-log (letter, vfile);
616
617 freesvers\-ts (&vers);
618 return (0);
619 }
620
621 /* Add the revision to the vendor branch */
622 static int
623 add\-rev (message, rcs, vfile, vers)
char *message;
RCSNode *rcs;
char *vfile;
char *vers;

int locked, status, ierrno;
char *tocvsPath;

if (noexec)
    return (0);

locked = 0;
if (vers != NULL)
{
    /* Before RCS.lock existed, we were directing stdout, as well as
       stderr, from the RCS command, to DEVNULL. I wouldn't guess that
       was necessary, but I don't know for sure. */
    /* Earlier versions of this function printed a 'fork failed' error
       when RCS.lock returned an error code. That's not appropriate
       now that RCS.lock is librarified, but should the error text be
       preserved? */

    if (RCS.lock (rcs, vbranch, 1) != 0)
        return (1);
    locked = 1;
    RCS_rewrite (rcs, NULL, NULL);
}

tocvsPath = wrap_tocvs_process_file (vfile);

status = RCS_checkin (rcs, tocvsPath, NULL ? vfile : tocvsPath, message, vbranch,
                      (RCS_FLAGSQUIET | RCS_FLAGSKEEPFILE | RCS_FILEMODTIME : 0)) ;
ierrno = errno;

if ((tocvsPath != NULL) && (unlink (tocvsPath) < 0))
    error (0, errno, "cannot remove %s", tocvsPath);

if (status)
{
    if (noexec)
    {
        ferror (logfp, 0, status == -1 ? ierrno : 0,
                "ERROR: Check-in of %s failed", rcs->path);
        error (0, status == -1 ? ierrno : 0,
                "ERROR: Check-in of %s failed", rcs->path);
    }
    if (locked)
    {
        (void) RCS_unlock(rcs, vbranch, 0);
        RCS_rewrite (rcs, NULL, NULL);
    }
    return (1);
}
    return (0);

/*
 * Add the vendor branch tag and all the specified import release tags to the
 * RCS file. The vendor branch tag goes on the branch root (1.1.1) while the
 * vendor release tags go on the newly added leaf of the branch (1.1.1.1,
 * 1.1.1.2, ...).
 *
 */

static int
add_ttags (rcs, vfile, vtag, targc, targv)
RCSNode *rcs;
char *vfile;
char *vtag;
int targc;
char *targv[];
{
    int i, ierrno;

    vers = 0;
    retcode = 0;
    struct FileInfo info;
    if (noexec)
        return (0);
    if ((retcode = RCS_settag (rcs, vtag, vbranch)) != 0)
    {
        ierrno = errno;
        ferror (logfp, 0, retcode == -1 ? ierrno : 0,
                "ERROR: Failed to set tag %s in %s", vtag, rcs->path);
        error (0, retcode == -1 ? ierrno : 0,
                "ERROR: Failed to set tag %s in %s", vtag, rcs->path);
        return (1);
    }
    RCS_rewrite (rcs, NULL, NULL);
    memset (&info, 0, sizeof info);

    for (i = 0; i < targc; i++)
    {
        char *tag = targv[i];

        /* Check that the tag referenced in the tag array exists... */
        if (tag == NULL)
        {
            error (0, ierrno, "Invalid vendor release tag %s", tag);
            continue;
        }
        if (strcmp (tag, vtag) == 0)
        {
            error (0, ierrno, "Vendor release tag is identical to vendor branch tag.
                        Vendor release tags go on leaf nodes of the branch.
                        Use RCS_merge to move the tag onto the branch root.");
            continue;
        }
        if (strcmp (tag, "branch") == 0)
        {
            error (0, ierrno, "Vendor release tag is identical to vendor branch tag.
                        Vendor release tags go on leaf nodes of the branch.
                        Use RCS_merge to move the tag onto the branch root.");
            continue;
        }
        if (strcmp (tag, "version") == 0)
        {
            error (0, ierrno, "Vendor release tag is identical to vendor branch tag.
                        Vendor release tags go on leaf nodes of the branch.
                        Use RCS_merge to move the tag onto the branch root.");
            continue;
        }
        if (strcmp (tag, "vendor") == 0)
        {
            error (0, ierrno, "Vendor release tag is identical to vendor branch tag.
                        Vendor release tags go on leaf nodes of the branch.
                        Use RCS_merge to move the tag onto the branch root.");
            continue;
        }
    }

    return (0);
}
import.c • 343

ﬁnfo.ﬁle = vﬁle;
/∗ Not used, so don’t worry about it. ∗/
ﬁnfo.update dir = NULL;
ﬁnfo.fullname = ﬁnfo.ﬁle;
ﬁnfo.repository = repository;
ﬁnfo.entries = NULL;
ﬁnfo.rcs = NULL;
vers = Version TS (&ﬁnfo, NULL, vtag, NULL, 1, 0);
for (i = 0; i < targc; i++)
{
if ((retcode = RCS settag (rcs, targv[i], vers−>vn rcs)) == 0)
RCS rewrite (rcs, NULL, NULL);
else
{
ierrno = errno;
fperror (logfp, 0, retcode == −1 ? ierrno : 0,
"WARNING: Couldn't add tag %s to %s", targv[i],
rcs−>path);
error (0, retcode == −1 ? ierrno : 0,
"WARNING: Couldn't add tag %s to %s", targv[i],
rcs−>path);
}
}
freevers ts (&vers);
return (0);

720

730

740

}

750

/∗
∗ Stolen from rcs/src/rcsfnms.c, and adapted/extended.
∗/
struct compair
{
char ∗suﬃx, ∗comlead;
};
static const struct compair comtable[ ] =
{

760

770

780

790

800

/∗
∗ comtable pairs each ﬁlename suﬃx with a comment leader. The comment
∗ leader is placed before each line generated by the $Log keyword. This
∗ table is used to guess the proper comment leader from the working ﬁle’s
∗ suﬃx during initial ci (see InitAdmin()). Comment leaders are needed for
∗ languages without multiline comments; for others they are optional.
∗
∗ I believe that the comment leader is unused if you are using RCS 5.7, which
∗ decides what leader to use based on the text surrounding the $Log keyword
∗ rather than a speciﬁed comment leader.
∗/
{"a", "-- "},
/∗ Ada
∗/
{"ada", "-- "},
{"adb", "-- "},
{"asm", ";; "},
/∗ assembler (MS-DOS) ∗/
{"ads", "-- "},
/∗ Ada
∗/
{"bas", "' "},
/∗ Visual Basic code ∗/
{"bat", ":: "},
/∗ batch (MS-DOS) ∗/
{"body", "-- "},
/∗ Ada
∗/
{"c", " * "},
/∗ C
∗/
{"c++", "// "},
/∗ C++ in all its inﬁnite guises ∗/
{"cc", "// "},
{"cpp", "// "},
{"cxx", "// "},
{"m", "// "},
/∗ Objective-C ∗/
{"cl", ";;; "},
/∗ Common Lisp ∗/
{"cmd", ":: "},
/∗ command (OS/2) ∗/
{"cmf", "c "},
/∗ CM Fortran ∗/
{"cs", " * "},
/∗ C∗
∗/
{"csh", "# "},
/∗ shell
∗/
{"dlg", " * "},
/∗ MS Windows dialog ﬁle ∗/
{"e", "# "},
/∗ eﬂ
∗/
{"epsf", "% "},
/∗ encapsulated postscript ∗/
{"epsi", "% "},
/∗ encapsulated postscript ∗/
{"el", "; "},
/∗ Emacs Lisp ∗/
{"f", "c "},
/∗ Fortran
∗/
{"for", "c "},
{"frm", "' "},
/∗ Visual Basic form ∗/
{"h", " * "},
/∗ C-header ∗/
{"hh", "// "},
/∗ C++ header ∗/
{"hpp", "// "},
{"hxx", "// "},
{"in", "# "},
/∗ for Makeﬁle.in ∗/
{"l", " * "},
/∗ lex (conﬂict between lex and
∗ franzlisp) ∗/
{"mac", ";; "},
/∗ macro (DEC-10, MS-DOS, PDP-11,
∗ VMS, etc) ∗/
{"mak", "# "},
/∗ makeﬁle, e.g. Visual C++ ∗/
{"me", ".\\\" "},
/∗ me-macros t/nroﬀ ∗/
{"ml", "; "},
/∗ mocklisp ∗/
{"mm", ".\\\" "},
/∗ mm-macros t/nroﬀ ∗/
{"ms", ".\\\" "},
/∗ ms-macros t/nroﬀ ∗/


static char *
get_comment (user)

char = user;

{ char = cp, *suffix;
char = suffix, path;
int i;
char = retval;
suffix, path = xmalloc (strlen (user) + 5);
cp = strrchr (user, '.');
if (cp != NULL)
{ cp++;;

/* Convert to lower-case, since we are not concerned about the * case-ness of the suffix. */
(void) strcpy (suffix, path, cp);
for (cp = suffix, path; *cp; cp++)
if (isupper (*cp))
 { *cp = tolower (*cp);
suffix = suffix, path;
}
else
 suffix = **;
/* will use the default */
for (i = 0; i++)
{ if (comtable[i].suffix == NULL)
 { /* Default. Note we'll always hit this case before we ever return NULL. */
 retval = comtable[i].comlead;
 break;
 } if (strcmp (suffix, comtable[i].suffix) == 0)
 { retval = comtable[i].comlead;
 break;
 }
}
free (suffix, path);
return retval;

/* Create a new RCS file from scratch. */
This probably should be moved to rcs.c now that it is called from
places outside import.c.
Return value is 0 for success, or nonzero for failure (in which
case an error message will have already been printed). */
int
addrcs_file (message, rcs, user, addvhead, keyopt,
addvbranch, vtag, targe, targv,
desctext, desclen, remotehp, addlogfp)
/* Log message for the addition. Not used if addvhead == NULL. */
char *message;
/* Filename of the RCS file to create. */
char *rcs;
/* Filename of the file to serve as the contents of the initial
revision. Even if addvhead is NULL, we use this to determine
the modes to give the new RCS file. */
char *user;
/* Revision number of head that we are adding. Normally 1.1 but
could be another revision as long as ADD_VBRANCH is a branch
from it. If NULL, then just add an empty file without any
revisions (similar to the one created by "rcs -i"). */
char *addvhead;
/* Keyword expansion mode, e.g., "b" for binary. NULL means the
default behavior. */
char *key_opt;
/* Vendor branch to import to, or NULL if none. If non-NULL, then
stag should also be non-NULL. */
char *addvbranch;
char *vtag;
int targe;
char *targv[];
/* If non-NULL, description for the file. If NULL, the description
will be empty. */
char *desctext;
sizet desclen;
/* Remote branchpoint, NULL if none */
char *remotehp;
/* Write errors to here as well as via error (), or NULL if we should
use only error (). */
FILE *addlogfp;

{ FILE *fprcs, *fpuser;
struct stat sb;
struct tm *ftm;
timenow;
char altdate[MAXDATELEN];
char *author;
int i, eremo, err = 0;
mode_t mode;
char *tocvsPath;
char *userfile;
char *localopt = keyopt;
char *remoteopt = NULL;
mode_t filemode;

if (localopt == NULL)
{
    if (wrapnamehas (user, WRAP_RCSOPTION))
    {
        localopt = remoteopt = wraprcsoption (user, 0);
    }
}
tocvsPath = wraptocvsprocessfile (user);
userfile = (tocvsPath == NULL ? user : tocvsPath);
/* Opening in text mode is probably never the right thing for the
server (because the protocol encodes text files in a fashion
which does not depend on what the client or server OS is, as
documented in cvssclient.texi), but as long as the server just.
runs on unix it is a moot point. */

/* If PreservePermissions is set, then make sure that the file
is a plain file before trying to open it. Long-standing (although
often unpopular) CVS behavior has been to follow symlinks, so we
maintain that behavior if PreservePermissions is not on.

NOTE: this error message used to be 'cannot fstat', but is now
'cannot lstat'. I don't see a way around this, since we must
stat the file before opening it. -twp */

if (CVS STAT (userfile, &sb) < 0)
    error (i, errno, 'cannot lstat %h', user);
ftype = sb.stat mode & S_IFMT;

fpuser = NULL;
if (fpuser perms || ftype == S_IFREG)
{
    fpuser = CVS FOPEN (userfile,
        ((local_opt != NULL && strcmp (local_opt, "b") == 0)
            ? "rb"
            : "r")
    );
    if (fpuser == NULL)
        { /* not fatal, continue import */
            if (addvhead != NULL)
                fperror (addvhead, ierrno, error (0, errno, "ERROR: cannot read file %h", userfile));
            goto read_error;
        }
    fprcs = CVS FOPEN (rcs, "w+b");
    if (fprcs == NULL)
        {
            ierrno = errno;
            goto write_error_noclose;
        }
/* putadmin() */
/*
if (addvhead != NULL)
    { if (fprintf (fprcs, "head %s:012", addvhead) < 0)
        goto write_error;
    } else
        { if (fprintf (fprcs, "head :012") < 0)
            goto write_error;
        }
    if (remotehp != NULL) {
        if (fprintf (fprcs, "remote_branchpoint %s:012", remotehp) < 0)
            goto write_error;
    } else {
        if (fprintf (fprcs, "remote_branchpoint :012") < 0)
            goto write_error;
    }
    if (addvbranch != NULL)
        { if (fprintf (fprcs, "branch %s:012", addvbranch) < 0)
            goto write_error;
        }
    if (fprintf (fprcs, "access :012") < 0 ||
        fprintf (fprcs, "symbols " ) < 0)
        goto write_error;
    for (i = targc - 1; i >= 0; i--)
        { /* RCS writes the symbols backwards */
            assert (addvbranch != NULL);
            if (fprintf (fprcs, "%s:%s.1",
                targv[i], addvbranch) < 0)
                goto write_error;
        }
    if (addvbranch != NULL)
        { if (fprintf (fprcs, "%s:%s", vtag, addvbranch) < 0)
            goto write_error;
        }
    if (fprintf (fprcs, ":012") < 0)
        goto write_error;
if (fprintf (fprcs, "locks ; strict:\012") < 0 ||
   /* XXX: make sure @ processing works in the RCS file */
   fprintf (fprcs, "comment \[@\012", get_comment (user)) < 0)
{
  goto write_error;
}

if (local_opt != NULL)
{
  if (fprintf (fprcs, "expand \[@\012", local_opt) < 0)
  {
    goto write_error;
  }
}

if (fprintf (fprcs, "\012") < 0)
  goto write_error;

/* Write the revision(s), with the date and author and so on
   (that is "delta" rather than "deltatext" from rcsfile(5)). */
if (addvhead != NULL)
{
  if (use_file_modtime)
    now = sb.st_mtime;
  else
    (void) time (now);
  ftm = gmtime (now);
  (void) strftime (DATEFORM, ftm->tm_year + (ftm->tm_mon < 100 ? 0 : 1000),
                  ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
                  ftm->tm_min, ftm->tm_sec);
  author = getcaller (0);

  if (fprintf (fprcs, "%012Z\012", addvhead) < 0 ||
      fprintf (fprcs, "date %s; author %s; state %s %012",
               altdate1, author, state) < 0)
  goto write_error;

  if (fprintf (fprcs, "branches") < 0)
    goto write_error;

  if (addvbranch != NULL)
  {
    if (fprintf (fprcs, "%s", addvbranch) < 0)
      goto write_error;
  }

  if (fprintf (fprcs, "\012") < 0)
    goto write_error;

  if (fprintf (fprcs, "remote-branches") < 0)
    goto write_error;

  if (fprintf (fprcs, "next \012") < 0)
    goto write_error;

  ifdef PRESERVE_PERMISSIONS_SUPPORT
  /* Store initial permissions if necessary. */
  if (preserve_perm)
  {
    if (file_type == S_IFLNK)
    {
      char *link = xreadlink (userfile);
      if (fprintf (fprcs, "%s\012", link) < 0 ||
          expandfile signs (link, strlen (link), fprcs) < 0 ||
          fprintf (fprcs, "%s\012", link) < 0)
        goto write_error;
      free (link);
    }
  }
  else
    {
      if (fprintf (fprcs, "perms\012", sb.st_gid) < 0)
        goto write_error;
      if (fprintf (fprcs, "group\012", sb.st_gid) < 0)
        goto write_error;
      if (fprintf (fprcs, "permissions\012",
                   sb.st_mode & 07777) < 0)
        goto write_error;
      switch (file_type)
      {
        case S_IFREG: break;
        case S_IFCHR:
        case S_IFBLK:
          if (fprintf (fprcs, "special\012", sb.st_dev) < 0)
            switch (file_type)
            {
              case S_IFREG: break;
              case S_IFCHR:
                if (fprintf (fprcs, "perm\012",
                             "block") < 0)
                  (unsigned long) sb.st_dev) < 0)
goto write_error;
break;
default:
    error (0, 0, "can't import %s: unknown kind of special file", userfile);
}
}
}

if (addvbranch != NULL)
{
    if (fprintf(fprcs, "%012zu", addvbranch) < 0 ||
        fprintf(fprcs, "date %s; author %s; state Exp","012",
            altdate1, author) < 0 ||
        fprintf(fprcs, "branches %012zu") < 0 ||
        fprintf(fprcs, "next %012zu") < 0)
    goto write_error;
}

ifdef PRESERVE_PERMISSIONS_SUPPORT

/* Store initial permissions if necessary. */
if (preserve_perms)
{
    if (file_type == S_IFLNK)
    {
        char *link = xreadlink (userfile);
        if (fprintf(fprcs, "special %s %lu;", link, sb.st_gid) < 0 ||
            expand_at_signs(link, strlen(link), fprcs) < 0 ||
            fprintf(fprcs, "@",012zu) < 0)
        goto write_error;
    }
    else
    {
        if (fprintf(fprcs, "owner %s:1%012zu", sb.st_uid) < 0 ||
            fprintf(fprcs, "group %s:1%012zu", sb.st_gid) < 0 ||
            fprintf(fprcs, "permissions %s:1%012zu", sb.st_mode & 07777 < 0)
        goto write_error;
    }
    switch(file_type)
    {
    case S_IFREG: break;
    case S_IFCHR: break;
    case S_IFBLK:
        if (fprintf(fprcs, "special %s %lu;", "character",
                        "block"), sb.st_rdev) < 0)
        goto write_error;
        break;
    default:
        error (0, 0, "cannot import %s: special file of unknown type", userfile);
    }
}

if (fprintf(fprcs, "%012zu") < 0)
    goto write_error;
}

/* Now write the description (possibly empty). */
if (fprintf(fprcs, "%012zudeclist%012zu") < 0 ||
    fprintf(fprcs, "%s") < 0)
    goto write_error;

desctext != NULL
{
    /* The use of offset size for the second argument is very
        strange, since we are dealing with something which definitely
        fits in memory. */
    if (expand_at_signs(desctext, (off_t) desclen, fprcs) < 0)
    goto write_error;
}

if (fprintf(fprcs, "%012zu/0%012zu") < 0)
    goto write_error;

/* Now write the log messages and contents for the revision(s) (that
    is, "deltatext" rather than "delta" from rcsfile(5)). */
if (addvhead != NULL)
{
    if (fprintf(fprcs, "%012zu/0%012zu", addvhead) < 0 ||
        fprintf(fprcs, "%012zu") < 0)
    goto write_error;
    if (addvbranch != NULL)
{ /* We are going to put the log message in the revision on the branch. So putting it here seems kind of redundant, I guess (and that is what CVS has always done, anyway). */
  if (fprint (fpres, "Initial revision/012") < 0)
    goto write_error;
}
else
  if (expand_at_signs (message, (off_t) strlen (message), fpres) < 0)
    goto write_error;
}
if (fprint (fpres, "\012") < 0)
  if (expand_at_signs (message, (off_t) strlen (message), fpres) < 0)
    goto write_error;
}
/* Now copy over the contents of the file, expanding at signs.
   If preserve_perms is set, do this only for regular files. */
if (fpreserve_perms || file_type == S_IFREG)

char buf[8192];
signed int len;

while (1)
  { len = fread (buf, 1, sizeof buf, fpuser);
    if (len == 0)
      {
        if (ferror (fpuser))
          error (1, errno, "cannot read file \%s for copying", user);
        break;
      }
    if (expand_at_signs (buf, len, fpres) < 0)
      goto write_error;
  }
if (fprint (fpres, "\012") < 0)
  goto write_error;
if (addbranch != NULL) {
  if (fprint (fpres, "\012", addbranch) < 0 ||
      fprint (fpres, "\012", addbranch) < 0 ||
      expand_at_signs (message,
            (off_t) strlen (message), fpres) < 0 ||
      fprint (fpres, "\012", addbranch) < 0 ||
      fprint (fpres, "\012", addbranch) < 0)
    goto write_error;
}
}
if (fclose (fpres) == EOF)
  { errno = errno;
    goto write_error_noclose;
  }
/* Close fpuser only if we opened it to begin with. */
if (fpuser != NULL)
  { if (fclose (fpuser) < 0)
      error (0, errno, "cannot close \%s", user);
  }
/*
   * Fix the modes on the RCS files. The user modes of the original
   * user file are propagated to the group and other modes as allowed
   * by the repository umask, except that all write permissions are
   * turned off.
   */
mode = (unsigned char)
  (ST_MODE | (ST_MODE & S_IROWXU) >> 3 |
   (ST_MODE & S_IWUXR) >> 0) &
   "cvsumask\
   |S_IWWRITE | S_IWGRP | S_IWOTH);
if (chmod (rcs, mode) < 0)
  { errno = errno;
    if (addlogfile != NULL)
      perror (addlogfile, 0, errno,
        "WARNING: cannot change mode of file \%s", rcs);
      error (0, errno, "cannot change mode of file \%s", rcs);
      err++;
  }
if (tocvsPath)
  { if (unlink (tocvsPath) < 0)
      error (0, errno, "cannot remove \%s", tocsvPath);
    if (freeopt != NULL)
      free (freeopt);
  }
```c
write_error:
    ierrno = errno;
    if (fchdir (fpuser) == 0)
        error (0, errno, "cannot close %s", rcs);
write_error = close:
    if (fchdir (fpuser) < 0)
        error (0, errno, "cannot close %s", user);
    if (addlogfp != NULL)
        fperror (addlogfp, 0, ierrno, "ERROR: cannot write file %s", rcs);
    error (0, ierrno, "ERROR: cannot write file %s", rcs);
    if (ierrno == ENOSPC)
        
    read_error:
    if (tocvsPath)
        if (unlink_file_all (tocvsPath) < 0)
            error (0, errno, "cannot remove %s", tocvsPath);
        if (freopt != NULL)
            free (freopt);
        return (err + 1);

int expand signs (buf, size, fp)
char *cp, *next;
FILE *fp;

{ register char *cp, *next;
    cp = buf;
    while ((next = memchr (cp, '!', size)) != NULL)
    {
        int len;
        ++next;
        len = next - cp;
        if (fwrite (cp, 1, len, fp) != len)
            return EOF;
        if (putc ('!', fp) == EOF)
            return EOF;
        cp = next;
        size -= len;
    }
    if (fwrite (cp, 1, size, fp) != size)
        return EOF;

return 1;
}

/* Write an update message to (potentially) the screen and the log file. */
static void
add_log (ch, fname)
int ch;
{ char *fname;
    if (really_quiet) /* write to terminal */
        
    cvs_output (repository + repos_len + 1, 0);
    cvs_output (repos, 1);
    else if (repository[0] == '\n')
        
    cvs_output (repository, 0);
    cvs_output (repos, 1);
    cvs_output (fname, 0);
```
cvs_output ("\n", 1);

if (repos_len)
    /(= write to logfile=)
  (void) fprintf (logfp, "%c %s/%s\n", ch, repository + repos_len + 1, fname);
else if (repository[0])
  (void) fprintf (logfp, "%c %s/%s\n", ch, repository, fname);
else
  (void) fprintf (logfp, "%c %s\n", ch, fname);

/* This is the recursive function that walks the argument directory looking
 * for sub-directories that have CVS administration files in them and updates
 * them recursively.
 *
 * Note that we do not follow symbolic links here, which is a feature!
 */
static int import_descend_dir (message, dir, vtag, targc, targv)
char *message;
char *dir;
char *vtag;
int targc;
char *targv[];
{
struct saved cwd;
char *cp;
int errno, err;
char *rcs = NULL;

if (islink (dir))
    return (0);
if (savecwd (&cwd))
{
    fprintf (logfp, 0, 0, "ERROR: cannot get working directory");
    return (1);
}

/* Concatenate DIR to the end of REPOSITORY. */
if (repository[0] == '\0')
{
    char *new = xstrdup (dir);
    free (repository);
    repository = new;
}
else
{
    char *new = xmalloc (strlen (repository) + strlen (dir) + 10);
    strcpy (new, repository);
    (void) strcat (new, "/");
    (void) strcat (new, dir);
    free (repository);
    repository = new;
}
#endif CLIENT_SUPPORT
if (!quiet && !client_active)
else
if (quiet)
#endif
sendif
error (0, 0, "Importing \%s", repository);

if (CVS_CHDIR (dir) < 0)
{
    errno = errno;
    fprintf (logfp, 0, ierrno, "ERROR: cannot chdir to \%s", repository);
    error (0, ierrno, "ERROR: cannot chdir to \%s", repository);
    err = 1;
    goto out;
}
#endif CLIENT_SUPPORT
if (!client_active && !isdir (repository))
else
if (isdir (repository))
sendif
{
    rcs = xmalloc (strlen (repository) + sizeof (RCSEXT) + 5);
    (void) sprintf (rcs, "%s\n", repository, RCSEXT);
    if (isfile (repository) || isfile(rcs))
    {
        fprintf (logfp, 0, 0, "ERROR: \%s is a file, should be a directory!", repository);
        error (0, 0, "ERROR: \%s is a file, should be a directory!", repository);
        err = 1;
        goto out;
    }
if (noexec == 0 && CVS_MKDIR (repository, 0777) < 0)
  {
    ierrno = errno;
    fprintf (logfp, 0, ierrno,
      /*EBUS: cannot mkdir Is -- not added", repository);*/
    error (0, ierrno,
      /*EBUS: cannot mkdir Is -- not added", repository);*/
    err = 1;
    goto out;
  }
}
}
}
1540 err = import_descend (message, vtag, targc, targv);
out:
if (rcs != NULL)
  free (rcs);
if ((cp = strchr (repository, '/')) != NULL)
  *cp = '\0';
else
  repository[0] = '\0';
if (restorecwd (&cwd, NULL))
  errorexit ()
1550 free (cwd);
return (err);
A.32  lock.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
   Copyright (c) 1989-1992, Brian Berliner
   You may distribute under the terms of the GNU General Public License as
   specified in the README file that comes with the CVS source distribution.
*/

#include "cvs.h"

struct lock {
    /* This is the directory in which we may have a lock named by the
       readlock variable, a lock named by the writelock variable, and/or
       a lock named CVSLCK. The storage is not allocated along with the
       struct lock; it is allocated by the ReaderLock caller or in the
       case of writelocks, it is just a pointer to the storage allocated
       for the >key field. */
    char *repository;
    /* Do we have a lock named CVSLCK? */
    int haveCVSLCK;
    /* Note there is no way of knowing whether the readlock and writelock
       exist. The code which sets the locks doesn't use SIGbeginCritsect

    There are two uses. One is the ability to prevent there from being
    two writers at the same time. This is necessary for any number of
    reasons (fileattr code, probably others). Commit needs to lock the
    whole tree so that nothing happens between the up-to-date check and
    the actual checkin.

    The second use is the ability to ensure that there is not a writer
    and a reader at the same time (several readers are allowed). Reasons
    for this are:

    * Readlocks ensure that once CVS has found a collection of rcs
      files using FindNames, the files will still exist when it reads
      them (they may have moved in or out of the attic).
    * Readlocks provide some modicum of consistency, although this is
      kind of limited—see the node Concurrency in cvs.texinfo.
    * Readlocks ensure that the RCS file does not change between
      RCS_parse and RCS_reparsercsfile time. This one strikes me as
      important, although I haven't thought up what bad scenarios might
      be.
    * Readlocks ensure that we won't find the file in the state in
      which it is in between the calls to addrcsfile and RCS_checkin
      in commat.c (when a file is being added). This state is a state in
      which the RCS file parsing routines in rcs.c cannot parse the file.
    * Readlocks ensure that a reader won't try to look at a
      half-written fileattr file (fileattr is not updated atomically).

    (see also the description of anonymous read-only access in
    "Password authentication security" node in doc/cvs.texinfo).

    While I'm here, I'll try to summarize a few random suggestions
    which periodically get made about how locks might be different:

    1. Check for EROFS. Maybe useful, although in the presence of NFS
       EROFS does not mean that the file system is unchanging.
    2. Provide a means to put the cvslocks in some directory apart from
       the repository (CVSROOT/locks; a -l option in modules; etc.).
    3. Provide an option to disable locks for operations which only
       read (see above for some of the consequences).
    4. Have a server internally do the locking. Probably a good
       long-term solution, and many people have been working hard on code
       changes which would eventually make it possible to have a server
       which can handle various connections in one process, but there is
       much, much work still to be done before this is feasible.
    5. Like #4 but we shared memory or something so that the servers
       merely need to all be on the same machine. This is a much smaller
       change to CVS (it functions much like #2; shared memory might be an
       unneeded complication although it presumably would be faster). */
static void remove_locks(void);
static int reader_exist(char *repository);
static int set_lock(struct lock *lock, int will_wait);
static void clear_lock(struct lock *lock);
static void set_lock_name(struct lock *stat, struct stat *statp);
static int set_writelock(proc PROTO((Node * p, void * closure)));
static int unsetproc PROTO((Node * p, void * closure));
static int write_lock PROTO ((struct lock *lock));
static void lock_simple_remove PROTO ((struct lock * lock));
static void lock_wait PROTO (char * repository);
static void lock_obtained PROTO (char * repository);

/* Malloc'd array containing the username of the whoever has the lock.
   Will always be non-NULL in the cases where it is needed. */
static char * lockers_name;
/* Malloc'd array specifying name of a readlock within a directory.
   Or NULL if none. */
static char * readlock;
/* Malloc'd array specifying name of a writelock within a directory.
   Or NULL if none. */
static char * writelock;
/* Malloc'd array specifying the name of a CVSLCK file (absolute pathname).
   Will always be non-NULL in the cases where it is used. */
static char * masterlock;
static List * locklist;

define L_OK 0 /* success */
define L_ERROR 1 /* error condition */
define L_LOCKED 2 /* lock owned by someone else */

/* This is the (single) readlock which is set by Reader_Lock. The
   repository field is NULL if there is no such lock. */
static struct lock global_readlock;

/* List of locks set by lock_free_for_write. This is redundant
   with locklist, sort of. */
static List * lock_free_list;

/* If we set locks with lock_free_for_write, then locked_dir contains
   the malloc'd name of the repository directory which we have locked.
   locked_list is the same thing packaged into a list and is redundant
   with locklist the same way that lock_free_list is. */
static char * locked_dir;
static List * locked_list;

/* Clean up all outstanding locks */

void lock_cleanup()
{
    remove_locks();

dellist (lockers_readlist);
    if (locked_dir != NULL)
        dellist (locked_list);
    free (locked_dir);
    locked_dir = NULL;
    locked_list = NULL;
}

/* Remove locks without discarding the lock information */

static void remove_locks()
{
    /* clean up simple locks (if any) */
    if (global_readlock.repository != NULL)
        lock_simple_remove (&global_readlock);
        global_readlock.repository = NULL;

    /* clean up multiple locks (if any) */
    if (locklist != (List *) NULL)
        (void) walklist (locklist, unlock_proc, NULL);
        locklist = (List *) NULL;
}

/* walklist proc for removing a list of locks */
static int unlock_proc (p, closure)
    Node *p;
    void *closure;
    {
        lockimplremove (struct lock *p->data);
        return (0);
    }

    /* Remove the lock files. */

    static void lockimplremove (lock)
    struct lock *lock;
    {
        char *tmp;
        /
        If readlock is set, the lock directory *might* have been created,
        but since Reader_lock doesn't use SIG_beginCrSect the way that set_lock
        does, we don't know that. That is why we need to check for
        existence_error here. */
        if (readlock != NULL)
            {
                tmp = xmalloc (strlen (lock->repository) + strlen (readlock) + 10);
                (void) sprintf (tmp, "%s/%s", lock->repository, readlock);
                if (CVS_UNLINK (tmp) < 0 && !existence_error (errno))
                    error (0, errno, "failed to remove lock %s", tmp);
                free (tmp);
            }
        /
        If writelock is set, the lock directory *might* have been created,
        but since write_lock doesn't use SIG_beginCrSect the way that set_lock
        does, we don't know that. That is why we need to check for
        existence_error here. */
        if (writelock != NULL)
            {
                tmp = xmalloc (strlen (lock->repository) + strlen (writelock) + 10);
                (void) sprintf (tmp, "%s/%s", lock->repository, writelock);
                if (CVS_UNLINK (tmp) < 0 && !existence_error (errno))
                    error (0, errno, "failed to remove lock %s", tmp);
                free (tmp);
            }
        if (lock->have_lckdir)
            {
                tmp = xmalloc (strlen (lock->repository) + sizeof (CVSLCK) + 10);
                SIG_beginCrSect ();
                if (CVS_RMDIR (tmp) < 0)
                    error (0, errno, "failed to remove lock dir %s", tmp);
                lock->have_lckdir = 0;
                SIG_endCrSect ();
                free (tmp);
            }
        /
        * Create a lock file for readers
        */
        int Reader.Lock (xrepository)
            char *xrepository;
            {
                int err = 0;
                FILE *fp;
                char tmp;
                if (noexec)
                    return (0);
                /
                if (global_readlock.repository != NULL)
                    {
                        error (0, 0, "Reader.Lock called while read locks set - Help!");
                        return (1);
                    }
                if (readlock == NULL)
                    {
                        readlock = xmalloc (strlen (hostname) + sizeof (CVSRFL) + 40);
                        (void) sprintf (readlock, "%s.%ld", CVSRFL, getpid());
                    }
                /
                if (HAVE_LONG_FILE_NAMES)
                    "%s.%ld", CVSRFL, hostname,
                    else
                        "%s.%ld", CVSRFL,
                endif
                    (long) getpid();
                }
        /
        /* remember what we're locking (for Lock_checkup) */
global readlock.repository = xrepository;

/* get the lock dir for our own */
if (set.lock ((global)readlock, 1) != L_OK)
{
    error (0, 0, "failed to obtain dir lock in repository "/", xrepository);
    if (readlock != NULL)
        free (readlock);
    readlock = NULL;
    /* We don't set global readlock.repository to NULL. I think this
     * only works because recurse.c will give a fatal error if we return
     * a nonzero value. */
    return (1);
}

/* write a read-lock */
tmp = xmalloc (strlen (xrepository) + strlen (readlock) + 10);
(void) sprintf (tmp, "%s/%s", xrepository, readlock);
if (((fp = CVS_POPEN (tmp, "r+")) == NULL || fclose (fp) == EOF)
    error (0, errno, "cannot create read lock in repository "/", xrepository);
    if (readlock != NULL)
        free (readlock);
    readlock = NULL;
    err = 1;
}
free (tmp);

/* free the lock dir */
clear_lock (&global_readlock);
return (err);
}

/* Lock a list of directories for writing */
static char *lockerrorrepos;
static int lockerror;

static int WriteLock PROTO ((List * list));

static int WriteLock (list)
List *list;
{
    char *waitrepos;
    if (noexec)
        return (0);
    /* We only know how to do one list at a time */
    if (locklist != (List *) NULL)
        error (0, 0, "WriteLock called while write locks set - Help!");
    return (1);
}
waitrepos = NULL;
for (;)
{
    /* try to lock everything on the list */
    lockerror = L_OK;
    / * wait for set_listlock Proc */
    lockerrorrepos = (char *) NULL; / * not for set_listlock Proc */
    locklist = list; / * wait for LockCleanup */
    if (lockersname != NULL)
        free (lockersname);
    lockersname = xstrdup ("unknown");
(void) walklist (list, set_listlockproc, NULL);
switch (lock_error)
{
    case L_LOCKED: /* Someone already had a lock */
        remove_locks (); / * clean up any locks we set */
        lockwait (lockerrorrepos); / * sleep a while and try again */
        waitrepos = xstrdup (lockerrorrepos);
        continue;
    case L_OK: /* we got the locks set */
        if (waitrepos != NULL)
            free (waitrepos);
        LockCleanup (); / * clean up any locks we set */
        error (0, 0, "lock failed - giving up");
        return (1);
    case L_ERROR: /* Real Error */
        if (waitrepos != NULL)
            free (waitrepos);
        LockCleanup (); / * clean up any locks we set */
        error (0, 0, "lock failed - giving up");
        return (1);
}

350
{    lock_obtained (wait_repos);
    free (wait_repos);
}    return (0);
}
default:
    if (wait_repos != NULL)
        free (wait_repos);
    error (0, 0, "unknown lock status %d in Writer_Lock",
        lock_error);
    return (1);
}

/* walklist proc for setting write locks */
static int
set_writelock_proc (p, closure)

void *p;
void *closure;
{
    /* if some lock was not OK, just skip this one */
    if (lock_error != L_OK)
        return (0);
    /* apply the write lock */
    lock_error_repos = p->key;
    lock_error = writelock ((struct lock *)p->data);
    return (0);
}

/* Create a lock file for writers returns L_OK if lock set ok, L_LOCKED if
lock held by someone else or L_ERROR if an error occurred */
static int
write_lock (lock)
struct lock *lock;

int status;
FILE *fp;
char *tmp;

if (write_lock == NULL)
{
    write_lock = xmalloc (strlen (hostname) + sizeof (CVSWFL) + 40);
    (void) sprintf (write_lock,
        "%s/%s", CVSWFL, hostname,
        "%s.WFL", hostname,
        (long) getpid());
}

/* make sure the lock dir is ours (not necessarily unique to us!) */
/* status = set_lock (lock, 0); */
if (status == L_OK)
{
    /* we now own a writer - make sure there are no readers */
    if (readers_exist (lock->repository))
    {
        /* clean up the lock dir if we created it */
        if (status == L_OK)
        {
            clear_lock (lock);
        }
    }
    /* indicate we failed due to read locks instead of error */
    return (L_LOCKED);
}

/* write the write-lock file */
/tmp = xmalloc (strlen (lock->repository) + strlen (write_lock) + 10);
(void) sprintf (tmp, "%s/%s", lock->repository, write_lock);
if ((fp = CVS=fopen (tmp, "w")) == NULL || fclose (fp) == EOF)
{
    int xerrno = errno;
    if ( CVS_UNLINK (tmp) < 0 && !existence_error (errno))
        error (0, errno, "failed to remove lock %s", tmp);
    /* free the lock dir if we created it */
    if (status == L_OK)
    {
        clear_lock (lock);
    }
}
/* return the error */
error (0, errno, "cannot create write lock in repository '1s'",
lock->repository);
free (tmp);
return (L_ERROR);
} free (tmp);
return (L_OK);
} else
/ * readers_exist() returns 0 if there are no reader lock files remaining in *
* the repository; else 1 is returned, to indicate that the caller should *
* sleep a while and try again. */
static int
readers_exist (repository)
470 char *repository;
{ char *line;
  DIR *dirp;
  struct dirent *dp;
  struct stat sb;
  int ret = 0;
#endif /* CVSSMUDGELOCKS */
again:
480 if (dirp = CVS_OPENDIR (repository)) == NULL)
  error (1, 0, "cannot open directory '1s'", repository);
  errno = 0;
while ((dp = readdir (dirp)) != NULL)
  { if (CVS_FNMATCH (CVSRFLPAT, dp->d_name, 0) == 0)
    { 
      line = xmalloc (strlen (repository) + strlen (dp->d_name) + 5);
      (void) sprintf (line, "%s/%s", repository, dp->d_name);
      if (CVS_STAT (line, &sb) != -1)
      { /* If the create time of the file is more than CVSLCKAGE */
        /* seconds ago, try to clean-up the lock file, and if */
        /* successful, re-open the directory and try again. */
        /**
        if (now >= (sb.st_ctime + CVSLCKAGE) && CVS_UNLINK (line) != -1)
          { (void) closedir (dirp);
            free (line);
            goto again;
          }
        */
endif /* CVSSMUDGELOCKS */
500 endif /* CVSSMUDGELOCKS */
500  { /* If the file doesn't exist, it just means that it disappeared */
    /* between the time we did the readdir and the time we did */
    /* the stat. */
    if (existence_error (errno))
      error (0, errno, "cannot stat '1s'", line);
      errno = 0;
      free (line);
      ret = 1;
      break;
    } errno = 0;
  }
530 if (errno == 0)
    error (0, errno, "error reading directory '1s'", repository);
  closedir (dirp);
return (ret);
} /* Set the static variable lockers_name appropriately, based on the stat */
static void
set_lockers_name (statp)
{
    struct passwd *pw;
    if (lockers_name != NULL)
        free (lockers_name);
    if ((pw = (struct passwd *) getpwnam (statp->st_uid)) !=
        (struct passwd *) NULL)
        
        if (lockers_name = xstrdup (pw->pw_name);
    } else
        
        (void) sprintf (lockers_name, "uid%lu", (unsigned long) statp->st_uid);
    }
*/

static int
set_lock (lock, will_wait)
{
    int waited;
    struct stat sb;
    mode_t omask;
    #ifdef CVS_FUDGELOCKS
    time_t now;
    #endif
    masterlock = xmalloc (strlen (lock->repository) + sizeof (CVSLCK) + 10);
    (void) sprintf (masterlock, "%s/%s", lock->repository, CVSLCK);
    /∗ Persistently tries to make the directory “lckdir”, which serves as a ∗
     * lock. If the create time on the directory is greater than CVSLCKAGE ∗
     * seconds old, just try to remove the directory. ∗/
    waited = 0;
    lock->have_lckdir = 0;
    for (;;) {
        int status = -1;
        omask = umask (cvsumask);
        SIGbeginCrsct ();
        if (CVS_STAT (masterlock, &sb) < 0)
            
            if (existence_error (errno))
                continue;
            (void) umask (omask);
            if (status != -1)
                return status;
        SIGendCrsct ();
        /∗ Find out who owns the lock. If the lock directory is ∗
         * non-existent, re-try the loop since someone probably just ∗
         * removed it (thus releasing the lock). ∗/
        if (errno != EEXIST)
            
            error (0, errno, "failed to create lock directory in repository "lock->repository",
            lock->repository);
            return (L_ERROR);
        if (errno != EEXIST)
            
            if (lock->have_lckdir)
                goto out;
        SIGendCrsct ();
        /∗ Note that it is up to the callers of set_lock() to arrange for signal ∗
         * handlers that do the appropriate things, like remove the lock ∗
         * directory before they exit. ∗/
        waited = 0;
        lock->have_lckdir = 0;
    }
630  
#define CVS_FUDGELOCKS
  /*
   * If the create time of the directory is more than CVSLCKAGE seconds
   * ago, try to clean-up the lock directory, and if successful, just
   * quietly retry to make it.
   */
  (void) time (&now);
  if (now >= (sb st ctime + CVSLCKAGE))
  { if (CVS_RMDIR (masterlock) == 0)
      continue; }
  */
  */
  /* set the lockers name */
  set locker name (&sb);
  /* if he wasn't willing to wait, return an error */
  if (! will wait)
    return (LOCKED);
  lock wait (lock->repository);
  waited = 1;
  }
  */
  /* Clear master lock. We don't have to recompute the lock name since
  clear lock is never called except after a successful set lock().
  */
  static void
  clear lock (lock)
  struct lock *lock;
  {
    SIG_beginCrSect ();
    if (CVS_RMDIR (masterlock) < 0)
      error (0, errno, "failed to remove lock dir '%s'", masterlock);
    lock->have lckdir = 0;
    SIG_endCrSect ();
  }
  */
  /* Print out a message that the lock is still held, then sleep a while.
  */
  static void
  lock wait (repos)
  {
    time tnow;
    (void) time (&now);
    error (0, 0, "[%8.8s] waiting for %s's lock in %s",
      ctime (&now) + 11, lockers name, repos);
    /* Call cvs_flusherr to ensure that the user sees this message as
     * soon as possible. */
    cvs_flusherr ();
    (void) sleep (CVSLCKSLEEP);
  }
  */
  /* Print out a message when we obtain a lock.
  */
  static void
  lock obtained (repos)
  {
    time tnow;
    (void) time (&now);
    error (0, 0, "[%8.8s] obtained lock in %s",
      ctime (&now) + 11, repos);
    /* Call cvs_flusherr to ensure that the user sees this message as
     * soon as possible. */
    cvs_flusherr ();
  }
  */
  static int
  lock_filesdoneproc PROTO ((void *callerdat, int err,
    char *repository, char *update dir, List *entries));
  /*
   * Create a list of repositories to lock
   */
  /* ARGUSED */
  static int
  lock_filesdoneproc (callerdat, err, repository, update dir, entries)
  void *callerdat;
  int err;
  char *repository;
  char *update dir;
  List *entries;
{ Node *p;
    p = getnode();
    p->type = LOCK;
    p->key = x_strdup(repository);
    p->data = xmalloc(sizeof(struct lock));
    (struct lock *)p->data->repository = p->key;
    (struct lock *)p->data->have_lckdir = 0;
    /* FIXME-KRP: this error condition should not simply be passed by. */
    if (p->key == NULL || addnode (lock_tree_list, p) != 0)
        freenode (p);
    return (err);
}

void lock_tree_for_write (argc, argv, local, aflag)
int argc;
char **argv;
int local;
aflag;
{ int err;
    /* Run the recursion processor to find all the dirs to lock and lock all
     * the dirs
     */
    lock_tree_list = getlist ();
    err = start_recursion ((FILEPROC) NULL, lock_filesdoneproc,
                            (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL, argc,
                            argv, local, W_LOCAL, aflag, 0, (char *) NULL, 0);
    sortlist (lock_tree_list, fsortcmp);
    if (WriterLock (lock_tree_list) != 0)
        error (0, 0, "lock failed - giving up");
}

/* Lock a single directory in REPOSITORY. It is OK to call this if
   a lock has been set with lock_dir_for_write; the new lock will replace
   the old one. If REPOSITORY is NULL, don't do anything. */
void lock_dir_for_write (repository)
char *repository;
{ if (repository != NULL && locked_dir == NULL || strcmp (locked_dir, repository) != 0))
    { Node *node;
        if (locked_dir != NULL)
            LockCleanup ();
        locked_dir = x_strdup (repository);
        locked_list = getlist ();
        node = getnode ();
        node->type = LOCK;
        node->key = x_strdup (repository);
        node->data = xmalloc (sizeof (struct lock));
        (struct lock *)node->data->repository = node->key;
        (struct lock *)node->data->have_lckdir = 0;
        (void) addnode (locked_list, node);
        WriterLock (locked_list);
    }
A.33 log.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Print Log Information
 */

/* This line exists solely to test some pcl-cvs/ChangeLog stuff. You can delete it, if indeed it's still here when you read it. -Karl */

/* Prints the RCS "log" (rlog) information for the specified files. With no argument, prints the log information for all the files in the directory (recursive by default). */

/*
#include "cvs.h"
*/

/* This structure holds information parsed from the -r option. */
struct option_revlist
{
    /* The next -r option. */
    struct option_revlist *next;
    /* The first revision to print. This is NULL if the range is `rev', or if no revision is given. */
    char *first;
    /* The last revision to print. This is NULL if the range is `rev:', or if no revision is given. If there is no colon, first and last are the same. */
    char *last;
    /* Nonzero if there was a trailing `.', which means to print only the head revision of a branch. */
    int branchhead;
};

/* This structure holds information derived from option_revlist given a particular RCS file. */
struct revlist
{
    /* The next pair. */
    struct revlist *next;
    /* The first numeric revision to print. */
    char *first;
    /* The last numeric revision to print. */
    char *last;
    /* The number of fields in these revisions (one more than numdots). */
    int fields;
};

/* This structure holds information parsed from the -d option. */
struct datelist
{
    /* The next date. */
    struct datelist *next;
    /* The starting date. */
    char *start;
    /* The ending date. */
    char *end;
    /* Nonzero if the range is inclusive rather than exclusive. */
    int inclusive;
};

/* This structure is used to pass information through start_recursion. */
struct log_data
{
    /* Nonzero if the -R option was given, meaning that only the name of the RCS file should be printed. */
    int nameonly;
    /* Nonzero if the -h option was given, meaning that only header information should be printed. */
    int header;
    /* Nonzero if the -t option was given, meaning that only the header and the descriptive text should be printed. */
    int longheader;
    /* Nonzero if the -N option was seen, meaning that tag information should not be printed. */
    int notags;
    /* Nonzero if the -b option was seen, meaning that only revisions on the default branch should be printed. */
    int defaultbranch;
    /* If not NULL, the value given for the -r option, which lists sets of revisions to be printed. */
    struct option_revlist *revlist;
/* If not NULL, the date pairs given for the -d option, which
  select date ranges to print. */
struct datelist *datelist;
/* If not NULL, the single dates given for the -d option, which
  select specific revisions to print based on a date. */
struct datelist *singedatelist;
/* If not NULL, the list of states given for the -s option, which
  only prints revisions of given states. */
List *statealist;
/* If not NULL, the list of login names given for the -w option, 
  which only prints revisions checked in by given users. */
List *authorlist;

/* This structure is used to pass information through walklist. */
struct log_data {
  struct log_data *log_data;
  struct revlist *revlist;
  RCSNode *rl;}

static Dtype log_dirproc PROTO ((void) callerdat,
  char *dir, 
  char *repository, char *update_dir, 
  List *entries));
static int log_fileproc PROTO ((void) callerdat, struct file_info *finfo));
static struct option *revlist PROTO (const char *);
static void log_parse_date PROTO (struct log_data *, const char *);
static void log_parse_prot (List **, const char *);
static struct revlist *log_expand_prot PROTO ((RCSNode *,
  struct option *revlist *,
  int));
static void log_free_prot PROTO ((struct revlist *));
static int log_count_requested PROTO ((struct log_data *,
  struct revlist *,
  RCSNode *, RCSVers *));
static int log_symbol PROTO ((Node *, void *));
static int log_count PROTO ((Node *, void *));
static int log_count_prot (Node *, void *));
static int log_count_prot (Node *, void *));
static void log_free_prot (struct log_data *, struct revlist *
  RCSNode *, const char *));
static void log_branch PROTO ((struct log_data *,
  struct revlist *,
  RCSNode *, const char *));
static void log_version PROTO ((struct log_data *,
  struct revlist *,
  RCSNode *, RCSVers *, int));
static int log_branch PROTO ((Node *, void *));
static int version compare PROTO ((const char *,
  const char *, int));

static const char *const log_usage[] = {
  "Usage: \%s [-1[branch] [-r[revisions]] [-d dates] [-s states]]\n", 
  "-1[branch] only prints one branch, \"\n", 
  "-r[revisions] prints revisions \n", 
  "-d[date] only prints revisions on the default branch.\n", 
  "-s[states] only prints revisions of given states.\n", 
  "-w[logins] only prints revisions checked in by given users.\n", 
  "(Specify the --help global option for a list of other help options)\n", 
  NULL}

int

int optind = 0;
while ((c = getopt (argc, argv, "\bh1:br:rs:tw:\")) != 
  break;

switch (c) {
  case 'b':
    log_data.default_branch = 1;
    break;
```c
180  case 'd':
181      log_parse_date (&log_data, optarg);
182      break;
183  case 'h':
184      log_data.header = 1;
185      break;
186  case 'l':
187      log_data.local = 1;
188      break;
189  case 'n':
190      log_data.notags = 1;
191      break;
192  case 'r':
193      rl = log_parse_revlist (optarg);
194      for (prl = &log_data.revlist; 
195       prl != NULL; 
196       prl = &(*prl)->next)
197          ;
198      *prl = rl;
199      break;
200  case 's':
201      log_parse_list (&log_data.statelist, optarg);
202      break;
203  case 't':
204      log_data.long_header = 1;
205      break;
206  case 'u':
207      if (optarg != NULL)
208          log_parse_list (&log_data.authorlist, optarg);
209      else 
210          log_parse_list (&log_data.authorlist, getcaller ());
211      break;
212  case '?':
213      default: 
214          usage (log_usage);
215      break;
216  }
220  wrap_setup ();
221
222  ifdef CLIENT_SUPPORT
223      if (client_active)
224          {
225              int i;
226
227              /* We're the local client. Fire up the remote server. */
228              start_server ()
229          }
230  endif
231
232      for (i = 1; i < argc && argv[i][0] == '-' || i++)
233          send_arg (argv[i]);
234      send_subnames (argc - i, argv + i, SEND_EXPAND_WILD);
235      send_lists (argc - i, argv + i, local, SEND_NOCONTENTS);
236      send_server ('log/012', 0);
240      err = get_responses_and_close ();
241      return err;
243      send_if
244          err = start_recursion (log_dieproc, (FILESDONEPROC) NULL, log_dirproc,
245          (DIRLEAVEPROC) NULL, (void *) &log_data, 
246          argc = optind, argv = optind, local, 
247          W LOCAL | W REPOS | W ATTIC, i, 
248          (char *) NULL, 1);
250      return (err);
252      */
253      /* Parse a revision list specification. */
254      */
255      static struct option_revlist *
256          log_parse_revlist (argstring)
257          const char *targeting;
260      {
261          char *copy;
262          struct option_revlist *ret, **spr;
263
266          /* Unfortunately, rlog accepts -r without an argument to mean that 
267          latest revision on the default branch, so we must support that 
268          for compatibility. */
269          if (argstring == NULL) 
270```
ret = (struct option_revlst *) xmalloc (sizeof *ret);
ret->first = NULL;
ret->last = NULL;
ret->next = NULL;
ret->branchhead = 0;
return ret;
}
ret = NULL;
pr = &ret;

// Copy the argument into memory so that we can change it. We
don't want to change the argument because, at least as of this
writing, we will use it if we send the arguments to the server.
We never bother to free up our copy. */
copy = xstrdup (argstring);
while (copy != NULL)
{
    char *comma;
    char *cp;
    char *first, *last;
    struct option_revlst *r;
    comma = strchr (copy, ',');
    if (comma != NULL)
        *comma++ = '\0';
    first = copy;
    cp = strchr (copy, ':');
    if (cp == NULL)
        last = copy;
    else
    {
        *cp++ = '\0';
        last = cp;
    }
    if (*first == '\0')
        first = NULL;
    if (*last == '\0')
        last = NULL;
    r = (struct option_revlst *) xmalloc (sizeof *r);
r->next = NULL;
r->first = first;
r->last = last;
if ((first != last ||
    strlen (first) - 1] != ',')
    {
        r->branchhead = 0;
    }
else
    {
        r->branchhead = 1;
        first[strlen (first) - 1] = '\0';
    }
    *pr = r;
    pr = &r->next;
copy = comma;
}
return ret;

/* Parse a date specification. */

static void
log_parse_date (log_data, argstring)
struct log_data *log_data;
const char *argstring;
{
    char *orig_copy, *copy;
    /* Copy the argument into memory so that we can change it. We
don't want to change the argument because, at least as of this
writing, we will use it if we send the arguments to the server. */
copy = xstrdup (argstring);
orig_copy = copy;
while (copy != NULL)
{
    struct datelist *nd, *pd;
    char *cpend, *cp, *ds, *de;
    nd = (struct datelist *) xmalloc (sizeof *nd);
    cpend = strchr (copy, ';');
    if (cpend != NULL)
Source code

```c
++ = '\0';

if ((cp = strchr (copy, '>')) != NULL) {
  ++cp;
  nd->inclusive = 1;
  if (*cp == 'a') {
    ds = cp;
    de = copy;
  } else if ((cp = strchr (copy, '<')) != NULL) {
    ++cp;
    nd->inclusive = 1;
    if (*cp == 'a') {
      ds = copy;
      de = cp;
    } else {
      ds = NULL;
      de = copy;
    }
  } else {
    ds = NULL;
    de = copy;
  }
}

if (ds == NULL) nd->start = NULL;
else if (*ds != '\0') nd->start = Make_Date (ds);
else {
  nd->start = Make_Date (*"1/1/1970 UTC");
}

if (*de != '\0') nd->end = Make_Date (de);
else {
  nd->end = Make_Date (*"2038-01-01");
}

nd->next = *pd;
*pd = nd;

copy = cpend;

free (orig_copy);
```

/*
 * Parse a comma separated list of items, and add each one to *PLIST.
 * static void
 * log_parse_list (plist, argstring)
 * List **plist;
 * const char *argstring;
 */

while (1) {
  Node *p;
  char *cp;
  p = getnode ();
  cp = strchr (argstring, ',');
if (cp == NULL)
    p->key = xstrdup(argstring);
else
    {
        size_t len;
        len = cp - argstring;
        p->key = xmalloc(len + 1);
        strncpy(p->key, argstring, len);
        p->key[len + 1] = '\0';
    }

if (*plist == NULL)
    *plist = getlist();
if (addnode(*plist, p) != 0)
    freenode(p);
if (cp == NULL)
    break;
argstring = cp + 1;

static int printlock_proc(PROTO((Node *, void *)));
static int printlock_proc(lock, foo)
    Node *lock;
    void *foo;
{
    cvs_output("
        \n        ");
    cvs_output(lock->data, 0);
    cvs_output(" ", 1);
    cvs_output(lock->key, 0);
    return 0;
}

/* Do an rlog on a file */

static int log_fileproc(callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    struct log_data *log_data = (struct log_data *) callerdat;
    Node *p;
    RCSNode *rcsfile;
    char buf[50];
    struct revlist *revlist;

    if ((rcsfile = finfo->rcs) == NULL)
    {
        /* no rcs file. What do we know about this file? */
        p = findnode(finfo->entries, finfo->file);
        if (p != NULL)
        {
            Entnode *e;
            e = (Entnode *) p->data;
            if (e->version[0] == '0' && e->version[1] == '\0')
            {
                if (really_quiet)
                    error(0, 0, " último ha sido agregado, pero no committing ",
                        finfo->file);
                return(0);
            }
        }
    }

    if (really_quiet)
        error(0, 0, " no se ha conocido nada sobre %s", finfo->file);
    return(1);
}

if (log_data->nameonly)
{
    cvs_output(rcsfile->path, 0);
    cvs_output(" \n", 1);
    return 0;
}

/* We will need all the information in the RCS file. */
RCSfullyparse(rcsfile);

/* Turn any symbolic revisions in the revision list into numeric revisions. */
revlist = log_expand_revlist(rcsfile, log_data->revlist,
data

log_data->default_branch);

/* The output here is intended to be exactly compatible with the
output of log. I'm not sure whether this code should be here
or in rcs.c. I put it here because it is specific to the log
function, even though it uses information gathered by the
functions in rcs.c. */
cvs_output("\n", 1);
cvs_output("RCS file: ", 0);
cvs_output(rcsfile->path, 0);
cvs_output("Working file: ", 0);
if (finfo->update_file[0] == 'l')
cvs_output(finfo->file, 0);
else
{
cvs_output(finfo->update_file, 0);
cvs_output("/", 0);
cvs_output(finfo->file, 0);
}
cvs_output("\nhead: ", 0);
if (rcsfile->head != NULL)
{
cvs_output(" ", 1);
cvs_output(rcsfile->head, 0);
}
cvs_output("\nbranch: ", 0);
if (rcsfile->branch != NULL)
{
cvs_output(" ", 1);
cvs_output(rcsfile->branch, 0);
}
cvs_output("\nlocks: ", 0);
if (rcsfile->strict_locks)
cvs_output(" strict", 0);
walklist(RCS<ref>verbs</ref>, printlocks<ref>, NULL);
cvs_output("\nascess list: ", 0);
if (rcsfile->access != NULL)
{
const char *cp;

while (cp = rcsfile->access)
{
    cp = rcsfile->access;

    while (cp != '\0')
    {
        cp = rcsfile->access;

        while (isspace(*cp) & &
                *cp != '\0')
            ++cp;
        while (isspace(*cp) & &
                *cp != '\0')
            ++cp;
    }
}
}
for (log_data->notags)
{
    List *syms;

cvs_output("\nasymbolic names: ", 0);
syms = RCS<ref>symbols</ref>(rcsfile);
walklist(syms, log_symbol, NULL);
}
cvs_output("\nkeyword substitution: ", 0);
if (rcsfile->expand == NULL)
cvs_output("Expand ", 2);
else
cvs_output(rcsfile->expand, 0);
cvs_output("\ntotal revisions: ", 0);
printf(buf, "%d", walklist(rcsfile->versions, log_count, NULL));
cvs_output(buf, 0);
if (log_data->header & & ! log_data->long_header)
{
cvs_output("\ntselected revisions: ", 0);
log_data_and_cvs_log_data = log_data;
log_data_and_cvs_reclist = reclist;
log_data_and_cvs_rca = rcsfile;
/* If any single dates were specified, we need to identify the revisions they select. Each one selects the single revision which is otherwise selected, of that date or earlier. The singledate routine will fill in the start date for each specific revision. */
if (log_data->singledate != NULL) {
walklist (rcsfile->versions, loglist->singledate, (void *) &log_data->sdata);
}

sprintf (buf, "%d", walklist (rcsfile->versions, logcount->print, (void *) &log_data->sdata));
csvoutput (buf, 0);
}
csvoutput ("\n", 1);

if (! log_data->header || ! log_data->long_header) {
csvoutput ("description:\n", 0);
if (rcsfile->desc != NULL) csvoutput (rcsfile->desc, 0);
}

if (! log_data->header && ! log_data->long_header && rcsfile->head != NULL) {
p = findnode (rcsfile->versions, rcsfile->head); if (p == NULL) {
    error (1, 0, "can not find head revision in \%s", rcsfile->fullname);
    while (p != NULL) {
        RCSVers *vers;
        vers = (RCSVers *) p->data;
        log_version (log_data, revlist, rcsfile, vers, 1);
        if (vers->next == NULL) p = NULL;
        else {
            p = findnode (rcsfile->versions, vers->next);
            if (p == NULL) {
                error (1, 0, "can not find next revision \%s in \%s", vers->next, rcsfile->fullname);
            }
        }
        log_tree (log_data, revlist, rcsfile, rcsfile->head);
    }
}
csvoutput("\n",
0);

/* Free up the new revlist and restore the old one. */
free_relist (revlist);

/* If singledatelist is not NULL, free up the start dates we added to it. */
if (log_data->singledatelist != NULL) {
    struct datelist *d;
    for (d = log_data->singledatelist; d != NULL; d = d->next) {
        if (d->start != NULL) free (d->start);
        d->start = NULL;
    }
}

return 0;

/* Fix up a revision list in order to compare it against versions.
   Expand any symbolic revisions. */
static struct revlist *
logexpand_relist (rcs, revlist, default_branch); RCSNode *rcs;

struct option_relist *revlist;
int default_branch;
{
    struct option_relist *r;
    struct revlist *rel, *pr;
    ret = NULL;
p = &ret;
    for (r = revlist; r != NULL; r = r->next)
```c
struct revlist *nr;

nr = (struct revlist *) xmalloc(sizeof *nr);

if (r->first == NULL && r->last == NULL)
{
    /* If both first and last are NULL, it means that we want
     * just the head of the default branch, which is RCShead. */
    nr->first = RCShead (rcs);
    nr->last = xstrdup (nr->first);
    nr->fields = numdots (nr->first) + 1;
}
else if (r->branchhead)
{
    char *branch;

    /* Print just the head of the branch. */
    if (isdigit (r->first[0]))
        nr->first = RCS_getbranch (rcs, r->first, 1);
    else
    {
        branch = RCS_whatbranch (rcs, r->first);
        if (branch == NULL)
        {
            error (0, 0, "warning: " "is not a branch in " "
                   r->first, rcs->path);
            free (nr);
            continue;
        }
        nr->first = RCS_getbranch (rcs, branch, 1);
        free (branch);
    }
    if (nr->first == NULL)
    {
        error (0, 0, "warning: no revision " " in " "
               r->first, rcs->path);
        free (nr);
        continue;
    }
    nr->last = xstrdup (nr->first);
    nr->fields = numdots (nr->first) + 1;
}
else
{
    if (r->first == NULL || isdigit (r->first[0]))
        nr->first = xstrdup (r->first);
    else
    {
        if (RCS_nodeisbranch (rcs, r->first))
            nr->first = RCS_getbranch (rcs, r->first);
        else
        {
            nr->first = RCS_gettag (rcs, r->first, 1, (int *) NULL);
            if (nr->first == NULL)
            {
                error (0, 0, "warning: no revision " " in " "
                       r->first, rcs->path);
                free (nr);
                continue;
            }
        }
    }
    if (r->last == r->first)
        nr->last = xstrdup (nr->first);
    else if (r->last == NULL || isdigit (r->last[0]))
        nr->last = xstrdup (r->last);
    else
    {
        if (RCS_nodeisbranch (rcs, r->last))
            nr->last = RCS_gettag (rcs, r->last, 1, (int *) NULL);
        else
        {
            nr->last = RCS_getbranch (rcs, r->last, 1);
            if (nr->last == NULL)
            {
                error (0, 0, "warning: no revision " " in " "
                       r->last, rcs->path);
                if (nr->first != NULL)
                {
                    free (nr->first);
                    free (nr);
                    continue;
                }
            }
        }
    }

    /* Process the revision numbers the same way that rlog
does. This code is a bit cryptic for my tastes, but
keeping the same implementation as rlog ensures a
certain degree of compatibility. */
    if (r->first == NULL)
    {
        nr->fields = numdots (nr->last) + 1;
    }
```
if (nr->fields < 2)
    nr->first = xstrdup (*cp);
else
    {
        char *cp;
        nr->first = xstrdup (nr->last);
        cp = strchr (nr->first, ',');
        strcpy (cp, *cp);
    }
else if (nr->last == NULL)
    {
        nr->fields = numdots (nr->first) + 1;
        nr->last = xstrdup (nr->first);
        if (nr->fields < 2)
            nr->last[0] = '\0';
    }
    else
    {
        nr->fields = numdots (nr->first) + 1;
        if (nr->fields != numdots (nr->last) + 1)
            
            && version_compare (nr->first, nr->last, 
            nr->fields - 1) != 0))
            {
                error [0, 0,
                "invalid branch or revision pair \%s:\%s in \'\%s'",
                r->first, r->last, rcs->path);
                free (nr->first);
                free (nr->last);
                free (nr);
                continue;
            }
    }
if (version_compare (nr->first, nr->last, nr->fields) > 0)
    {
        char *tmp;
        tmp = nr->first;
        nr->first = nr->last;
        nr->last = tmp;
    }
}

nr->next = NULL;
*pr = nr;
pr = &nr->next;

/* If the default branch was requested, add a revlist entry for it. This is how rlog handles this option. */
if (defaultbranch
    && (rcs->head != NULL || rcs->branch != NULL))

struct revlist *nr;

    nr = (struct revlist *) xmalloc (sizeof *nr);
    if (rcs->branch != NULL)
        nr->first = xstrdup (rcs->branch);
    else
    {
        char *cp;
        nr->first = xstrdup (rcs->head);
        cp = strchr (nr->first, ',');
        *cp = '\0';
        nr->last = xstrdup (nr->first);
        nr->fields = numdots (nr->first) + 1;
        nr->next = NULL;
        *pr = nr;
    }

return ret;

/* Free a revlist created by log\_expand\_revlist. */
static void
log\_free\_revlist (revlist)
struct revlist *revlist;

struct revlist *r;

while (r != NULL)
{
    struct revlist *next;
    if (r->first != NULL)
        free (r->first);
    if (r->last != NULL)
        free (r->last);
    next = r->next;
    free (r);
    r = next;
}

/* Return nonzero if a revision should be printed, based on the
   options provided. */
static int
log_version_requested (log_data, revlist, rcs, vnode)
struct log_data *log_data,
struct revlist *revlist,
RCSNode *rcs,
VNODE *vnode;
{
    /* Handle the list of states from the -s option. */
    if (log_data->statelist != NULL
        && findnode (log_data->statelist, vnode->state) == NULL)
        return 0;

    /* Handle the list of authors from the -w option. */
    if (log_data->authorlist != NULL
        && findnode (log_data->authorlist, vnode->author) == NULL)
        return 0;

    /* rlog considers all the -d options together when it decides
     * whether to print a revision, so we must be compatible. */
    if (log_data->singlesetlist != NULL || log_data->singlesetlist != NULL)
    {
        struct datelist *d;
        for (d = log_data->singlesetlist; d != NULL; d = d->next)
        {
            int cmp;
            cmp = RCSdatecmp (vnode->date, d->start);
            if (cmp > 0 || (cmp == 0 && d->inclusive))
            {
                cmp = RCSdatecmp (vnode->date, d->end);
                if (cmp < 0 || (cmp == 0 && d->inclusive))
                    break;
            }
        }
    }

    /* Look through the list of specific dates. We want to
    select the revision with the exact date found in the
    start field. The commit code ensures that it is
    impossible to check in multiple revisions of a single
    file in a single second, so checking the date this way
    should never select more than one revision. */
    for (d = log_data->singlesetlist; d != NULL; d = d->next)
    {
        if (d->start != NULL
            && RCSdatecmp (vnode->date, d->start) == 0)
            break;
    }

    if (d == NULL)
        return 0;
}

/* If the -r or -b options were used, REVLIST will be non NULL,
   and we print the union of the specified revisions. */
if (revlist != NULL)
{
    char *v;
    int vfields;
    struct revlist *r;
    /* This code is taken from rlog. */
    v = vnode->version;
    vfields = numdots(v) + 1;
    for (r = revlist; r != NULL; r = r->next)
    {
        if (vfields == r->fields + (r->fields & 1)
           && version_compare(v, r->first, r->fields) >= 0
           && version_compare(v, r->last, r->fields) <= 0)
        {
            return 1;
        }
    }
    /* We get here, then the -b and/or the -r option was used,
     * but did not match this revision, so we reject it. */
    return 0;
}
/* By default, we print all revisions. */
return 1;
}

/* Output a single symbol. This is called via walklist. */
*/
*/ARGSUSED*/
static int
log_symbol(p, closure)
    Node *p;
    void *closure;
{
    cvsoutput("\n\t", 2);
    cvsoutput(p->key, 0);
    cvsoutput("\t", 2);
    cvsoutput(p->data, 0);
    return 0;
}
/* Count the number of entries on a list. This is called via walklist. */
*/
*/ARGSUSED*/
static int
log_count(p, closure)

Node *p;
void *closure;
{
    return 1;
}
/* Sort out a single date specification by narrowing down the date
 * until we find the specific selected revision. */
*/
static int
log_findsingledate(p, closure)
    Node *p;
    void *closure;
{
    struct logdataandrcs *data = (struct logdataandrcs *) closure;
    Node *pv;
    RCSVers *vnode;
    struct datelist *holdingsingle, *holddate;
    int requested;
    pv = findnode(data->rcs->versions, p->key);
    if (pv == NULL)
        error(1, 0, "missing version '%s' in RCS file "%s",
               p->key, data->rcs->path);
    vnode = (RCSVers *) pv->data;
    /* We are only interested if this revision passes any other tests.
     * Temporarily clear logdata->singledatelist to avoid confusing
     * logversion requested. We also clear logdata->datelist,
     * because rlog considers all the -d options together. We don't
     * want to reject a revision because it does not match a date pair
     * if we are going to select it on the basis of the singledate. */
    holdingsingle = data->logdata->singledatelist;
    data->logdata->singledatelist = NULL;
    holddate = data->logdata->datelist;
    data->logdata->datelist = NULL;
    requested = logversion_requested(data->logdata, data->revlist,
                                      data->rcs, vnode);
/* For each single date, if this revision is before the
specified date, but is closer than the previously selected
revision, select it instead. */

for (d = data->logdata->singledatelist; d != NULL; d = d->next)
{
  if (RCSdatecmp(vnode->date, d->end) <= 0
     && (d->start == NULL
         || RCSdatecmp(vnode->date, d->start) > 0))
    { if (d->start != NULL)
       free(d->start);
       d->start = xstrdup(vnode->date);
    }
  }

return 0;
}

/* Count the number of revisions we are going to print. */

static int
log_count_print(p, closure)

Node *p;

void *closure;

struct logdata *data = (struct logdata **) closure;
Node *pv;

pv = findnode(data->rcs->versions, p->key);
if (pv == NULL)
  exit(1, 0, "missing version '%s' in RCS file '%s'",
       p->key, data->rcs->path);

if (logversionrequested(data->logdata, data->reclist, data->rcs,
                       (RCSVers *) p->data))
  return 1;
else
  return 0;
}

/* Print the list of changes, not including the trunk, in reverse
 * order for each branch. */

static void
log_tree(logdata, reclist, rcs, ver)

struct logdata *logdata;
struct reclist *reclist;
RCSNode *rcs;

const char *ver;

Node *p;
RCSVers *vnode;

p = findnode(rcs->versions, ver);
if (p == NULL)
  exit(1, 0, "missing version '%s' in RCS file '%s'",
       ver, rcs->path);

vnode = (RCSVers *) p->data;
if (vnode->next != NULL)
  log_tree(logdata, reclist, rcs, vnode->next);
if (vnode->branches != NULL)
  { Node *head, *branch;

    /* We need to do the branches in reverse order. This breaks
     * the list abstraction, but so does most of the branch
     * manipulation in rcs.c. */
    head = vnode->branches->list;
    for (branch = head->prev; branch != head; branch = branch->prev)
      { log_abranch(logdata, reclist, rcs, branch->key);
        log_tree(logdata, reclist, rcs, branch->key);
      }

  }

/* Log the changes for a branch, in reverse order. */

static void
log_abranch(logdata, reclist, rcs, ver)
struct logdata *logdata;
struct revlist *revlist;
RCSNode *rcs;
const char *version;

output = logbranch (logdata, revlist, rcs, vnode, 0);

getlocks (logdata, revlist, rcs, vnode, 0);

*/
static void
log_output (logdata, revlist, rcs, ver, trunk)
struct logdata *logdata;
struct revlist *revlist;
RCSNode *rcs;
RCSVers *ver;
int trunk;

if (! logversion_requested (logdata, revlist, rcs, ver))
    return;

cvsoutput ("--------------------------\revision ", 0);
cvsoutput (ver->version, 0);

p = findnode (RCSgetlocks (rcs), ver->version);

if (p != NULL)
{
    cvsoutput ("\locked by: ", 0);
    cvsoutput (p->data, 0);
    cvsoutput ("\", 1);
}

cvsoutput ("\date: ", 0);
(void) sscanf (buf, "%04d/%02d/%02d %02d:%02d:%02d",
year, mon, mday, hour, min, sec);

cvsoutput (buf, 0);

cvsoutput ("\author: ", 0);

cvsoutput (ver->author, 0);

cvsoutput ("\state: ", 0);

cvsoutput (ver->state, 0);

cvsoutput ("\", 1);

if (! trunk)
{
    padd = findnode (ver->other, "add");
    pdel = findnode (ver->other, "delete");
}
else if (ver->next == NULL)
{
    padd = NULL;
    pdel = NULL;
}
else
{
    Node *nextp;
    RCSVers *nextver;

    nextp = findnode (rcs->versions, ver->next);
    if (nextp == NULL)
        error (1, 0, "missing version \%s in \%s",
ver->next, rcs->path);

    nextver = (RCSVers *) nextp->data;
    pdel = findnode (nextver->other, "delete");
    padd = findnode (nextver->other, "add");
}
if (padd != NULL)
{
cvs_output("* lines: *", 0);
cvs_output((padd->data, 0);
cvs_output("-*", 2);
cvs_output((pdel->data, 0);
}
if (ver->branches != NULL)
{
cvs_output("\nbranches: ", 0);
walklist (ver->branches, log_branch, (void *) NULL);
}
cvs_output("\n", 1);
p = findnode (ver->other, "log");
/* The p->date == NULL case is the normal one for an empty log
message (rcs-14 in sanity.sh). I don't think the case where
p->data is "" can happen (getrcskey in rcs.c checks for an
empty string and set the value to NULL in that case). My guess
would be the p == NULL case would mean an RCS file which was
missing the "log" keyword (which is illegal according to
rcsfile.5). */
if ((p == NULL || p->data == NULL || p->data[0] == '\0'))
cvs_output(" *** emptylog message *** ", 0);
else
{
/* FIXME: Technically, the log message could contain a null
byte. */
cvs_output (p->data, 0);
if (p->data[strlen (p->data) - 1] == '\n')
cvs_output("\n", 1);
}
/* Output a branch version. This is called via walklist. */
/*ARGSUSED*/
static int log_branch (p, closure)
Node *p;
void *closure;
{
cvs_output("", 2);
if ((numdots (p->key) & 1) == 0)
cvs_output (p->key, 0);
else
{
char *f, *cp;
f = xstrdup (p->key);
for (f = strchr (f, '.');
    f != NULL;
    f = strchr (f, '.'))
    free (f);
}
cvs_output(";", 1);
return 0;
}
/* Print a warm fuzzy message */
/*ARGSUSED*/
static Dtype log_dirproc (callerdat, dir, repository, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repository;
    char *update_dir;
    List *entries;
{
if (!isdir (dir))
    return (R_SKIP_ALL);
if (!quiet)
    error (0, 0, "Logging %s", update_dir);
    return (R_PROCESS);
    }
/* Compare versions. This is taken from RCS compartial. */
/*ARGSUSED*/
static int version_compare (v1, v2, len)
    const char *v1;
    const char *v2;
    int len;
    { while (1)
{ 
    int d1, d2, r;
    if (*v1 == '\0')
        return 1;
    if (*v2 == '\0')
        return -1;
    while (*v1 == '0')
        ++v1;
    for (d1 = 0; isdigit (v1[d1]); ++d1) :
    while (*v2 == '0')
        ++v2;
    for (d2 = 0; isdigit (v2[d2]); ++d2) :
        if (d1 != d2)
            return d1 < d2 ? -1 : 1;
    r = memcmp (v1, v2, d1);
    if (r != 0)
        return r;
    --len;
    if (len == 0)
        return 0;
    v1 += d1;
    v2 += d1;
    if (*v1 == '.')
        ++v1;
    if (*v2 == '.')
        ++v2;
}
A.34 login.c

/*! 
 * Copyright (c) 1995, Cyclic Software, Bloomington, IN, USA
 * 
 * You may distribute under the terms of the GNU General Public License as 
 * specified in the README file that comes with CVS.
 * 
 * Allow user to log in for an authenticating server.
 */

#include "cvs.h"
#include "getline.h"

#ifdef AUTH_CLIENT_SUPPORT /* This covers the rest of the file. */

/* There seems to be very little agreement on which system header getpass is declared in. With a lot of fancy autoconfiguration, 
we could perhaps detect this, but for now we'll just rely on _CRAY, since Cray is perhaps the only system on which our own 
declaration won't work (some Crays declare the %CVS% thing as 
varadic, believe it or not). On Cray, getpass will be declared */

#ifndef _CRAY
extern char *getpass();
#endif

#ifndef CVS_PASSWORD_FILE
#define CVS_PASSWORD_FILE " .cvspass"
#endif

static char *construct_cvspass_filename PROTO ((void));

/* The return value will need to be freed. */
static char * construct_cvspass_filename ()
{
    char *homedir, *passfile;

    /* Environment should override file. */
    if ((passfile = getenv("CVS_PASSFILE")) != NULL) 
        return xmalloc(strlen(passfile) + 3);

    /* Construct absolute pathname to user's password file. */
    /* todo: does this work under OS/2 ? */
    homedir = get_home_dir();
    if (!homedir)
    {
        error (1, errno, "could not find out home directory");
        return (char *) NULL;
    }

    passfile = (char *) xmalloc(strlen(homedir) + strlen(CVS_PASSWORD_FILE) + 3);
    strcpy(passfile, homedir);

    ifndef NO_SLASH_AFTER_HOME
    /* NO_SLASH_AFTER_HOME is defined for VMS, where foo:/bar/.cvspass is not 
    a legal filename but foo:/bar/.cvspass is. A more clean solution would 
    be something more along the lines of a "join a directory to a filename" 
    kind of thing.... */
    strcat(passfile, "/");
    endif

    strcat(passfile, CVS_PASSWORD_FILE);

    /* Safety first and last, Scouts. */
    if (!file (passfile))
    /* chmod() is too polite. */
    chmod (passfile, 0600);

    return passfile;
}

static const char *const login_usage[] =
{
    "Usage: %s %s
    "
    "(Specify the --help global option for a list of other help options)\n",
    NULL,

    /* Prompt for a password, and store it in the file "CVS/.cvspass".
     */
    /* Because the user might be accessing multiple repositories, with 
    different passwords for each one, the format of "/.cvspass is:
    */
    "userhost:/path Aclearertext\password
    
    "
};
Of course, the “user” might be left off -- it’s just based on the
value of CVSroot.

The “A” before “cleartext_password” is a literal capital A. It’s a
version number indicating which form of scrambling we’re doing on
the password -- someday we might provide something more secure than
the trivial encoding we do now, and when that day comes, it would
be nice to remain backward-compatible.

Like .netrc, the file’s permissions are the only thing preventing
it from being read by others. Unlike .netrc, we will not be
fascist about it, at most issuing a warning, and never refusing to
work.

int
login (argc, argv)
int argc;
char **argv;
{
    char *passfile;
    FILE *fp;
    char *typed_password, *found_password;
    char *linebuf = (char *) NULL;
    size_t linebuf_len;
    int root_len, already_entered = 0;
    int line_length;
    if (argc < 0)
        usage (login_usage);
    if (CVSroot_method != pserver_method)
    {
        error (0, 0, "can only use pserver method with 'login' command");
        error (1, 0, "CVSROOT: %s", CVSroot_original);
    }
    if (! CVSroot_username)
    {
        error (0, 0, "CVSROOT "%s" is not fully-qualified.",
                CVSroot_original);
        error (1, 0, "Please make sure to specify "user@host"!");
    }
    printf ("(Logging in to %s@%s)
\n", CVSroot_username, CVSroot_hostname);
    fflush (stdout);
    passfile = construct_cvspass_filename ();
    typed_password = getpass ("CVS password: ");
    typed_password = scramble (typed_password);
    root_len = strlen (CVSroot_original);
    if (fp = CVS_FOPEN (passfile, "r"))
    {
        while (getline (&linebuf, &linebuf_len, fp) > 0)
        {
            if (strncmp (CVSroot_original, linebuf, root_len) == 0)
            {
                already_entered = 1;
                break;
            }
if (fclose (fp) < 0) {
    error (0, errno, "cannot close tmp", passfile);
} else if (fexist (errno)) {
    error (0, errno, "cannot open tmp", passfile);
}

if (already entered) {
    /* This user/host has a password in the file already. */
}

strtok (linebuf, " ");
found_password = strtok (NULL, "\n");
if (strcmp (found_password, typed_password)) {
    /* type password and found password don’t match, so we’ll
     * have to update passfile. We replace the old password
     * with the new one by writing a tmp file whose contents are
     * exactly the same as passfile except that this one entry
     * gets typed password instead of found password. Then we
     * rename the tmp file on top of passfile.

     */
    char +tmp_name;
    FILE *tmpfp;

    tmp_name = cvs_temp_name ();
    if ((tmpfp = CVS_FOPEN (tmp_name, "w")) == NULL) {
        error (1, errno, "unable to open temp file \n", tmp_name);
        return 1;
    }
    chmod (tmp_name, 0600);

    fp = CVS_FOPEN (passfile, "r");
    if (fp == NULL) {
        error (1, errno, "unable to open tmp file \n", passfile);
        if (linebuf)
            free (linebuf);
        return 1;
    }
    /* I’m not paranoid, they really ARE out to get me: */
    chmod (passfile, 0600);

    while (getline (&linebuf, &linebuf_len, fp)) >= 0)
    {
        if (strncmp (CVSroot_original, linebuf, root_len)) {
            if (fprintf (tmpfp, "%s\n", linebuf) == EOF) {
                error (0, errno, "cannot write \n", tmp_name);
            }
        } else {
            if (fprintf (tmpfp, "%s %s\n", CVSroot_original, typed_password) == EOF) {
                error (0, errno, "cannot write \n", tmp_name);
            }
        }
        if (line_length < 0 & & feof (fp))
            error (0, errno, "cannot read \n", passfile);
       -free (linebuf);
        if (fclose (tmpfp) < 0)
            error (0, errno, "cannot close tmp", tmp_name);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close \n", passfile);
    } /* FIXME: rename file would make more sense (e.g. almost
    * always faster). */
    copyfile (tmp_name, passfile);
    unlink_file (tmp_name);
    chmod (passfile, 0600);
    free (tmp_name);
}
}

else {
    if (linebuf)
        free (linebuf);
    if (fp = CVS_FOPEN (passfile, "r")) == NULL) {
        error (1, errno, "could not open \n", passfile);
        free (passfile);
        return 1;
    }
    if (fprintf (fp, "%s %s\n", CVSroot_original, typed_password) == EOF) {
        error (0, errno, "cannot write \n", passfile);
    }
    if (fclose (fp) < 0)
/* UTter, total, raving paranoia, I know. */
chmod (passfile, 0600);
memset (type, password, 0);
free (type, password);
free (passfile);
free (cvs_password);
cvs_password = NULL;
return 0;
}

/* Returns the scrambled password. The server must descrumble before hashing and comparing. */
char *
get_cvs_password ()
{
    int found_it = 0;
    int root_len;
    char *password;
    char linebuf = (char *) NULL;
    size_t linebuf_len;
    FILE *fp;
    char *passfile;
    int line_length;

    /* If someone (i.e., login()) is calling connect()/server() out of context, then assume they have supplied the correct, scrambled password. */
    if (cvs_password)
        return cvs_password;

    if (getenv ("CVS_PASSWORD") != NULL)
    {
        /* In previous versions of CVS one could specify a password in CVS_PASSWORD. This is a bad idea, because in BSD variants of unix anyone can see the environment variable with 'ps'. But for users who were using that feature we want to at least let them know what is going on. After printing this warning, we should fall through to the regular error where we tell them to run "cvs login" (unless they already ran it, of course). */
        if (errno)
            "CVS_PASSWORD is no longer supported; ignored");
    }

    /* Else get it from the file. First make sure that the CVSROOT variable has the appropriate fields filled in. */
    if (CVSroot_method != pserver_method)
    {
        error (1, 0, "can only call GET_CVS_PASSWORD with pserver method");
        error (1, 0, "CVSROOT: \%s"");
    }
    else if (! CVSroot_username)
    {
        error (1, 0, "CVSROOT \%s" is not fully-qualified.");
        error (1, 0, "Please make sure to specify \%s\$USERNAME\%s");
    }
    passfile = construct_cvs_passfile ();
    fp = CVS_POPEN (passfile, "r");
    if (fp == NULL)
    {
        error (0, errno, "could not open \%s", passfile);
        free (passfile);
        error (1, 0, "use \"cvs login\" to log in first");
    }
    root_len = strlen (CVSroot_original);

    /* Check each line to see if we have this entry already. */
    while ((line_length = getline (&linebuf, &linebuf_len, fp)) >= 0)
    {
        /* This is it! So break out and deal with linebuf. */
        if (strncmp (CVSroot_original, linebuf, root_len) == 0)
            /* This is it! So break out and deal with linebuf. */
            found_it = 1;
            break;
    }
    if ( line_length < 0 && feof (fp))
        error (0, errno, "cannot read \%s", passfile);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close \%s", passfile);
    if (found_it)
Source code

```c
{ /* linebuf now contains the line with the password. */
    char *tmp;
    strtok (linebuf, " ");
    password = strtok (NULL, "\n");
    /* Give it permanent storage. */
    tmp = xstrdup (password);
    memset (password, 0, strlen (password));
    free (linebuf);
    return tmp;
} else
    {
        if (linebuf)
            free (linebuf);
        error (0, 0, "cannot find password");
        error (1, 0, "use "cvs login" to log in first");
    } /* NOTREACHED */
return NULL;
}

static const char *const logout_usage[] = {
    "Usage: %s %s
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

/* Remove any entry for the CVStree repository found in "CVS/.cvspass". */
int logout (argc, argv)
int argc;
char **argv;
{
    char *passfile;
    FILE *fp;
    char *tmp_name;
    FILE *tmp_fp;
    char *linebuf = (char *) NULL,
    size_t linebuf_len;
    int root_len, found = 0;
    int line_length;
    if (argc < 0)
        usage (logout_usage);
    if (CVSroot_method != pserver_method)
        {
            error (0, 0, "can only use pserver method with "logout" command");
            error (1, 0, "CVSROOT: %s", CVSroot_original);
        }
    if (! CVSroot_username)
        {
            error (0, 0, "CVSROOT \"%s\" is not fully-qualified.",
                CVSroot_original);
            error (1, 0, "Please make sure to specify \"user@host\"!");
        }

    /* Hmm. Do we want a variant of this command which deletes all
     * the entries from the current .cvspass? Might be easier to
     * remember than "rm .cvspass" but then again if people are
     * mucking with HOME (common in Win95 as the system doesn't set
     * it), then this variant of "cvs logout" might give a false sense
     * of security, in that it wouldn't delete entries from any
     * .cvspass files but the current one. */
    printf ("(Logging out of %s@%s)\n", CVSroot_username, CVSroot_hostname);
    fflush (stdout);
    /* IF we have a password for this "[user]host/path" already
     * THEN
     *   drop the entry
     * ELSE
     *   do nothing
     */
    passfile = construct_cvspass_filename ();
    tmp_name = cvs_temp_name ();
    if ((tmp_fp = CVS_FOPEN (tmp_name, "w")) == NULL)
        {
            error (1, errno, "unable to open temp file %s", tmp_name);
            return 1;
        }
    chmod (tmp_name, 0600);
    root_len = strlen (CVSroot_original);
```
fp = CVS_FOPEN (passfile, "r");
if (fp == NULL)
    error (1, errno, "Error opening %s", passfile);

/* Check each line to see if we have this entry. */
/* Copy only those lines that do not match this entry */
while ((line_length = getline (&linebuf, &linebuf_len, fp)) >= 0)
{  
    if (strncmp (CVSroot_original, linebuf, root_len))
    {
        if (fprintf (tmp_fp, "%s", linebuf) == EOF)
            error (0, errno, "cannot write %s", tmp_name);
    
    } else
        found = 1;
}
if (line_length < 0 && !feof (fp))
    error (0, errno, "cannot read %s", passfile);
if (linebuf)
    free (linebuf);
if (fclose (fp) < 0)
    error (0, errno, "cannot close %s", passfile);
if (fclose (tmp_fp) < 0)
    error (0, errno, "cannot close %s", tmp_name);

if (! found)
{
    printf ("Entry not found for %s\n", CVSroot_original);
    unlink_file (tmp_name);
}
else
{
    /* FIXME: rename_file would make more sense (e.g. almost
        always faster). */
    copy_file (tmp_name, passfile);
    unlink_file (tmp_name);
    chmod (passfile, 0600);
    return 0;
}

#endif /* AUTH_CLIENT_SUPPORT from beginning of file. */
A.35 logmsg.c

/ * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"
#include "getline.h"

static int find_type PROTO((Node * p, void * closure));
static int fmtproc PROTO((Node * p, void * closure));
static int logfile_write PROTO((char * repositary, char * filter, 
    char * message, FILE * logfp, List * changes));
static int rcsinfo PROTO((char * repositary, char * template));
static int titleproc PROTO((Node * p, void * closure));
static int update_logfile PROTO((char * repositary, char * filter));
static void setupimplefile PROTO((FILE * xfp, char * xprefix, List * changes));
static int editinfo PROTO((char * repositary, char * template));
static int verifymsg PROTO((char * repositary, char * script));

static FILE *fp;
static char *strlist; 
static char *strlist_format; /* The format for strlist's contents. */
static char *editinfo; 
static char *verifymsg; 
static Ctype type;

30 / *
 * Puts a standard header on the output which is either being prepared for an
 * editor session, or being sent to a logfile program. The modified, added,
 * and removed files are included (if any) and formatted to look pretty. */
static char *prefix;
static int col;
static char *tag;
static void setupimplefile (xfp, xprefix, changes)
FILE *xfp,
char *xprefix;
{
/* set up statics */
fp = xfp;
prefix = xprefix;

type = T_MODIFIED;
if (walklist (changes, find_type, NULL) != 0) {
    (void) fprintf (fp, "%LaModified Files:\n", prefix);
    col = 0;
    (void) walklist (changes, fmtproc, NULL);
    (void) fprintf (fp, "%s\n", prefix);
    if (tag != NULL) {
        free (tag);
        tag = NULL;
    }
}

60 type = T_ADDED;
if (walklist (changes, find_type, NULL) != 0) {
    (void) fprintf (fp, "%LaAdded Files:\n", prefix);
    col = 0;
    (void) walklist (changes, fmtproc, NULL);
    (void) fprintf (fp, "%s\n", prefix);
    if (tag != NULL) {
        free (tag);
        tag = NULL;
    }
}

70 type = T_REMOVED;
if (walklist (changes, find_type, NULL) != 0) {
    (void) fprintf (fp, "%LaRemoved Files:\n", prefix);
    col = 0;
    (void) walklist (changes, fmtproc, NULL);
    (void) fprintf (fp, "%s\n", prefix);
    if (tag != NULL) {
        free (tag);
        tag = NULL;
    }
}

/ *
static int find_type(Node *p, void *closure)
{
    struct logfile_info *li = (struct logfile_info *) p->data;
    if (li->type == type)
        return (1);
    else
        return (0);
}

static int fmt_proc(Node *p, void *closure)
{
    struct logfile_info *li = (struct logfile_info *) p->data;
    if (li->type == type)
    {
        if (li->tag == NULL)
            return (0);
        else
            fprintf(fp, "Tag: %s\n", li->tag);
        free(li->tag);
        li->tag = xstrdup(fp, "\t");
        col = strlenfp, "\t");
        if (col == 0)
            col = 8;
        else if (col > 8 && (col + strlen(p->key)) > 70)
            col = 8;
        fprintf(fp, "%s\n", p->key);
        col += strlen(p->key) + 1;
    }
    return (0);
}

void do_editor(char *dir, char *messagep, char *repository, List *changes)
{
    static int reuse_log_message = 0;
}
char *line;
int line_length;
size_t line_chars_allocated;
char *fname;
struct stat *post_stbuf, post_stbuf;
int retcode = 0;
char *p;

if (noexec || reuse_log_message)
    return;

/* Abort creation of temp file if no editor is defined */
if (strcmp (Editor, "") == 0 && editinfo == "")
    error (1, 0, "no editor defined, must use -e or -a");

/* Create a temporary file */
fname = cvsedit::name ();
again:
    if (!fp = CVS_OPEN (fname, "w")) == NULL)
        error (1, 0, "cannot create temporary file "fname",

if (*messagep)
    {
        (void) fprintf (fp, "%s", *messagep);
        if (! (*messagep)[0] == \0 ||
            (*messagep)[strlen (*messagep) - 1] != \n')
            (void) fprintf (fp, "%\n");
    }
else
    (void) fprintf (fp, "%\n");

if (repository != NULL)
    /* task templates on if necessary */
    (void) Parse_Info (CVSROOTADM_RCSINFO, repository, rcsinfoc, 1);
else
    {
        FILE *tfp;
        char buf [1024];
        char *p;

        size_t n;
        size_t nwrite;

        /* Why "b"? */
        tfp = CVS_OPEN (CVSADM_TEMPLATE, "rb");
        if (tfp == NULL)
            {
                if (existence_error (errno))
                    error (1, errno, "cannot read %s", CVSADM_TEMPLATE);
            }
        else
            {
                while (!feof (tfp))
                    {
                        n = fread (buf, 1, sizeof buf, tfp);
                        nwrite = n;
                        p = buf;
                        while (nwrite > 0)
                            {
                                n = fwrite (p, 1, nwrite, fp);
                                nwrite -= n;
                                p += n;
                            }
                if (error (tfp))
                    error (1, errno, "cannot read %s", CVSADM_TEMPLATE);
            }
        if (fclose (tfp) < 0)
            error (0, errno, "cannot close %s", CVSADM_TEMPLATE);
    }

    (void) fprintf (fp,
    "%s--------------------------------------------------------------------------\n", CVSEDITPREFIX);

    (void) fprintf (fp,
    "\033[1m\033[31m| Editor Log | Lines beginning with '\"\l\"\r\"' are removed automatically|\n", CVSEDITPREFIX, CVSEDITPREFIXLEN, CVSEDITPREFIX, CVSEDITPREFIX);
    if (dir != NULL && -dir)
        (void) fprintf (fp, "\033[31m| Committing in %s|\n", CVSEDITPREFIX, CVSEDITPREFIX);

    if (changes != NULL)
        setup_impfile (fp, CVSEDITPREFIX, changes);

    (void) fprintf (fp,
    "%s--------------------------------------------------------------------------\n", CVSEDITPREFIX);

    /* finish off the temp file */
    if (fclose (fp) == EOF)
270   error (1, errno, "Es", fname);
   if ( CVS_STAT (fname, &pre_stbuf_atime) == -1)
      pre_stbuf_atime = 0;
   if (editinfoeditor)
      free (editinfoeditor);
   editinfoeditor = (char *) NULL;
   sifdef CLIENT2SUPPORT
   if (client_active)
      /* nothing, leave editinfoeditor NULL */
   else
   sendif
   if (repository != NULL)
      (void) ParseInfo (CVSROOTADM>EditINFO, repository, editinfo, 0);
   /* run the editor */
   run_setup (editinfoeditor, Editor, runarg (fname),
              if (retcode = runxex (RUNTTY, RUNTTY, RUNTTY, RUN_NORMAL | RUN_SIGHNORE)) == 0)
      error (editinfoeditor ? 1 : 0, retcode == -1 ? errno : 0,
      editinfoeditor ? "logfile verification failed" : "warning: editor session failed");
   /* put the entire message back into the *messagep variable */
   fp = openfile (fname, "r");
   if (*messagep)
      free (*messagep);
   if ( (CVS_STAT (fname, &post_stbuf) != 0)
      error (1, errno, "cannot find size of temp file Es", fname);
   else
      /* On NT, we might read less than stsize, so we don't read more. So this works, */
      /* messagep = (char *) xmalloc (post_stbuf.stsize + 1);
      *messagep[0] = '\0';
   line = NULL;
   line_chars_allocated = 0;
   if (*messagep)
   {p = *messagep;
    while (1)
    { line_length = getline (&line, &line_chars_allocated, fp);
       if (line_length == -1)
       { if (error (fp))
            error (0, errno, "warning: cannot read Es", fname);
            break;
       }
       if (strncmp (line, CVSEDITPREFIX, CVSEDITPREFIXLEN) == 0)
      continue;
       (void) strcpy (p, line);
       p += line_length;
       }
   }
   if (fclose (fp) < 0)
      error (0, errno, "warning: cannot close Es", fname);
   if (pre_stbuf_atime != post_stbuf_atime ||
      *messagep != NULL ||
      strcmp (*messagep, "\0") == 0)
   { for (;;)
      { (void) print ("Use log message unchanged or not specified\n"));
        (void) print ("a)short, c)ontinue, e)dit, !)reuse this message unchanged for remaining dir\n"));
        (void) print ("Action: (continue) ");
        (void) flush (stdout);
        line_length = getline (&line, &line_chars_allocated, stdin);
        if (line_length < 0)
        { error (0, errno, "cannot read from stdin" );
           if (unlink_file (fname) < 0)
              error (0, errno,
              "warning: cannot remove temp file Es", fname);
           error (1, 0, "aborting" );
        }
        else if (line_length == 0)
           /* line == "\0" || line == "c" || line == "C" */
           break;
       }
if (*line == 'a' || *line == 'A')
{
    if (unlink_file (fname) < 0)
        error (0, errno, "warning: cannot remove temp file \%s", fname);
    error (1, 0, "aborted by user");
}
if (*line == 'u' || *line == 'U')
go to again;
if (*line == '!' )
    { 
        reuse_log_message = 1;
        break;
    }
}
(void) printf ("Unknown input\a");
}
if (line)
    free (line);
if (unlink_file (fname) < 0)
    error (0, errno, "warning: cannot remove temp file \%s", fname);
    free (fname);

/* Runs the user-defined verification script as part of the commit or import process. This verification is meant to be run whether or not the user included the -m attribute, unlike the do_editor function, this is independant of the running of an editor for getting a message.
*/
void
do_verify (message, repository)
    char *message;
    char *repository;
{
    FILE *fp;
    char *fname;
    int retcode = 0;

#ifdef CLIENT_SUPPORT
    if (client_active)
    /* The verification will happen on the server. */
        return;
#endif

#ifdef
    /* FIXME? Do we really want to skip this on noexec? What do we do for the other administrative files? */
    if (noexec)
        return;
    /* If there's no message, then we have nothing to verify. Can this case happen? And if so why would we print a message? */
    if (message == NULL)
    {
        cvs_output ("No message to verify\n", 0);
        return;
    }
    /* Get a temp filename, open a temporary file, write the message to the temp file, and close the file. */
    fname = cvstemp_filename (1);
    fp = fopen (fname, "w");
    if (fp == NULL)
        error (1, errno, "cannot create temporary file \%s", fname);
    else
    {
        fprintf (fp, "%s", message);
        if (strlen (message) != 0)
            (message)[strlen (message) - 1] = '\0';
        (void) fprintf (fp, "%s", "\n");
        if (fclose (fp) == EOF)
            error (1, errno, "\%s", fname);
    }
    /* Get the name of the verification script to run */
    if (repository != NULL)
        (void) Parse_Info (CVSROOTADM_VERIFYSIG, repository, verifysigproc, 0);
    /* Run the verification script */
    if (verifysig_script)
    {
        run_setup (verifysig_script);
        runMsg (fname);
        if (retcode == run_msg (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL | RUN_SIGIGNORE)) != 0)
        {
            /* Since following error() exits, delete the temp file now. */
        }
/* Delete the temp file */
unlink (fname);
free (fname);

/* callback proc for ParseInfo for resinfo templates this routine basically *
copies the matching template onto the end of the tempfile we are setting *
up */
/* ARGUSED */
static int
rcsinfo_proc (repository, template)
char *repository;
char *template;
{
    static char *last_template;
    FILE *tfp;

    /* nothing to do if the last one included is the same as this one */
    if (last_template && strcmp (last_template, template) == 0)
        return (0);

    /* set up static vars for update_logfile_proc */
    message = xmessage;
    logfp = xlogfp;
    changes = xchanges;

    /* call ParseInfo to do the actual logfile updates */
    (void) ParseInfo (CVSROOTADMLOGINFO, repository, update_logfile_proc, 1);
}

/* Uses setup_tmpfile() to pass the updated message on directly to any *
logfile programs that have a regular expression match for the checked in *
directory in the source repository. The log information is fed into the *
specified program as standard input. */

static FILE *logfp;
static char *message;
static List *changes;

void
Update_Logfile (repository, xmessage, xlogfp, xchanges)
char *repository;
char *xmessage;
FILE *xlogfp;
List *xchanges;
{
    /* nothing to do if the list is empty */
    if (xchanges == NULL || xchanges->list->next == xchanges->list)
        return;

    /* set up static vars for update_logfile_proc */
    message = xmessage;
    logfp = xlogfp;
    changes = xchanges;

    /* call ParseInfo to do the actual logfile updates */
    (void) ParseInfo (CVSROOTADMLOGINFO, repository, update_logfile_proc, 1);
}
static int
update_logfile_proc (repository, filter)
char *repository;
char *filter;
{
    return (logfile_write (repository, filter, message, logfp, changes));
}

/*
 * concatenate each filename/version onto strlist
 */

static int
title_proc (p, closure)
Node *p;
void *closure;
{
struct logfileinfo *li;
char *c;
li = (struct logfile_info *) p->data;
if (li->type == type)
{
    /* Until we decide on the correct logging solution when we add
directories or perform imports, T_TITLE nodes will only
 tack on the name provided, regardless of the format string.
You can verify that this assumption is safe by checking the
code in add.c (add_directory) and import.c (import). */

    str_list = xrealloc (str_list, strlen (str_list) + 5);
    (void) strcat (str_list, "*");

    if (li->type == T_TITLE)
    {
        str_list = xrealloc (str_list, strlen (str_list) + strlen (p->key) + 5);
        (void) strcat (str_list, p->key);
    }
    else
    {
        /* All other nodes use the format string. */

        for (c = str_list; *c != '\0'; c++)
        {
            switch (++c)
            {
            case 'v':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->revold) ? strlen (li->revold) : 0)
                        + 10);
                (void) strcat (str_list, li->revold ? "XXX" : "NONE");
                break;
            case 'u':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->reivnew) ? strlen (li->reivnew) : 0)
                        + 10);
                (void) strcat (str_list, li->reivnew ? "XXX" : "NONE");
                break;
            case 'c':
            case 't':
            case 's':
            case 'd':
            case 'g':
            case 'p':
            case 'r':
            case 'z':
            case 'f':
            case 'q':
                break;
            case 's':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->revold) ? strlen (li->revold) : 0)
                        + 10);
                (void) strcat (str_list, li->revold ? "XXX" : "NONE");
                break;
            case 'p':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->reivnew) ? strlen (li->reivnew) : 0)
                        + 10);
                (void) strcat (str_list, li->reivnew ? "XXX" : "NONE");
                break;
            case 'd':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->revold) ? strlen (li->revold) : 0)
                        + 10);
                (void) strcat (str_list, li->revold ? "XXX" : "NONE");
                break;
            case 'g':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->reivnew) ? strlen (li->reivnew) : 0)
                        + 10);
                (void) strcat (str_list, li->reivnew ? "XXX" : "NONE");
                break;
            case 'r':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->revold) ? strlen (li->revold) : 0)
                        + 10);
                (void) strcat (str_list, li->revold ? "XXX" : "NONE");
                break;
            case 'f':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->reivnew) ? strlen (li->reivnew) : 0)
                        + 10);
                (void) strcat (str_list, li->reivnew ? "XXX" : "NONE");
                break;
            case 'q':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->revold) ? strlen (li->revold) : 0)
                        + 10);
                (void) strcat (str_list, li->revold ? "XXX" : "NONE");
                break;
            case 'z':
                str_list = xrealloc (str_list,
                        strlen (str_list) + (li->reivnew) ? strlen (li->reivnew) : 0)
                        + 10);
                (void) strcat (str_list, li->reivnew ? "XXX" : "NONE");
                break;
        }
    }
    return (0);
}

/*
 * Writes some stuff to the logfile "filter" and returns the status of the
 */
filter program.

static int
logfile_write (repository, filter, message, logfp, changes)
char *repository;
char *filter;
char *message;
FILE *logfp;
List *changes;
{
  FILE *pipefp;
  char *prog;
  char *cp;
  int c;
  int pipestatus;
  char *fmt_percent; /* the location of the percent sign that starts the format string. */

  /* The user may specify a format string as part of the filter. Originally, '%s' was the only valid string. The string that was substituted for it was:
   <repository-name> <file1> <file2> <file3> ....
   Each file was either a new directory/import (T_TITLE), or a added (T_ADD), modified (T_MODIFIED), or removed (T_REMOVED) file.
   It is desirable to preserve that behavior so lots of commitlog scripts won't die when they get this new code. At the same time, we'd like to pass other information about the files (like version numbers, statuses, or checkin times).
   The solution is to allow a format string that allows us to specify those other pieces of information. The format string will be composed of '%' followed by a single format character, or followed by a set of format characters surrounded by '{' and }' as separators. The format characters are:
   s = filename
   V = old version number (pre-checkin)
   v = new version number (post-checkin)
   For example, valid format strings are:
       %
       %s
       %{{sVv}}
   There's no reason that more items couldn't be added (like modification date or file status [added, modified, updated, etc.]) - the code modifications would be minimal (logmsg.c (title.proc) and commit.c (checkfileproc)).
   The output will be a string of tokens separated by spaces. For backwards compatibility, the first token will be the repository name. The rest of the tokens will be comma-delimited lists of the information requested in the format string. For example, if '/u/src/master' is the repository, '%{{sVv}}' is the format string, and three files (ChangeLog, Makefile, foo.c) were modified, the output might be:
   /u/src/master ChangeLog,1,1,1,2 Makefile,1,3,1,4 foo.c,1,12,1,19
   Why this duplicates the old behavior when the format string is '%s' is left as an exercise for the reader. */

  fmt_percent = strchr (filter, '%');
  if (fmt_percent)
    {
      int len;
      char *repos;
      char *fmt_begin, *fmt_end; /* beginning and end of the format string specified in filter. */
      char *fmt_continue; /* where the string continues after the format string (we might skip a '}') somewhere in there... */

      /* Grab the format string. */
      if (((*(fmt_percent + 1) == ' ') || (*(fmt_percent + 1) == '\0'))) /* The percent stands alone. This is an error. We could be treating '%' like any other formatting character, but using it as a formatting character seems like it would be a mistake. */
/* Would be nice to also be giving the line number. */
error(0, 0, "loginfo: '%%' not followed by formatting character");
fmt begin = fmt percent + 1;
fmt end = fmt begin;
fmt continue = fmt begin;
}
else if (*fmt percent + 1 == '(', + 1; } /* The percent has a set of characters following it. */
fmt begin = fmt percent + 2;
fmt end = strchr(fmt begin, ');
if (fmt end)
{ /* Skip over the ']' character. */
fmt continue = fmt end + 1;
}
else
/* There was no close brace - assume that format string continues to the end of the line. */
/* Would be nice to also be giving the line number. */
error(0, 0, "loginfo: '}' missing");
fmt end = fmt begin + strlen(fmt begin);
fmt continue = fmt end;
}
else
/* The percent has a single character following it. FIXME:
   \%\% should expand to a regular percent sign. */
fmt begin = fmt percent + 1;
fmt end = fmt begin + 1;
fmt continue = fmt end;
}
len = fmt end - fmt begin;
strlist format = xmalloc(sizeof(char) * (len + 1));
strncpy(strlist format, fmt begin, len);
strlist format[len] = '\0';
/* Allocate an initial chunk of memory. As we build up the string we will realloc it. */
if (strlist)
strlist = xmalloc(1);
strlist[0] = '\0';
/* Add entries to the string. Don't bother looking for entries if the format string is empty. */
if (strlist format[0] != '\0')
{ 
    type = TITLE; 
    (void) walklist (changes, title proc, NULL); 
    type = ADDED; 
    (void) walklist (changes, title proc, NULL); 
    type = MODIFIED; 
    (void) walklist (changes, title proc, NULL); 
    type = REMOVED; 
    (void) walklist (changes, title proc, NULL); 
}
free(strlist format); 
/* Construct the final string. */
repos = Short Repository (repository);
prog = xmalloc((fmt percent - filter) + strlen(repos) 
+ strlen(strlist) + strlen(fmt continue) 
+ 10);
(void) strncpy prog, filter, fmt percent - filter); 
prog[fmt percent - filter] = '\0';
(void) strcat (prog, '%'); 
(void) strcat (prog, repos); 
(void) strcat (prog, strlist); 
(void) strcat (prog, '%');
(void) strcat (prog, fmt continue);
/* To be nice, free up some memory. */
free(strlist); 
strlist = (char *) NULL;
} else
{
/ There's no format string. */
prog = xstrdup (filter);
}
if (pipefp = run_popen (prog, "w")) == NULL
{
  if (!noexec)
    error (0, 0, "cannot write entry to log filter: \%s", prog);
  free (prog);
  return (1);
}
(void) fprintf (pipefp, "Update of \%s",
(void) fprintf (pipefp, "In directory \%s:",
else
  fprintf (pipefp, "cannot get working directory: \%s",
     strerror (errno));
free (prog);
820 (void)
fprintf (pipefp, "Update of \%s
",
fprintf (pipefp, "In directory \%s:
",
830}
setup_tmpfile (pipefp, "", changes);
(void) fprintf (pipefp, "Log Message: \%s",
(void) fprintf (pipefp, "Status: \%s",
free (cp);
pipestatus = pclose (pipefp);
return (((pipestatus == -1) || (pipestatus == 127)) ? 1 : 0;
} /*
  We choose to use the *last* match within the editinfo file for this
  repository. This allows us to have a global editinfo program for the
  root of some hierarchy, for example, and different ones within different
  sub-directories of the root (like a special checker for changes made to
  the "src" directory versus changes made to the "doc" or "test"
  directories.
*/
#endif
*/ ARGUSED */
static int
editinfo_proc (repository, editor)
  char *repository;
  char *editor;
{
  /* nothing to do if the last match is the same as this one */
  if (editinfo_editor && strcmp (editinfo_editor, editor) == 0)
    return (0);
  if (editinfo_editor)
    free (editinfo_editor);
  editinfo_editor = xstrdup (editor);
  return (0);
}
870 /* This routine is called by ParseInfo. It assigns the name of the
  message verification script to the global variable verify_script */
static int
verifymsg_proc (repository, script)
  char *repository;
  char *script;
{
  if (verifymsg_script && strcmp (verifymsg_script, script) == 0)
    return (0);
  if (verifymsg_script)
    free (verifymsg_script);
  verifymsg_script = xstrdup (script);
  return (0);
A.36 main.c

/∗
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License
 * as specified in the README file that comes with the CVS source distribution.
 * This is the main C driver for the CVS system.
 * Credit to Dick Grune, Vrije Universiteit, Amsterdam, for writing
 * the shell-script CVS system that this is based on.
 */
#include "cvs.h"
#ifndef HAVE_WINSOCK_H
#include <winsock.h>
#else extern int gethostname ();
#endif char *program_name;
char *program_path;
char *command_name;
/* I'd dynamically allocate this, but it seems like gethostname
 requires a fixed size array. If I'm remembering the RFCs right,
 256 should be enough. */
#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN 256
#endif char hostname[MAXHOSTNAMELEN];
int use_editor = 1;
int use_cvsrc = 1;
int cswrite = CVS_READ_DFLT;
int really_quiet = 0;
int quiet = 0;
int trace = 0;
int nocac = 0;
int logoff = 0;
/* Set if we should be writing CVSDM directories at top level. At
 least for now we'll make the default be off (the CVS 1.9, not CVS
 1.9.2, behaviour). */
int top_level_admin = 0;
mode_t cvs_umask = UMASK_DFLT;
char *CurDir;
/* Defaults, for the environment variables that are not set */
char *Tmpdir = TMPDIR_DFLT;
char *Editor = EDITOR_DFLT;
static const struct cmd
{
    char *fullname;  /* Full name of the function (e.g. "commit") */
    /∗ Synonyms for the command, nick1 and nick2. We supply them
 mostly for two reasons: (1) CVS has always supported them, and
 we need to maintain compatibility, (2) if there is a need for a
 version which is shorter than the fullname, for ease in typing.
 Synonyms have the disadvantage that people will see "new" and
 then have to think about it, or look it up, to realize that is
 the operation they know as "add". Also, this means that one
 cannot create a command "cvs new" with a different meaning. So
 new synonyms are probably best used sparingly, and where used
 should be abbreviations of the fullname (preferably consisting
 of the first 2 or 3 or so letters).

 One thing that some systems do is to recognize any unique
 abbreviation, for example "annotat" "annota", etc., for
 "annotate". The problem with this is that scripts and user
 habits will expect a certain abbreviation to be unique, and in
 a future release of CVS it may not be. So it is better to
 accept only an explicit list of abbreviations and plan on
 supporting them in the future as well as now. */
    char *nick1;
    char *nick2;
    int (*func) ();  /* Function takes (argc, argv) arguments. */
} cmds[] =

}
90 {  
  "add",  "ad",  "rew",  add },  
  "admin",  "ade",  "rca",  admin },  
  "annotate",  "ana",  NULL,  annotate },  
  "checkout",  "co",  "get",  checkout },  
  "commit",  "ct",  "com",  commit },  
  "diff",  "dif",  "dif",  diff },  
  "edit",  NULL,  NULL,  edit },  
  "editors",  NULL,  NULL,  editors },  
  "export",  "exp",  "ext",  export },  
  "history",  "hi",  "his",  history },  
  "import",  "im",  "imp",  import },  
  "init",  NULL,  NULL,  init },  
#endif  

100 /*  
     CVS usage messages never have followed the GNU convention of  
     putting metavariables in uppercase. I don't know whether that  
     is a good convention or not, but if it changes it would have to  
     change in all the usage messages. For now, they consistently  
     use lowercase, as far as I know. Punctuation is pretty funky,  
     though. Sometimes they use none, as here. Sometimes they use  
     single quotes (not the TeX-ish "'" stuff), as in --help-options.  
     Sometimes they use double quotes, as in cvs -H add.  
     */  

110 static const char *const usg[] =  
{  
  /* CVS usage messages never have followed the GNU convention of  
     putting metavariables in uppercase. I don't know whether that  
     is a good convention or not, but if it changes it would have to  
     change in all the usage messages. For now, they consistently  
     use lowercase, as far as I know. Punctuation is pretty funky,  
     though. Sometimes they use none, as here. Sometimes they use  
     single quotes (not the TeX-ish "'" stuff), as in --help-options.  
     Sometimes they use double quotes, as in cvs -H add.  
     */  

120 static const char *const cmd_usg[] =  
{  
  /* Some people think that a bug-reporting address should go here. IMHO,  
     the web sites are better because anything else is very likely to go  
     obsolete in the years between a release and when someone might be  
     reading this help. Besides, we could never adequately discuss  
     bug reporting in a concise enough way to put it in a help message. */  

130  /*  
     The Concurrent Versions System (CVS) is a tool for version control.
     */  

140 /*  
     I really don't think I want to try to define "version control"  
     in one line. I'm not sure one can get more concise than the  
     paragraph in .../cvs.spec without assuming the reader knows what  
     version control means. */  

150 /*  
     For CVS updates and additional information, see in
     */  

160 /*  
     I was going to put this at the top, but usage() wants the %s to  
     be in the first line. */  

170 /*  
     * Pascal Molli's CVS site at http://www.loria.fr/~molli/cvs-index.html
     */

180 static const char *const cmd_usage[] =  
{  
  "CVS commands are:\n",  

"The Concurrent Versions System (CVS) is a tool for version control.\n"  
"Pascal Molli's CVS site at http://www.loria.fr/~molli/cvs-index.html\n",  
"Cyclic Software at http://www.cyclic.com/ or\n",  
}  

190  /*  
     * specification is:\n     */
add Add a new file/directory to the repository.
add Administer front end for the client.
annotate Show last revision where each line was modified.
checkout Checkout source for editing.
commit Check files into the repository.
diff Show differences between revisions.
edit Get ready to edit a checked-out file.
editors See who is editing a checked-out file.
export Export source from CVS, similar to checkout.
history Show repository access history.
import Import sources into CVS, using vendor branches.
init Create a CVS repository if it doesn't exist.

static const char *const opt_usage[] =
{
    "CVS global options (specified before the command name) are:",
    "-W Make checked-out files read-write (default).",
    "-R Make checked-out files read-only.",
    "-q Cause CVS to be somewhat quiet.",
    "-Q Cause CVS to be really quiet.",
    "-N Do not execute anything that will change the disk.",
    "-L Turn history logging off.",
    "-n Do not execute anything.",
    "-s VAR=VAL Set CVS user variable.",
    "-t Show trace of program execution -- try with -n.",
    "-i Turn history logging off.",
    "-m Do not execute anything that will change the disk.",
    "-t Show trace of program execution -- try with -n.",
    "-v CVS version and copyright.",
    "(Specify the --help option for a list of other help options)\n", NULL,
};

static const char *const opt_cmd[] =
{
    "log Print out history information for files.
",
    "status Display status information on checked out files.
", "init Create a CVS repository if it doesn't exist.
", "admin Administration front end for rcs
", "rsync CVS command synonyms are:
", "-x Encrypt all net traffic.
", "-t Show trace of program execution -- try with -n.
", "-i Turn history logging off.
", "-m Do not execute anything that will change the disk.
", "-t Show trace of program execution -- try with -n.
", "-v CVS version and copyright.
", "(Specify the --help option for a list of other help options)\n", NULL,
};

static const char *cmd_synonyms() {
    char *synonyms;
    char *line;
    const struct cmd *c = &cmds[0];
    /* Three more for title, "specify --help" line, and NULL. */
    int numcmds = 3,
    while (c->fullname != NULL) {
        numcmds++;
        c++;
    }

    synonyms = (char **) xmalloc(numcmds * sizeof(char *));
    line = synonyms;
    line += "CVS command synonyms are:\n";
    for (c = &cmds[0]; c->fullname != NULL; c++) {
        if (c->nick1 || c->nick2) {
            line = xmalloc(strlen(c->fullname) + 1);
            sprintf(line, "CVS command synonyms are:\n",
                c->fullname,
                c->nick1 ? "nick1: ": ",
                c->nick2 ? "nick2: ": ");
            line++;
    }
    return synonyms;
}
The following commands do not use a checked-out working directory. We conservatively assume that everything else does. Feel free to add to this list if you are certain something doesn’t use the WD. */
if (!strcmp(cmd_name, "checkout") ||
    (strcmp(cmd_name, "status") != 0) &&
    (strcmp(cmd_name, "diff") != 0) &&
    (strcmp(cmd_name, "updata") != 0) &&
    (strcmp(cmd_name, "history") != 0) &&
    (strcmp(cmd_name, "export") != 0) &&
    (strcmp(cmd_name, "import") != 0) &&
    (strcmp(cmd_name, "annote") != 0) &&
    (strcmp(cmd_name, "watchers") != 0) &&
    (strcmp(cmd_name, "cleanup") != 0))
{| ret |= CVS.CmdIgnoreADMROOT; }

return ret;

static RETSIGTYPE
main_cleanup (sig)
int sig;
{
    ifndef DONT_USE_SIGNALS
    const char *name;
    char temp[256];
    switch (sig)
    {
    ifdef SIGINT
    case SIGINT:
        name = "interrupt";
        break;
    endif
    ifdef SIGQUIT
    case SIGQUIT:
        name = "QUIT";
        break;
    endif
    ifdef SIGPIPE
    case SIGPIPE:
        name = "broken pipe";
        break;
    endif
    return ret;
    }
```c
#define SIGTERM

#define DEFAULT

#define TZSET

#define NOT

#define Editor

#define name

#define string

#define env

#define INITIIZE

#define env

#define INITIIZE

#define PROGRAM

#define path

#define update

#define component

#define name

#define tmpdir

#define program

#define tzset

#define int

#define break

#define error(1, 0, "received %s signal", name);

#define / * !DON'T USE SIGNALS */

int main (argc, argv)

int argv;

{ char *argv;

extern char *CVSROOT = CVSROOT_DEFAULT;
extern char *version_string;
extern char *config_string;
char *c, *end;
const struct cmd *cmd = cm;

int c, err = 0;
tmpdir_update_env, cvs_update_env;
free(CVSROOT = 0;
free(Editor = 0;
free_tmpdir = 0;

int help = 0; /* Has the user asked for help? This lets us support the 'cvs -H cmd' convention to give help for cmd. */

static struct option long_options[] =
{ "help", 0, NULL, "H" },
{ "version", 0, NULL, "v" },
{ "help-commands", 0, NULL, 1 },
{ "help-synonyms", 0, NULL, 2 },
{ "help-options", 0, NULL, 4 },
{ "alias-root", required_argument, NULL, 3 },
{ 0, 0, 0, 0 };

/* getopt_long stores the option index here, but right now we don't use it. */
int option_index = 0;
int need_to_create_root = 0;

#define SYSTEM_INITIALIZATION

#define \ Hook for OS-specific behavior, for example socket subsystems on NT and OS2 or dealing with windows and arguments on Mac. */

#define SYSTEM_INITIALIZATION &args, &argv);

#endif

#define HAVE_TZSET

/* On systems that have tzset (which is almost all the ones I know of), it's a good idea to call it. */

tzset ();

#endif

/* Just save the last component of the path for error messages */

#define program_path = xstrdup (argv[0]);

#endif

#define ARGV\NOT\PROGRAM\NAME

/* On some systems, e.g. VMS, argv[0] is not the name of the command which the user types to invoke the program. */

#define program_name = "cvs";

else

program_name = last_component (argv[0]);

#endif

/* Query the environment variables up-front, so that * they can be overridden by command line arguments */

cvs\update\env = 0;

tmpdir_update_env = "tmpdir; /* TMPDIR_DEFAULT must be set */

if ((cp = getenv (TMPDIR\_ENV)) != NULL) {
  Tmpdir = cp;
  tmpdir_update_env = 0; /* it's already there */
}

if ((cp = getenv (EDITOR\_ENV)) != NULL)
  Editor = cp;
else if ((cp = getenv (EDITOR2\_ENV)) != NULL)
  Editor = cp;
```

Editor = cp;
class if ( (cp = getenv (EDITOR_ENV)) != NULL)
    Editor = cp;
if ((cp = getenv (CVSROOT_ENV)) != NULL)
{
    CVSmroot = cp;
    cvs_update_env = 0; /* it's already there */
}
if (getenv (CVSREAD_ENV) != NULL)
cvswrite = 0;
/* Set this to 0 to force getopt initialization. getopt() sets
this to 1 internally. */
optind = 0;
/* We have to parse the options twice because else there is no
chance to avoid reading the global options from ".cvsrc". Set
opterr to 0 for avoiding error messages about invalid options.
*/
opterr = 0;
while ((c = getopt_long(argc, argv, "+f", NULL, NULL)) != EOF)
{
    if (c == 'f')
        use_cvsrc = 0;
}
/* Scan .cvsrc file for global options. */
if (use_cvsrc)
    read_cvsrc (&argc, &argv, "cvs");
optind = 0;
opterr = 1;
while ((c = getopt_long(argc, argv, "Qqrwtnlvb:T:e:d:Hfz:s:xa", long_options, &option_index)) != EOF)
{
    switch (c)
    {
    case 1:
        /* help-commands */
        usage (cmd_usage);
        break;
    case 2:
        /* help-synonyms */
        usage (cmd_synonyms());
        break;
    case 4:
        /* help-options */
        usage (opt_usage);
        break;
    case 3:
        /* allow-root */
        root_allowopt (optarg);
        break;
    case 'Q':
        really_quiet = 1;
        /* FALL THROUGH */
    case 'q':
        quiet = 1;
        break;
    case 'r':
        cvsadmin (optarg);
        break;
    case 'w':
        cvswrite = 1;
        break;
    case 't':
        trace = 1;
        break;
    case 'n':
        noexec = 1;
        break;
    case 'l':
        logoff = 1;
        break;
    case 'y':
        /* Having the year here is a good idea, so people have
some idea of how long ago their version of CVS was
released. */
        (void) fputs (version_string, stdout);
        (void) fputs (config_string, stdout);
        (void) fputs ("\n\nCopyright (c) 1989-1998 Brian Berliner, david d 'zoo' zuhn, \nJeff Polk, and other authors\n\n", stdout);
(void) fputs("\n", stderr);
(void) fputs("CVS may be copied only under the terms of the GNU General Public License,\n", stderr);
(void) fputs("a copy of which can be found with the CVS distribution kit.\n", stderr);
(void) fputs("\n", stderr);

(void) fputs("Specify the --help option for further information about CVS\n", stderr);

exit (0);
break;
case "n":
/* This option used to specify the directory for RCS executables. But since we don't run them any more, 
this is a noop. Silently ignore it so that .cvrc and scripts and std.conf and such can work with
either new or old CVS. */
break;
case "T":
Tmpdir = xstrdup(optarg);
free(Tmpdir) = 1;
tmpdir_update_env = 1; /* need to update environment */
break;
case "e":
Editor = xstrdup(optarg);
free(Editor) = 1;
break;
case "d":
CVSroot = xstrdup(optarg);
free(CVSroot) = 1;
cvs_update_env = 1; /* need to update environment */
break;
case "H":
help = 1;
break;
case "t":
usecvrc = 0; /* unnecessary, since we've done it above */
break;
case "x":
#endif
CLIENT_SUPPORT

gzip_level = atof(optarg);
if (gzip_level <= 0 || gzip_level > 9)
error (1, b,
"gzip compression level must be between 1 and 9");
#endif
/* If no CLIENT_SUPPORT, we just silently ignore the gzip level, so that users can have it in their .cvrc and not cause any trouble. */
break;
case "s":
variable_set (optarg);
break;
case "x":
#endif
CLIENT_SUPPORT

cvsencrypt = 1;
#endif
/* If no CLIENT_SUPPORT, ignore -x, so that users can have it in their .cvrc and not cause any trouble. We will issue an error later if stream authentication is not supported. */
break;
case "+":
default: usage (usg);
break;
}
}
}

for (cm = cmds; cm->fullname; cm++)
{
if (cm->nick1 && bstrcmp(command_name, cm->fullname))
break;
if (cm->nick2 && bstrcmp(command_name, cm->fullname))
break;
if (!strcmp(command_name, cm->fullname))

/* Look up the command name. */

command_name = argv[0];
for (cm = cmds; cm->fullname; cm++)
{
if (cm->nick1 && bstrcmp(command_name, cm->fullname))
break;
if (cm->nick2 && bstrcmp(command_name, cm->fullname))
break;
if (!strcmp(command_name, cm->fullname))

/* Look up the command name. */


break;

if (lcm->fullname) 
  usage (cm Usage);  /* no match */
else 
  command_name = cm->fullname;  /* Global pointer for later use */
/* This should probably remain a warning, rather than an error, */
/* for quite a while. For one thing the version of VC distributed */
/* with GNU emacs 19.94 invokes 'cvs rlog' instead of 'cvs log'. */
640  if (strcmp (argv[0], "rlog") == 0) 
  { 
    error (0, 0, "warning; the rlog command is deprecated"); 
    error (0, 0, "use the synonymous log command instead"); 
  }
if (help) 
  argc = -1;  /* some functions only check for this */
else 
  { 
  */ The user didn't ask for help, so go ahead and authenticate, */
  /* set up CVSROOT, and the rest of it. */
  */ The UMASK environment variable isn't handled with the */
  /* others above, since we don't want to signal errors if the */
  /* user has asked for help. This won't work if somebody adds */
  /* a command-line flag to set the umask, since we'll have to */
  /* parse it before we get here. */
  if (cp = getenv (CVSUMASK_ENV)) != NULL) 
760  { 
    */ FIXME: Should be accepting symbolic as well as numeric mask. */
    /* cvsumask = strtol (cp, &end, 8) & 0777;
    if (end == '0') 
      error (1, errno, "invalid umask value in %s (Is),
        CVSUMASK_ENV, cp);
  } 
  if (defined (AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI)) && defined (SERVER_SUPPORT) 
760  if (strcmp (command_name, "server") == 0) ||
      (strcmp (command_name, "kserver") == 0) ||
      (strcmp (command_name, "gserver") == 0) { 
      static int authenticated = 0;
      /* The reason that --allow-root is not a command option */
      /* is mainly the comment in server() about how argc, argv */
      /* might be from .cvsrc. I'm not sure about that, and */
      /* I'm not sure it is only true of command options, but */
      /* it seems easier to make it a global option. */
      
      if (authenticated) 
580  { 
        */ send the list of allowed auth modes to the client */
      ifdef AUTH_SERVER_SUPPORT 
        print ("password");
      ifdef AUTH_SERVER_SUPPORT 
        print ("gssapi");
      ifdef AUTH_SERVER_SUPPORT 
590  printf ("kerberos 94");
      ifdef SERVER_SUPPORT 
600  server_active = strcmp (command_name, "server") == 0;
      /* Fiddling with CVSROOT doesn't make sense if we're running */
      /* in server mode, since the client will send the repository */
      /* directory after the connection is made. */
      if (server_active) 
700  
      endif /* (AUTH_SERVER_SUPPORT || HAVE_GSSAPI) && SERVER_SUPPORT */
      endif SERVER_SUPPORT 
700  printf ("*/


char *CVSADMRoot;

/* See if we are able to find a 'better' value for CVSroot
   in the CVSADM_ROOT directory. */
CVSADMRoot = NULL;

/* cvs import shouldn't check CVS/Root; in general it
   ignores CVS directories and CVS/Root is likely to
   specify a different repository than the one we are
   importing to. */
if (lookup_command_attribute(command_name) & CVS_CMD_IGNOREADMROOT) {
    CVSADMRoot = NameRoot((char *) NULL, (char *) NULL);
}
if (CVSADMRoot != NULL) {
    CVSroot = CVSADMRoot;
cvs_update_env = 1; /* need to update environment */
    /* Let -d override CVS/Root file. The user might want
       to change the access method, use a different server
       (if there are two server machines which share the
       repository using a networked file system), etc. */
    else if (
        ifndef CLIENTSUPPORT
            getenv("CVS_IGNORE_REMOTE_ROOT") & &
        ifdef
            strcmp (CVSroot, CVSADMRoot) != 0)
            /* Once we have verified that this root is usable,
               we will want to write it into CVS/Root.
               Don't do it for the "login" command, however.
               Consider: if the user executes "cvs login" with
               the working directory inside an already checked
               out module, we'd incorrectly change the
               CVS/Root file to reflect the CVSROOT of the
               "cvs login" command. Ahh, the things one
               discovers. */
        if (lookup_command_attribute(command_name) & CVS_CMD_USES_WORK_DIR) {
            need_to_create_root = 1;
        }
    }
}

/* Now we've reconciled CVSROOT from the command line, the
   CVS/Root file, and the environment variable. Do the
   last sanity checks on the variable. */
if (! CVSroot) {
    error (0, 0, "No CVSROOT specified! Please use the '-d' option");
    error (1, 0, "or set the %s environment variable.", CVSROOT_ENV);
}
if (! CVSroot) {
    error (0, 0, "CVSROOT is set but empty! Make sure that the");
    error (0, 0, "specification of CVSROOT is legal, either via the");
    error (0, 0, "'CVSROOT is set but empty! Make sure that the");
    error (1, 0, "CVSROOT_ENV");
    error (1, 0, "CVS/Root file (if any)");
}

/* Now we're 100% sure that we have a valid CVSROOT
   variable. Parse it to see if we're supposed to do
   remote accesses or use a special access method. */
if (parse_cvsroot (CVSroot))
    error (1, 0, "Bad CVSROOT.");

    */
    * Check to see if we can write into the history file. If not,
    * we assume that we can't work in the repository.
Debe, only if the history file exists.

/*
 * If cleanup is active
 * {
 * char *path;
 * int save_errno;
 * 
 * path = xmalloc(strlen(CVSroot_directory) + sizeof (CVSROOTADM) + 20 + sizeof (CVSROOTADM_HISTORY));
 * (void) sprintf(path, "%s/%s", CVSroot_directory, CVSROOTADM);
 * if (isaccessible(path, R_OK | W_OK))
 * {
 * save_errno = errno;
 * /* If this is "cvs init", the root need not exist yet. */
 * if (strcmp(command, "init") != 0)
 * {
 * error(1, save_errno, "%s", path);
 * }
 * }
 * else strcat(path, ".");
 * (void) strcat(path, CVSTARADM_HISTORY);
 * if (isfile(path) && isaccessible(path, R_OK | W_OK))
 * {
 * save_errno = errno;
 * error(0, 0, "Sorry, you don't have read/write access to the history file");
 * error(1, save_errno, "%s", path);
 * }
 * free(path);
 * }
 *
* If evw update env
* {
* char *env;
* env = xmalloc(strlen(CVSROOT_ENV) + strlen(CVSroot) + 1 + 1);
* (void) sprintf(env, "%s=%s", CVSROOT_ENV, CVSroot);
* (void) putenv(env);
* /* do not free env, as putenv has control of it */
* }
* */
*/

/* This is only used for writing into the history file. For remote connections, it might be nice to have hostname and/or remote path, on the other hand I'm not sure whether it is worth the trouble. */

#elif defined SERVER_SUPPORT
If server is active
CurDir = xstrdup("<remote>");
else
{
CurDir = xgetwd();
if (CurDir == NULL)
error(1, errno, "cannot get working directory");
}
if (Tmpdir == NULL || Tmpdir[0] == '\0')
Tmpdir = "/tmp";

#elif defined HAVE_PUTENV
If tmpdir update env
{
char *env;
env = xmalloc(strlen(TMPDIR_ENV) + strlen(Tmpdir) + 1 + 1);
(void) sprintf(env, "%s=%s", TMPDIR_ENV, Tmpdir);
(void) putenv(env);
/* do not free env, as putenv has control of it */
}

#elif defined DONT_USE_SIGNALS
/* make sure we clean up on error */
#endif

#elif defined SIGUP
(void) SIG_register(SIGUP, main_cleanup);
(void) SIG_register(SIGUP, LockCleanup);
#endif

#elif defined SIGINT
(void) SIG_register(SIGINT, main_cleanup);
(void) SIG_register(SIGINT, LockCleanup);
#endif

#elif defined SIGQUIT
(void) SIG_register(SIGQUIT, main_cleanup);
#endif
(void) SIGregister (SIGQUIT, LockCleanup);
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
.nasa
{ 
  time_t unixtime;

  unixtime = get_date (rawdate, (struct timeb *) NULL);
  if (unixtime == (time_t) - 1)
    error (1, 0, "Can't parse date/time: %s", rawdate);
  return date_from_time_t (unixtime);
}

/* Convert a time_t to an RCS format date. This is mainly for the use of "cvs history", because the CVSROOT/history file contains time_t format dates; most parts of CVS will want to avoid using time_t's directly, and instead use RCS_datecmp, Make_Date, &c. Assuming that the time_t is in GMT (as it generally should be), then the result will be in GMT too.

Returns a newly malloc'd string. */

char *
date_from_time_t (unixtime)

1000  time_t unixtime;

1010  { 
struct tm *ftm;
char date[MAXDATELEN];
char *ret;

  ftm = gmtime (&unixtime);
  if (ftm == NULL)
    /* This is a system, like VMS, where the system clock is in local time. Hopefully using localtime here matches the "zero timezone" hack I added to getdate (getdate of course being the relevant issue for Make_Date, and for history.c too I think). */
  ftm = localtime (&unixtime);

  (void) strftime (date, DATEFORM,
    ftm->tm_year + (ftm->tm_year < 100 ? 0 : 1900),
    ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
    ftm->tm_min, ftm->tm_sec);
  ret = xstrdup (date);
  return (ret);
}

1030 void
usage (cpp)

1040  register const char *const *cpp;

1050  { 
(void) fprintf (stderr, *cpp++,
    "usage: \%s \%s
", program_name, command_name);

1060    for (; *cpp; cpp++)
      (void) fprintf (stderr, *cpp);
  error_exit ();
}
A.37 mkmodules.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS kit. */

#include "cvs.h"
#include "savcred.h"
#include "getline.h"

#define DBLKSIZ 4096 /* since GNU ndbm doesn't define it */

static int checkout_file PROTO(char *file, char *temp);
static char *maketemplate PROTO(void);
static void rename_file PROTO(char *temp, char *real);

#define MY_NDBM
static void rename_file PROTO(char *temp);
static void write_bmfile PROTO(char *temp);

/* Structure which describes an administrative file. */
struct adminfile { /* Name of the file, within the CVSROOT directory. */
    char *filename;
    /* This is a one line description of what the file is for. It is not
currently used, although one wonders whether it should be, somehow.
If NULL, then don't process this file in mkmodules (FIXME?: a bit of
a kludge; probably should replace this with a flags field). */
    char *errmsg;
    /* Contents which the file should have in a new repository. To avoid
problems with brain-dead compilers which choke on long string constants,
this is a pointer to an array of char * terminated by NULL—each of
the strings is concatenated. */
    const char *const *contents;
};

static const char *const loginfo_contents[] = {
    "# The 'loginfo' file controls where 'cvs commit' log information\n", # Is sent. The first entry on a line is a regular expression which must match\n", "# the directory that the change is being made to, relative to the\n", "# $CVSROOT. If a match is found, then the remainder of the line is a filter\n", "# program that should expect log information on its standard input.\n", "#\n", "# If the repository name does not match any of the regular expressions in this\n", "# file, the '""$CVSROOT/commitlog' line is used, if it is specified.\n", "#\n", "# If the name ALL appears as a regular expression it is always used\n", "# in addition to the first matching regex or DEFAULT.\n", "#\n", "# You may specify a format string as part of the\n", "# filter. The string is composed of a '%' followed by\n", "# by a single format character, or followed by a set of format\n", "# characters surrounded by '{' and '}' as separators. The format\n", "# characters are: \n", "#\n", "# a = file name\n", "# v = old version number (pre-checkin)\n", "# V = new version number (post-checkin)\n", "#\n", "# For example:\n", "#DEFAULT (echo '\"\"; id; echo \$a; date; cat) >> $CVSROOT/\n", "#\n", "#DEFAULT (echo '\"\"; id; echo $a; date; cat) >> $CVSROOT/commitlog\n", NULL
};

static const char *const rcsinfo_contents[] = {
    "# The '\rcsinfo' file is used to control templates with which the editor\n", "# is invoked on commit and import.\n", "#\n", "# The first entry on a line is a regular expression which is tested\n", "# against the directory that the change is being made to, relative to the\n", "# $CVSROOT. For the first match that is found, then the remainder of the\n", "# line is the name of the file that contains the template.\n", "#\n", "# If the repository name does not match any of the regular expressions in this\n", "#
"# file, the "\DEFAULT\" line is used, if it is specified.\n",
"# file.
"# If the name "\ALL\" appears as a regular expression it is always used\n",
"# in addition to the first matching regex or "\DEFAULT\".\n",
NULL
};

static const char *const editinfo_contents[] = {
  "The "\editinfo\" file is used to allow verification of logging\n",
  "information. It works best when a template (as specified in the\n",
  "rcinfo file) is provided for the logging procedure. Given a\n",
  "template with locations for, a bug-id number, a list of people who\n",
  "reviewed the code before it can be checked in, and an external\n",
  "process to catalog the differences that were code reviewed, then\n",
  "following test can be applied to the code:\n",
  "\#\n",
  "# Making sure that the entered bug-id number is correct.\n",
  "# Validating that the code that was reviewed is indeed the code being\n",
  "# checked in (using the bug-id number or a separate review\n",
  "# number to identify this particular code set.).\n",
  "#\n",
  "# If any of the above test failed, then the commit would be aborted.\n",
  "#\n",
  "# Actions such as mailing a copy of the report to each reviewer are\n",
  "# better handled by an entry in the logininfo file.\n",
  "#\n",
  "# One thing that should be noted is the the ALL keyword is not\n",
  "# supported. There can be only one entry that matches a given\n",
  "# repository.\n",
  NULL
};

static const char *const verifymsg_contents[] = {
  "The "\verifymsg\" file is used to allow verification of logging\n",
  "information. It works best when a template (as specified in the\n",
  "rcinfo file) is provided for the logging procedure. Given a\n",
  "template with locations for, a bug-id number, a list of people who\n",
  "reviewed the code before it can be checked in, and an external\n",
  "process to catalog the differences that were code reviewed, then\n",
  "following test can be applied to the code:\n",
  "\#\n",
  "# Making sure that the entered bug-id number is correct.\n",
  "# Validating that the code that was reviewed is indeed the code being\n",
  "# checked in (using the bug-id number or a separate review\n",
  "# number to identify this particular code set.).\n",
  "#\n",
  "# If any of the above test failed, then the commit would be aborted.\n",
  "#\n",
  "# Actions such as mailing a copy of the report to each reviewer are\n",
  "# better handled by an entry in the logininfo file.\n",
  "#\n",
  "# One thing that should be noted is the the ALL keyword is not\n",
  "# supported. There can be only one entry that matches a given\n",
  "# repository.\n",
  NULL
};

static const char *const commitinfo_contents[] = {
  "The "\commitinfo\" file is used to control pre-commit checks.\n",
  "The filter on the right is invoked with the repository and a list \n",
  "of files to check. A non-zero exit of the filter program will \n",
  "cause the commit to be aborted.\n",
  "\#\n",
  "# The first entry on a line is a regular expression which is tested\n",
  "# against the directory that the change is being committed to, relative\n",
  "# to the REVHOLD. For the first match that is found, then the remainder\n",
  "# of the line is the name of the filter to run.\n",
  "#\n",
  "# If the repository name does not match any of the regular expressions in this\n",
  "# file, the \DEFAULT\" line is used, if it is specified.\n",
  "#\n",
  "# If the name "\ALL\" appears as a regular expression it is always used\n",
  "# in addition to the first matching regex or \DEFAULT\".\n",
  NULL
};

static const char *const taginfo_contents[] = {
  "The "\taginfo\" file is used to control pre-tag checks.\n",
  "The filter on the right is invoked with the following arguments:\n",
  "\#\n",
  "# $1 -- tagname\n",
  "# $2 -- operation \"add\" for tag, \"mov\" for tag -F, and \"del\" for tag -d\n",
  "# $3 -- repository\n",
  "# $4 -- file revision [file revision ...]\n",
  "\#\n",
  "# A non-zero exit of the filter program will cause the tag to be aborted.\n",
  "#\n",
  "# The first entry on a line is a regular expression which is tested\n",
  "# against the directory that the change is being committed to, relative\n",
  "# to the REVHOLD. For the first match that is found, then the remainder\n",
  NULL
};
"# of the line is the name of the filter to run.\n",
"#\n",
"# If the repository name does not match any of the regular expressions in this\n",
"file, the \"DEFAULT\" line is used, if it is specified.\n",
"#\n",
"# If the name \"ALL\" appears as a regular expression it is always used\n",
"# in addition to the first matching regex or \"DEFAULT\".\n",
NULL\n};

static const char *const checkoutlist_contents[] = {
"# The \"checkoutlist\" file is used to support additional version controlled\n",
"# administrative files in $CVSROOT/CVSROOT, such as template files.\n",
"#\n",
"# The first entry on a line is a filename which will be checked out from\n",
"# the corresponding RCS file in the $CVSROOT/CVSROOT directory.\n",
"# The remainder of the line is an error message to use if the file cannot\n",
"# be checked out.\n",
"#\n",
"# File format:\n",
"#\n",
"# [whitespace]<filename><whitespace><error message><end-of-line>\n",
"#\n",
"# comment lines begin with \#'\n",
NULL\n};

static const char *const cvswrappers_contents[] = {
"# This file affects handling of files based on their names.\n",
"#\n",
"# The -t/-f options allow one to treat directories of files\n",
"# as a single file, or to transform a file in other ways on\",
"# its way in and out of CVS.\n",
"#\n",
"# The -m option specifies whether CVS attempts to merge files.\n",
"#\n",
"# The -k option specifies keyword expansion (e.g. -kb for binary).\n",
"#\n",
"# Format of wrapper file ($CVSROOT/CVSROOT/cvswrappers or .cvswrappers)\n",
"#\n",
"# wildcard [option value][option value]...\n",
"#\n",
"# where option is one of\n",
"# -f from cvs filter value: path to filter\n",
"# -t to cvs filter value: path to filter\n",
"# -m update methodologyvalue: MERGE or COPY\n",
"# -k expansion mode value: b, o, kkv, &c\n",
"#\n",
"# and value is a single-quote delimited value.\n",
"# For example:\n",
"#*.gif -k 'b'\n",
NULL\n};

static const char *const notify_contents[] = {
"# The \"notify\" file controls where notifications from watches set by\n",
"# \"cvs watch add\" or \"cvs edit\" are sent. The first entry on a line is\n",
"# a regular expression which is tested against the directory that the\n",
"# change is being made to, relative to the $CVSROOT. If it matches,\n",
"# then the remainder of the line is a filter program that should contain\n",
"# one occurrence of %s for the user to notify, and information on its\n",
"# standard input.\n",
"#\n",
"# \"ALL\" or \"DEFAULT\" can be used in place of the regular expression.\n",
"#\n",
"# For example:
",
"# mail %s -s "CVS notification"
",
NULL\n};

static const char *const modules_contents[] = {
"# Three different line formats are valid:\n",
"# key -a aliases...\n",
"# key [options] directory\n",
"# key [options] directory files...\n",
"#\n",
"# Where \"options\" are composed of:\n",
"# -i prog Run \"prog\" on \"cvs commit\" from top-level of module.\n",
"# -o prog Run \"prog\" on \"cvs checkout\" of module.\n",
"# -e prog Run \"prog\" on \"cvs export\" of module.\n",
"# -t prog Run \"prog\" on \"cvs rtag\" of module.\n",
"# -u prog Run \"prog\" on \"cvs update\" of module.\n",
"# -d dir Place module in directory \"dir\" instead of module name.\n",
"# -l Top-level directory only -- do not recurse.\n",
"#\n",
"# NOTE: If you change any of the \"Run\" options above, you'll have to\n",
"# release and re-checkout any working directories of these modules.\n",
"#\n",
"# And \"directory\" is a path to a directory relative to $CVSROOT.\n",
"#\n"
"The \"-a\" option specifies an alias. An alias is interpreted as if\n", "everything on the right of the \"-a\" had been typed on the command line.\n", "\#&", "You can encode a module within a module by using the special \&\".\n", "# character to interpose another module into the current module. Thus\n", "# can be useful for creating a module that consists of many directories\n", "# spread out over the entire source repository.\n", "# everything on the right of the "-a" had been typed on the command line.
", "$ SystemAuth=no
", "$ level of the new working directory when using the 'cvs checkout' command.
", "# The "-a" option specifies an alias. An alias is interpreted as if
", "# can be useful for creating a module that consists of many directories
", "# character to interpose another module into the current module. This
", "# another password, etc, and so saving old passwords, even hashed, is probably not a good idea. */
", "$ Set 'PreservePermissions' to 'yes' to save file status information
", "null CONFIG
", "null PASSWD
", "null PASSWD here too. This
", "null PASSWD contains "all" passwords, not just the ones
", "null currently being used. For example, it could be too easy to
", "null accidentally give someone read only access to CVSROOTADM
", "null (e.g. via anonymous CVS or cvsweb), and then if there are any
", "null guessable passwords for read/write access (usually there will be)
", "null they get read/write access."
", "null Another worry is the implications of storing old passwords--if
", "null someone used a password in the past they might be using it
", "null elsewhere, using a similar password, etc, and so saving old
", "null passwords, even hashed, is probably not a good idea. */
", "$ Set 'PreservePermissions' to 'yes' to save file status information
", "null TOPLEVELADMIN
", "null TOPLEVELADMIN is special-cased in mkmodules.
", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADMIN=mo/no\n", "null TOPLEVELADM
/ * Rebuild the checked out administrative files in directory DIR. */

int
mkmodules (dir)
char *dir;
{
    struct savedcwd cwd;
    char *temp;
    char *cp, *last, *fname;

ifdef MY_NDBM
    DBM = db;
endif

FILE *fp;
char *line = NULL;
size_t line_allocated = 0;
const struct adminfile *fileptr;

if (savedcwd &cwd)
    errorexit (1);
if (CVS_CHDIR (dir) < 0)
    error (1, errno, "cannot chdir to %s", dir);

/* First, do the work necessary to update the "modules" database. */
temp = make_tempfile ();
switch (checkout_file (CVSROOTADM_MODULES, temp))
{
    case 0:
    /* everything ok */
#ifdef MY_NDBM
    /* open it, to generate any duplicate errors */
    if (db = dbm_open (temp, O_RDONLY, 0666)) != NULL
        dbm_close (db);
#endif
    else
        writelnbfile (temp);
    renamebfile (temp);
#endif
    renamecfile (temp, CVSROOTADM_MODULES);
    break;

    case -1:
    /* fork failed */
    (void) unlink_file (temp);
    error (1, errno, "cannot check out %s", CVSROOTADM_MODULES);
    /* NOTREACHED */
    default:
    error (0, 0,
        ""cvs checkout" is less functional without a %s file",
        CVSROOTADM_MODULES);
    break;
}

(void) unlink_file (temp);
free (temp);

/* Checkout the files that need it in CVSROOT dir */
for (fileptr = filelist; fileptr && fileptr->filename; fileptr++)
{
    if (fileptr->errormsg == NULL)
        continue;
    temp = make_tempfile ();
    if (checkout_file (fileptr->filename, temp) == 0)
        renamecfile (temp, fileptr->filename);

#if 0
    /* If there was some problem other than the file not existing, */
    /* checkout file already printed a real error message. If the */
    /* file does not exist, it is harmless--it probably just means */
    /* that the repository was created with an old version of CVS */
    /* which didn't have so many files in CVSROOT. */
    /* */
#else
    error (0, 0, fileptr->errormsg, fileptr->filename);
#endif
    (void) unlink_file (temp);
    free (temp);
}

fp = CVS_FOPEN (CVSROOTADM_CHECKOUTLIST, "r");
if (fp)
{
    /* File format:
        * 
        * <whitespace><filename><whitespace><error message><end-of-line>
        * 
        * comment lines begin with "%s"
        */
    while (getline (&line, &line_allocated, fp) >= 0)
    {
        /* skip lines starting with "%s" */
}
if (line[0] == 'n')
    continue;
if ((last = strrchr (line, '\n')) != NULL)
    *last = '\0';  /* strip the newline */
/* Skip leading white space */
for (fname = line; *fname && isspace(*fname); fname++)
    *cp = '\0';
temp = make tempfile();
if ((checkout_file (fname, temp) == 0)
    rename_rcsfile (temp, fname);
else
    for (cp = fname; *cp && isspace(*cp); cp++)
        error (0, 0, cp, fname);
free (temp);
if (line)
    free (line);
if (error (fp))
    error (0, errno, "cannot read %s", CVSROOTADM_CHECKOUTLIST);
if (fclose (fp) < 0)
    error (0, errno, "cannot close %s", CVSROOTADM_CHECKOUTLIST);
#else
    for (cp++, cp < last && *last && isspace(*last); cp++)
        if (*cp && isspace(*cp))
            error (0, cp, fname);
free (temp);
#endif
if (restore_cwd (&cwd, NULL))
    error_exit();
free_cwd (&cwd);
return (0);

*/ Yeah, I know, there are NFS race conditions here.
*/
static char *
make_tempfile ()
{
    static int seed = 0;
    int fd;
    char *temp;
    if (seed == 0)
        seed = getpid();
    temp = xmalloc (sizeof (BAKPREFIX) + 40);
    while (1)
        { (void) sprintf (temp, "%s%d", BAKPREFIX, seed++);
          if (!(fd = CVS_OPEN (temp, O_CREAT|O_EXCL|O_RDWR, 0666)) != -1)
              break;
          if (errno != EEXIST)
              error (1, errno, "cannot create temporary file %s", temp);
        }
    if (close(fd) < 0)
        error(1, errno, "cannot close temporary file %s", temp);
    return temp;
}

static int
checkout_file (file, temp)
char *file;
char *temp;
{
    char rcs;  RCSNode *rcsnodes;
    int retcode = 0;
    if (noexec)
        return 0;
    rcs = xmalloc (strlen (file) + 5);
    strcpy (rcs, file);
strcat (rcs, RCSEXTC);

if (!isfile (rcs))
    { free (rcs);
      return (1);
    }
rcsnodel = RCSparsecfiles (rcs);
retcode = RCScheckout (rcsnodel, NULL, NULL, NULL, NULL, temp,
    (RCSCHECKOUTPROC) NULL, (void *) NULL);
if (retcode != 0)
    { error (0, 0, "failed to checkout %s file",
        file);
      freercsnodel (&rcsnodel);
      free (rcs);
      return (retcode);
    }
#endif MY_NDBM

static void
write_dbmfile (temp)
char *temp;
{
    char line[DBLKSIZ], value[DBLKSIZ];
    FILE *fp;
    DBM *db;
    char *cp, *vp;
    datum key, val;
    int len, cont, err = 0;
    fp = open_file (temp, "r");
    if ((db = dbmopen (temp, 0|ORDWR | O_CREAT | O_TRUNC, 0666)) == NULL)
        error (1, errno, "cannot open dbm file %s for creation",
            temp);
    for (cont = 0 ; fgets (line, sizeof (line), fp) != NULL ;)
        if ((cp = strchr (line, '\n'))) != NULL
            *cp = '\0'; /* strip the newline */
        vp = value;
        if (cont)
            vp += strlen (value);
        /* See if the line we read is a continuation line, and strip the */
        /* backslash if so. */
        if (len > 0)
            cp = &line[len - 1];
        else
            if (*cp == '\n')
                { cont = 1;
                  *cp = '\0';
                }
        else
            { cont = 0;
              }
    while (*vp && ispace (*vp))
        { vp = value;
          continue; /* empty line */
        }
    while (*vp && ispace (*vp))
        { vp = value;
          continue; /* comment line */
        }
    while (*vp)
        { key.dptr = vp;
          while (*vp && ispace (*vp))
              { vp++;
                key.dsize = vp - key.dptr;
                *vp++ = '\0'; /* NULL terminate the key */
            }
        while (*vp && ispace (*vp))
            { vp++;
              /* skip whitespace to value */
          if (*vp == '\0')
              { }
error (0, 0, "warning: NULL value for key 'la'. key:ptr);  
continue;
}
val:ptr = vp;
val:size = strlen (vp);
if (db:store (db, key, val, DBM::INSERT) == 1)
{
  error (0, 0, "duplicate key found for 'la'. key:ptr);  
err++;  
}
}
}
dbm:close (db);  
(void) fclose (fp);
if (err)
{
  char dotdir[50], dotpag[50], dotdb[50];
  (void) sprintf (dotdir, "%s.dir", temp);
  (void) sprintf (dotpag, "%s.pag", temp);
  (void) sprintf (dotdb, "%s.db", temp);
  (void) unlink_file (dotdir);
  (void) unlink_file (dotpag);
  (void) unlink_file (dotdb);
  error (1, 0, "DBM creation failed; correct above errors");
}
}
static void
rename_bdbfile (temp)
  char *temp;
  
660
{
  char newdir[50], newpag[50], newdb[50];
  char dotdir[50], dotpag[50], dotdb[50];
  char bakdir[50], bakpag[50], bakdb[50];
  (void) sprintf (dotdir, "%s.dir", temp);
  (void) sprintf (dotpag, "%s.pag", temp);
  (void) sprintf (dotdb, "%s.db", temp);
  (void) unlink_file (dotdir);
  (void) unlink_file (dotpag);
  (void) unlink_file (dotdb);
  (void) chmod (newdir, 0666);
  (void) chmod (newpag, 0666);
  (void) chmod (newdb, 0666);
  /* don't mess with me */

SIG_beginCrSect ();
{
  (void) unlink_file (bakdir);
  /* rm $modules.dir $modules.pag */
  (void) unlink_file (bakpag);
  (void) unlink_file (bakdb);
  (void) CVS::RENAME (dotdir, bakdir);
  /* mv $modules.dir $modules.dir */
  (void) CVS::RENAME (dotpag, bakpag);
  /* mv $modules.pag $modules.pag */
  (void) CVS::RENAME (dotdb, bakdb);
  /* mv $modules.db $modules.db */
  (void) CVS::RENAME (newdir, dotdir);
  /* mv "temp".dir modules.dir */
  (void) CVS::RENAME (newpag, dotpag);
  /* mv "temp".pag modules.pag */
  (void) CVS::RENAME (newdb, dotdb);
  /* mv "temp".db modules.db */
  /* OK – make my day */
  SIG_endCrSect ();
}

sendif    /* !MY::NDBM */

static void
rename_rcsfile (temp, real)
  
700
{
  char *bak;
  struct stat statbuf;
  char rcs;
  
  /* Set "a" bits if set in original. */
  rcs = xmalloc (strlen (real) + sizeof (RCSEX)) + 10;
  (void) sprintf (rcs, "%sz", real, RCSEX);
  if (statbuf.st_mode & 0611) < 0)
  error (0, errno, "warning: cannot stat %s", temp);
  bak = xmalloc (strlen (real) + sizeof (BAKPREFIX) + 10);
  (void) sprintf (bak, "%s", BAKPREFIX, real);
  (void) unlink_file (bak);
  /* rm $loginfo */
(void) CVS_RENAME (real, bak); /* move logininfo #logininfo */
(void) CVS_RENAME (temp, real); /* move "temp" logininfo */
free (bak);
}

const_char *const init[]usage[] = {
    "Usage: %s %s
    <*> specify -h global option for a list of other help options)
    NULL
};

int
init (argc, argv)
int argc;
char *argv1 argvv;
{
    /* Name of CVSROOT directory. */
    char *adm;
    /* Name of this administrative file. */
    char *info;
    /* Name of .v file for this administrative file. */
    char *infov;
    /* Exit status. */
    int err;

    const struct admin_file *fileptr;

    umask (cvsmask);

    if (argc == -1 || argc > 1)
        usage (init_usage);

    ifdef CLIENT_SUPPORT
    if (clientactive)
    {
        startserver ();
        ignoresetup ();
        sendinit (client, command ());
        return (1);
    }
    return (1);
    endif

    sendit /∗ CLIENT_SUPPORT ∗/
    /
    /* Note: we do *not* create parent directories as needed like the
    old cvsvinit.sh script did. Few utilities do that, and a
    non-existent parent directory is as likely to be a typo as something
    which needs to be created. */
    mkdirif_needed (CVSroot_directory);

    adm = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM) + 10);
    strcpy (adm, CVSroot_directory);

    strcat (adm, "/");
    strcat (adm, CVSROOTADM);
    mkdirif_needed (adm);

    /* This is needed because we pass "fileptr->filename" not "info"
    to addrc file below. I think this would be easy to change,
    thus making the need for CVS_CHDIR here, but I haven't looked
    closely (e.g., see wrappers calls within addrc file). */
    if (CVS_CHDIR (adm) < 0)
        error (1, errno, "cannot change to directory": adm);

    /* 80 is long enough for all the administrative file names, plus
    */
    /* and so on. */
    info = xmalloc (strlen (adm) + 80);
    infov = xmalloc (strlen (adm) + 80);
    for (fileptr = filelist; fileptr && fileptr->filename; ++fileptr)
    {
        if (fileptr->contents == NULL)
            continue;
        strcpy (info, adm);
        strcat (info, "/");
        strcat (info, fileptr->filename);
        strncat (info, RCREXT, RCSEXT);
        if (isfile (info, infov))
            /* We will check out this file in the mkmodules step.
            Nothing else is required. */
            continue;
        else
        {
            int retcode;

            if (isfile (info))
            {
                FILE *fp;
                const char *const *p;

                fp = open_file (info, "r");
                for (p = fileptr->contents; p != NULL; ++p)
if (fputs (∗p, fp) < 0)
    error (1, errno, "cannot write ∗a", info);
if (fclose (fp) < 0)
    error (1, errno, "cannot close ∗a", info);
/* The message used to say ∗ of ∗ and fileptr->filename after
   "initial checkin" but I fail to see the point as we know what
   file it is from the name. */
retcode = add_rcs_file (∗"initial checkin", info ∗v,
                        fileptr->filename, ∗"1.1", NULL,
                        NULL, NULL, NULL);
if (retcode != 0)
    /* add_rcs_file already printed an error message. */
    err = 1;
}

strcpy (info, adm);
strcat (info, ∗"/");
strcat (info, CVSROOTADM_HISTORY);
if (!isfile (info))
{
    FILE ∗fp;
    fp = openfile (info, ∗"w");
    if (fclose (fp) < 0)
        error (1, errno, "cannot close ∗a", info);
}
free (info);
free (info ∗v);
mkmodules (adm);
free (adm);
return 0;
A.38 modules.c

#include "cvs.h"
#include "savecwd.h"

/* Defines related to the syntax of the modules file. */
#define CVSMODULEOPTS "+ad:i:lo:e:s:t:u:"
#define CVSMODULESPEC '&'

struct sortrec {
    /* Name of the module, malloc’d. */
    char *modname;
    /* If Status variable is set, this is either def|status or the malloc’d
     * name of the status. If Status is not set, the field is left
     * uninitialized. */
    char *status;
    /* Pointer to a malloc’d array which contains (1) the raw contents
     * of the options and arguments, excluding comments, (2) a '\0',
     * and (3) the storage for the "comment" field. */
    char *rest;
    char *comment;
};

static int sortorder(PROTO(const PTR l, const PTR r));
static void saved_PROTO((char *k, int ks, char *d, int ds));

/* Open the modules file, and die if the CVSROOT environment variable
 * was not set. If the modules file does not exist, that’s fine, and
 * a warning message is displayed and a NULL is returned.
 */
DBM *open_module() {
    char *mfile;
    DBM *retval;
    if (CVSroot_original == NULL)
        error(0, 0, "must set the CVSROOT environment variable");
    error(1, 0, "or specify the '-d' global option");
    mfile = xmalloc(strlen(CVSroot_directory) + sizeof(CVSROOTADM) + sizeof(CVSROOTADM_MODULES) + 20);
    (void) sprintf(mfile, "%s/%s/%s", CVSroot_directory, CVSROOTADM, CVSROOTADM_MODULES);
    retval = dbmopen(mfile, O_RDONLY, 0666);
    free(mfile);
    return retval;
}

/* Close the modules file, if the open succeeded, that is */
void close_module(db) DBM *db;
    { if (db != NULL)
This is the recursive function that processes a module name.

Look up the module using the following scheme:
1) look for mname as a module name
2) look for mname as a directory
3) look for mname as a file
4) take mname up to the first slash and look it up as a module name

This is for checking out only part of a module.

---

```c
modules.c – 417

db glamorous (db);

90 }*/

int
do_module (db, mname, mtype, msg, callback, prog, where, shorten, lspec, runmod, prog, extra, arg)

100 DBM *db;
char *mname;
enum mtype *mtype;
char *msg;
CALLBACKPROC callback;
char *where;
int shorten;
int lspec;
int runmod;
char *extra, arg;

110 {

char *checkin_prog = NULL;
char *checkout_prog = NULL;
char *export_prog = NULL;
char *tag_prog = NULL;
struct saved_cwd cwd;
int cwd_saved = 0;
char *line;
int modargc;
int xmodargc;
char **modargv;
char *xmodargv;
char *evalue;
char *value = NULL;
char *where = NULL;
char *mfile = NULL;
char *spec_opt = NULL;
char *value = NULL;
int alias = 0;

datum key, val;
char *cp;
int c, err = 0;
int nonalias_opt = 0;

ifdef SERVER_SUPPORT
int restore_server_dir = 0;
char *server_dir_to_restore = NULL;

if (trace) {
140 char *buf;

/* We use cvs_outerr, rather than fprintf to stderr, because
this may be called by server code with error_use_protocol
set. */

buf = xmalloc (100
 + strlen (mname)
 + strlen (msg)
 + (where ? strlen (where) : 0)
 + (extra ? strlen (extra) : 0);

sprintf (buf, "%c-> do_module (%s, %s, %s, %s, %s, %s),
(server_active) ? 'S': ' ',
mname, msg, where ? where : "",
extra ? extra : ""),
cvs_outerr (buf, 0);
free (buf);
}
#endif

/* if this is a directory to ignore, add it to that list */
if (mname[0] == '/' && mname[1] != '\0')
{
    ignore_dir (mname+1);
    goto do_module_return;
}

/* strip extra stuff from the module name */
strip_trailing_slashes (mname);

170 /* Look up the module using the following scheme:
 1) look for mname as a module name
 2) look for mname as a directory
 3) look for mname as a file
 4) take mname up to the first slash and look it up as a module name
  (This is for checking out only part of a module.) */
/*
/* look it up as a module name */
```
key.dp = mname;
key.size = strlen (key.dp);
if (db != NULL)
    val = dbm->fetch (db, key);
else
    val.dp = NULL;
if (val.dp != NULL)
{
    /* null terminate the value XXX - is this space ours? */
    val.dp[val.size] = '\0';
}
/* If the line ends in a comment, strip it off */
if ((cp = strchr (val.dp, '\*')) != NULL)
{
    do
        cp-- = '\0';
    while (isspace (*cp));
}
/* Always strip trailing spaces */
cp = strchr (val.dp, '\t');
while (cp > val.dp & & isspace(*--cp))
    *cp = '\t';
value = val.dp;
where = xstrdup (mname);
goto found;
}
else
{
    char *file;
    char *atticfile;
    char *acp;
    int found = 0;

    /* check to see if mname is a directory or file */
    file = xmalloc (strlen (CVSroot_directory) + strlen (mname) + 10);
    (void) sprintf (file, "%s/%s", CVSroot_directory, mname);
    atticfile = xmalloc (strlen (CVSroot_directory) + strlen (mname) + sizeof (CVSATTIC) + sizeof (RCSEXT) + 15);
    if ((acp = strchr (mname, '/')) != NULL)
    {
        *acp = '\t';
        (void) sprintf (atticfile, "%s/%s/%s\t", CVSroot_directory, mname, CVSATTIC, mname, RCSEXT);
        *acp = '/';
        goto found;
    }
    else
    (void) sprintf (atticfile, "%s/%s/%s\t", CVSroot_directory, mname, RCSEXT);

    if (isdir (file))
    {
        value = mname;
        found = 1;
    } else
    {
        (void) strcat (file, RCSEXT);
        if (isdir (file) | | isdir (atticfile))
        {
            /* if mname was a file, we have to split it into "dir file" */
            if ((cp = strchr (mname, '/')) != NULL & & cp != mname)
            {
                char *slashp;
                /* put the ' ' in a copy so we don't mess up the original */
                xvalue = xmalloc (strlen (mname) + 2);
                value = strcpy (xvalue, mname);
                slashp = strchr (value, '/');
                *slashp = ' ';
            }
        } else
        {
            /* the only '/' at the beginning or no '/' at all
             * means the file we are interested in is in CVSROOT
             * itself so the directory should be ' ' */
            if (cp == mname)
            {
                /* drop the leading / if specified */
                xvalue = xmalloc (strlen (mname) + 10);
                value = strcpy (xvalue, " ");
                (void) strcat (xvalue, mname + 1);
            } else
        } else
        {
270  
    { /* otherwise just copy it */
        xvalue = xmalloc(strlen(mname) + 10);
        value = strcpy(xvalue, "", ");
        (void) strcat(xvalue, mname);
    }
    } = 1;
} else{
    /* This initialization suppresses a warning from gcc -Wall.*
    value = NULL;
    }
} free(atticfile);
free(file);
if (isfound) goto found;
}

280  */ look up everything to the first / as a module */
if (mname[0] != '/' && (cp = strchr(mname, '/'))) != NULL) {
    /* Make the slash the new end of the string temporarily */
    *cp = '\0';
    key.dptr = mname;
    key.dsize = strlen(key.dptr);

    /* do the lookup */
    if (db != NULL) val = dbm_fetch(db, key);
    else val.dptr = NULL;

    /* if we found it, clean up the value and life is good */
    if (val.dptr != NULL) {
        char *cp2;
        /* null terminate the value XXX - is this space ours? */
        val.dptr[val.dsize] = '\0';
        /* If the line ends in a comment, strip it off */
        if ((cp2 = strchr(val.dptr, '#')) != NULL) {
            do
                *cp2--; = '\0';
            while (isspace(*cp2));
        }
        value = val.dptr;

        /* mwhere gets just the module name */
        mwhere = xstrdup(mname);
        mfile = cp + 1;
        /* put the / back in mname */
        *cp = '/';
    } goto found;
}
/* put the / back in mname */
*cp = '/';

/* if we got here, we couldn't find it using our search, so give up */
error(0, 0, "cannot find module '%s' - ignored", mname);
err++; goto do_module_return;

340  */
/* At this point, we found what we were looking for in one *
 * of the many different forms. */
/* found:

    /* remember where we start */
    if (savecwd (&cwd))
        errorexit();
    cwd_saved = 1;
    /* copy value to our own string since if we go recursive we'll be *
    really screwed if we do another dbm lookup */
    zvalue = xstrdup(value);
    value = zvalue;

    /* search the value for the special delimiter and save for later */
if ((cp = strchr (value, CVSMODULESPEC)) != NULL)
{
    *cp = '0';  /* null out the special char */
    specopt = cp + 1;  /* save the options for later */
    if (cp != value) /* strip whitespace if necessary */
        while (isspace (*--cp))
            *cp = '0';
    if (cp == value)
    {
        /* we had nothing but special options, so skip any
            parsing and regular stuff entirely */
        /* If there were only special ones though, we must
            make the appropriate directory and cd to it */
        char *dir;
        /* XXX - XXX - MAJOR HACK - DO NOT SHIP - this needs to
            be typeout, but we don't know that here yet */
        if (!runmoduleprog)
            goto out;
        dir = where?
            where: mname;
        /* XXX - think about making null repositories at each dir here
            instead of just at the bottom */
        make directories (dir);
        if (CVSCHDIR (dir) < 0)
            error (0, errno, "cannot chdir to %s", dir);
        specopt = NULL;
        err++;
        goto out;
    }
    if (!isfile (CVSADM))
    {
        char *nullrepos;
        nullrepos = emptydir_name ();
        CreateAdmin (",", dir, nullrepos, (char *) NULL, (char *) NULL, 0, 0);
        if (!noexec)
            {
                FILE *fp;
                fp = open_file (CVSADM_ENTSTAT, "w+");
                if (fclose (fp) == EOF)
                    error (1, errno, "cannot close %s", CVSADM_ENTSTAT);
            }
    }
    if (server_active)
    {
        server_set_entstat (dir, nullrepos);
        endif
        free (nullrepos);
    }
    goto do_special;
}

/* don't do special options only part of a module was specified */
if (!mfile) specopt = NULL;

/* value now contains one of the following:
   * 1) dir
   * 2) dir file
   * 3) the value from modules without any special args
      / args / dir [ file ] [ file ] ... 
      or -a module [ module ] ... */
/* Put the value on a line with XXX prepended for getopt to eat */
line = xmalloc (strlen (value) + 10);
(void) sprintf (line, "%s %s", "XXX", value);
/* turn the line into an argv[] array */
line2argv (&modargc, &modargv, line, /* "t" */);
free (line);
modargc = xmodargc;
modargv = xmodargv;
/* parse the args */
optind = 0;
while ((c = getopt (modargc, modargv, CVSMODULEOPTS)) != -1)
switch (c)
{
    case 'a':
        alias = 1;
        break;
    case 'd':
        nonalias_opt = 1;
        if (mwhere)
            free (mwhere);
        mwhere = xstrdup (optarg);
        break;
    case 'i':
        nonalias_opt = 1;
        if (mwhere)
            free (mwhere);
        mwhere = xstrdup (optarg);
        break;
    case 'l':
        nonalias_opt = 1;
        local_specified = 1;
        break;
    case 'o':
        nonalias_opt = 1;
        if (checkout_prog)
            free (checkout_prog);
        checkout_prog = xstrdup (optarg);
        break;
    case 't':
        nonalias_opt = 1;
        if (tag_prog)
            free (tag_prog);
        tag_prog = xstrdup (optarg);
        break;
    case 'u':
        nonalias_opt = 1;
        if (update_prog)
            free (update_prog);
        update_prog = xstrdup (optarg);
        break;
    case '?':
        error (0, 0,
"modules file has invalid option for key\%s value \%s",
    key.dptr, val.dptr);
        err++;
        goto do_module_return;
}
}
modargc -= optind;
modargv += optind;
if (modargc == 0)
{
    error (0, 0, "modules file missing directory for module \%s", mname);
    ++err;
    goto do_module_return;
}

if (alias && nonalias_opt)
{
    /* The documentation has never said it is legal to specify
     -a along with another option. And I believe that in the past
     CVS has ignored the options other than -a, more or less, in this
     situation. */
    error (0, 0, "
-a cannot be specified in the modules file along with other options");
    ++err;
    goto do_module_return;
}

if (alias)
{
    int i;
    for (i = 0; i < modargc; i++)
    {
        if (strcmp (mname, modargv[i]) == 0)
            error (0, 0,
"module \%s in modules file contains infinite loop", mname);
        else
            err += do_module (db, modargv[i], mtype, msg, callback.proc,
where, shorten, local_specified,
run_module_prog, extra_arg);
    }
```c
  goto do_module; return;
}
if (mfile != NULL && modargc > 1)
{
  error(0, 0, "module '%s' is a request for a file in a module which is not a directory", mname);
  ++err;
  goto do_module; return;
}
/* otherwise, process this module */
err += callback_proc(&modargc, modargv, where, mwhere, mfile, shorten,
  local_specified, mname, msg);
free_names(&xmodargc, xmodargv);
/* if there were special include args, process them now */
do_special:
560 /* blow off special options if -l was specified */
if (local_specified)
  spec_opt = NULL;

ifdef SERVER_SUPPORT
  /* We want to check out into the directory named by the module.
  So we set a global variable which tells the server to glom that
directory name onto the front. A cleaner approach would be some
way of passing it down to the recursive call, through the
callback_proc, to start recursion, and then into the update_dir
in the struct file_info. That way the "Updating foo" message could
print the actual directory we are checking out into.

  For local CVS, this is handled by the chdir call above
  (directly or via the callback Proc). */
  if (server_active && spec_opt != NULL)
  {
    char *change_to = (char *)local_specified;

    change_to = where ? where : (mwhere ? mwhere : mname);
    server_dir_to_restore = server_dir;
    restore_server_dir = 1;
    server_dir = xmalloc((server_dir_to_restore != NULL
      ? strlen(server_dir_to_restore) + 0
      : strlen(change_to) + 5);
    server_dir[0] = '\0';
    if (server_dir_to_restore != NULL)
      {
        strcat(server_dir, server_dir_to_restore);
        strcat(server_dir, "/");
      }
    strcat(server_dir, change_to);
  }
#endif

while (spec_opt != NULL)
{
  char *next_opt;
  cp = strchr(spec_opt, CVSMODULE_SPEC);
  if (cp != NULL)
  {
    /* save the beginning of the next arg */
    next_opt = cp + 1;
    /* strip whitespace off the end */
    do
      cp = '\0';
      while (isspace(*++cp));
    }else
      next_opt = NULL;
  /* strip whitespace from front */
  while (isspace(*spec_opt))
    spec_opt++; +
  if (*spec_opt == '\0')
    error(0, 0, "Mal-formed %c option for module %s - ignored",
      CVSMODULE_SPEC, mname);
  else
    err += do_module(db, spec_opt, mtype, msg, callback_proc,
      (char *)NULL, 0, local_specified,
      run_module_prog, extra_arg);
  spec_opt = next_opt;
```

.SERVER_SUPPORT
if (server->active & & restore && server->dir)
{
    free (server->dir);
    server->dir = server->dir_to_restore;
}
#endif

if (err == 0 & & noexec & & m_type == CHECKOUT & & server->expanding)
{
    if (checkin_program != NULL)
    {
        server->prog (where ? where : mname, checkin_program, PROG->CHECKIN);
        server->prog (where ? where : mname, update_program, PROG->UPDATE);
    }
    else
    {  
      server->prog (where ? where : mname, run_module_program);
    }
}
#endif

FILE *fp;

if (checkin_program != NULL)
{
    fp = open_file (CVSADM->CIPROG, "r");
    (void) fprintf (fp, "\"%s\"", checkin_program);
    if (fclose (fp) == EOF)
      error (1, errno, "cannot close '%s'", CVSADM->CIPROG);
}
#endif

if (update_program != NULL)
{
    fp = open_file (CVSADM->UPROG, "r");
    (void) fprintf (fp, "\"%s\"", update_program);
    if (fclose (fp) == EOF)
      error (1, errno, "cannot close '%s'", CVSADM->UPROG);
}
#endif

/* cd back to where we started */
if (restore_cwd & & cword, NULL)
  error_exit ();
free_cwd & & cword;
cwd_saved = 0;

/* run checkout or tag prog if appropriate */
if (err == 0 & & run_module_program)
{
    if ((m_type == TAG & & tag_program != NULL) | |
        (m_type == CHECKOUT & & checkout_program != NULL) | |
        (m_type == EXPORT & & export_program != NULL))
    {
      /*
       * If a relative pathname is specified as the checkout, tag
       * or export proc, try to tack on the current "where" value.
       * If we can't find a matching program, just punt and use
       * whatever is specified in the modules file.
       */
      char *real_program = NULL;
      char *prog = (m_type == TAG ? tag_program :
                    (m_type == CHECKOUT ? checkout_program : export_program));
      char *real(where = (where != NULL ? where : mwhere);
      char *expanded_path;
      if ((*prog != '/') & & (*prog != '.'))
      {  
        real_program = xmalloc (strlen (real(where) + strlen (prog) + 10);
        (void) sprintf (real(where), "\%s/\%s", real(where), prog);
        if (isfile (real(where))
          prog = real(where);
      }
      /* XXX can we determine the line number for this entry ??? */
      if (expanded_path != NULL)
      {  
        run_setup (expanded_path);
        run_arg (real(where);
      }
      if (extra_arg)
        run_arg (extra_arg);
      if (!quiet)
      {  
        cvs_output (program_name, 0);
        cvs_output (" ", 1);
        cvs_output (command_name, 0);,
```c
720    cvs_output (" Executing ", 0);
    run_print (stdout);
    cvs_output ("\n", 0);
    err += run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL);
    free (expanded_path);
    free (real_prog);
}    free (mwhere);
    free (checkin_prog);
    free (checkout_prog);
    free (export_prog);
    free (tag_prog);
    free (update_prog);
    free (cwd_saved);
    free (xvalue);
    free (zvalue);
    return (err);
}

730    do_module (return:
        /* clean up */
        if (mwhere) free (mwhere);
        if (checkin_prog) free (checkin_prog);
        if (checkout_prog) free (checkout_prog);
        if (export_prog) free (export_prog);
        if (tag_prog) free (tag_prog);
        if (update_prog) free (update_prog);
        if (cwd_saved) free (cwd_saved);
        if (xvalue != NULL) free (xvalue);
        if (zvalue != NULL) free (zvalue);
        return (err);
    }

    /* - Read all the records from the modules database into an array.     
      - Sort the array depending on what format is desired.              
      - Print the array in the format desired.                           

    Currently, there are only two "desires":                              

    760    1. Sort by module name and format the whole entry including switches, 
      files and the comment field: (Including aliases)                  
      
      modulename -s switches, one per line, even if                      
      - it has many switches.                                            
      Directories and files involved, formatted                          
      to cover multiple lines if necessary.                             
      # Comment, also formatted to cover multiple                       
      # lines if necessary.                                             

    770    2. Sort by status field and print: (- not including aliases)    
      modulename STATUS Directories and files involved, formatted        
      to cover multiple lines if necessary.                             
      # Comment, also formatted to cover multiple                       
      # lines if necessary.                                             

    */

    static struct sortrec *head;

    static int s_max = 0;    /* Number of elements allocated */
    static int s_count = 0;  /* Number of elements used */

    static int Status;       /* Nonzero if the user is interested in status 
                              information as well as module name */

    static char def_status[] = "NONE";

    /* Sort routine for qsort:                                          
      - If we want the "Status" field to be sorted, check it first.     
      - Then compare the "module name" fields. Since they are unique, we don't 
        have to look further.                                          

    */
    static int sort_order (l, t)
        const PTR l;
        const PTR t;
    {
        int i;

        const struct sortrec *left = (const struct sortrec *) l;
        const struct sortrec *right = (const struct sortrec *) t;

        if (Status)
            /* If Sort by status field, compare them. */
            if (strcmp ((left->status, right->status)) != 0)
                return (i);
            } else
        return (sort_order (l, t), 0);
    } /* sort_order */
```
return (strcmp (left->modname, right->modname));

static void
saveval (k, ks, d, da)
char *k;
int ks;
char *d;
int da;
{
char *cp, *cp2;
struct sortrec *srec;
if (Status && d == '-' && *(d + 1) == 'a')
    return; /* We want "cos co -s" and it is an alias */
if (scount == smax)
    { 
    s_max += 64;
    s_head = (struct sortrec *) xrealloc ((char *) s_head, s_max + sizeof (*s_head));
    }
void *srec = &s_head[scount];
s_rec->modname = cp = xmalloc (ks + 1);
*(cp + ks) = '\0';
s_rec->rest = cp2 = xmalloc (ds + 1);
if (isspace (*cp))
    cp++;
/* Turn <spaces> into one ' ' – makes the rest of this routine simpler */
while (*cp)
    {
    if (isspace (*cp))
        {
        *cp2++ = ' ';
        while (isspace (*cp))
            cp++;
        } else
        {
        *cp2++ = *cp++;
        }
    *cp2 = '\0';
    /* Look for the "-s statusvalue" text */
    if (Status)
        {
        s_rec->status = defstatus;
        for (cp = s_rec->rest; (cp2 = strchr (cp, ' ')) != NULL; cp = ++cp2)
            { 
            if (*cp2 + 1) == 's' && *(cp2 + 2) == ' ' )
                { 
                char *status_start;
                cp2 += 3;
                status_start = cp2;
                while (*cp2 != ' ' && *cp2 != '\0')
                    cp2++;
                s_rec->status = xmalloc (cp2 - status_start + 1);
                strncpy (s_rec->status, status_start, cp2 - status_start);
                s_rec->status[cp2 - status_start] = '\0';
                cp = cp2;
                break;
                }
            }
        else
            cp = s_rec->rest;
        }
    else
        if (s_rec->comment)
            { 
            /* Find comment field, clean up on all three sides & compress blanks */
            if ((cp2 = cp = strchr (cp, ' ')) != NULL)
                { 
                if (*cp++ == ' ')
                    cp2 = '\0';
                if (*cp++ == ' ')
                    cp2 += 1;
                } else
                    s_rec->comment = cp;
            }
        else
            s_rec->comment = **;
        scount++;
    }
    /* Print out the module database as we know it. If STATUS is non-zero, print out status information for each module. */
void
cat_module (status)
{
    DBM *db;
    datum key, val;
    int i, c, wid, argc, cols = 80, indent, fill;
    int moduleargc;
    struct sortrec *s, *h;
    char *cp, *cp2, *argv;
    char *moduleargv;

    Status = status;
    /* Read the whole modules file into allocated records */
    if (! (db = open_module ()))
        error (1, 0, "failed to open the modules file");
    for (key = dbm_firstkey (db); key.dkptr != NULL; key = dbm_nextkey (db))
    {
        val = dbm_fetch (db, key);
        if (val.dkptr != NULL)
            save_d (key.dkptr, key.dsize, val.dkptr, val.dsize);
    }
    close_module (db);
    /* Sort the list as requested */
    qsort ((PTR *) s, s_count, sizeof (struct sortrec), sort_order);
    */
    /* Run through the sorted array and format the entries */
    indent = space for modulename + space for status field
    indent = 12 + (status + 12);
    fill = cols - (indent + 2);
    for (s_h = s_head, i = 0; i < s_count; i++, s_h++)
    {
        char *line;
        /* Print module name (and status, if wanted) */
        line = xmalloc (strlen (s_h->modname) + 15);
        sprintf (line, "%12s", s_h->modname);
        cvsm_output (line, 0);
        free (line);
        if (status)
        {
            line = xmalloc (strlen (s_h->status) + 15);
            sprintf (line, "%-11s", s_h->status);
            cvsm_output (line, 0);
            free (line);
        }
        line = xmalloc (strlen (s_h->modname) + strlen (s_h->rest) + 15);
        /* Parse module file entry as command line and print options */
        (void) sprintf (line, "%-5s", s_h->modname, s_h->rest);
        line2argv (&moduleargc, &moduleargv, line, "\t");
        free (line);
        argc = moduleargc;
        argv = moduleargv;
    }
    optind = 0;
    wid = 0;
    while ((c = getopt (argc, argv, CVSMODULE_OPTS)) != -1)
    {
        if (status)
        {
            if (c == 'a' || c == 'l')
            {
                char buf[8];
                sprintf (buf, "%.c", c);
                cvsm_output (buf, 0);
                wid += 3; /* Could just set it to 3 */
            }
            else
            {
                char buf[10];
                if (strlen (optarg) + 4 + wid > (unsigned) fill)
                {
                    int i;
                    cvsm_output ("\n", 1);
                    for (j = 0; j < indent; ++j)
                        cvsm_output (' ', 1);
                    wid = 0;
                    }
                sprintf (buf, "%.c", c);
                cvsm_output (buf, 0);
            }
        }
cvs_output(optarg, 0);
wid += strlen(optarg) + 4;
}
}
argc -= optind;
argv += optind;

/* Format and Print all the files and directories */
for (; argc--; argv++)
{
    if (strlen(*argv) + wid > (unsigned) fill)
    {
        for (j = 0; j < indent; ++j)
            cvs_output (*" ", 1);
        cvs_output (*" ", 1);
        wid = 0;
    }
    cvs_output (*" ", 1);
    cvs_output (*argv, 0);
    wid += strlen(*argv) + 1;
}
    cvs_output (*" ", 1);

/* Format the comment field – save ()], compressed spaces */
for (cp2 = cp = sh->comment; *cp; cp2 = cp)
{
    int j;
    for (j = 0; j < indent; ++j)
        cvs_output (*" ", 1);
    if (strlen(cp2) < (unsigned) (fill - 2))
    {
        cvs_output (cp2, 0);
        cvs_output (*" ", 1);
        break;
    }
    cp += fill - 2;
}
    while (*cp != ' ' && cp > cp2)
        cp--;
    if (cp == cp2)
    {
        cvs_output (cp2, 0);
        cvs_output (*" ", 1);
        break;
    }
    *cp++ = '\0';
    cvs_output (cp2, 0);
    cvs_output (*" ", 1);
}
free_names(&moduleargc, moduleargv);
/* FIXME-leak: here is where we would free sh->modname, sh->rest,
   and if applicable, sh->status. Not exactly a memory leak,
in the sense that we are about to exit(), but may be worth
noting if we ever do a multithreaded server or something of
the sort. */

/* FIXME-leak: as above, here is where we would free sh->head. */
A.39 myndbm.c

/*
 * Copyright (c) 1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * A simple ndbm-emulator for CVS. It parses a text file of the format:
 * key value
 * at open time, and loads the entire file into memory. As such, it is
 * probably only good for fairly small modules files. Ours is about 30K in
 * size, and this code works fine.
 */

#include <cassert.h>
#include "cvs.h"
#include "getline.h"

ifdef MY_NDBM

static void myndbm_load_file PROTO ((FILE *, List *));

/* Returns NULL on error in which case errno has been set to indicate
 the error. Can also call error() itself. */
/* ARGSUSED */

DBM *
myndbm_open (file, flags, mode)
char *file;
int flags;
int mode;
{
FILE *fp;
DBM *db;

fp = CVS_FOPEN (file, FOPEN_BINARY_READ);
if (fp == NULL & & ((errno & (flags & O_CREAT)))
    return ((DBM *) 0);

db = (DBM *) xmalloc (sizeof (*db));
db->dbm_list = getlist ();
db->modified = 0;
db->name = xstrdup (file);

if (fp != NULL)
{
    myndbm_load_file (fp, db->dbm_list);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close file", file);
}
return (db);
}

static int write_item PROTO ((Node *, void *));

static int
write_item (node, data)
Node *node;
void *data;
{
FILE *fp = (FILE *)data;
puts (node->key, fp);
puts (" ", fp);
puts (node->data, fp);
puts ("\012", fp);
return 0;
}

void
myndbm_close (db)
DBM *db;
{
if (db->modified)
{
    FILE *fp;
    fp = CVS_FOPEN (db->name, FOPEN_BINARY_WRITE);
    if (fp == NULL)
        error (1, errno, "cannot write %s", db->name);
    walklist (db->dbm_list, write_item, (void *)fp);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", db->name);
}
free (db->name);
dellist (&db->dbm_list);
free ((char *) db);
}

datum

mydbm_fetch (db, key)
         DBM *db;
         datum key;
         {
             Node *p;
             char *s;
             datum val;
             /* make sure it's null-terminated */
             s = xmaxalloc (key.dsize + 1);
             (void) strncpy (s, key.dptr, key.dsize);
             [key.dsize] = '\0';
             p = findnode (db->dbm_list, s);
             if (p)
                 { val.dptr = p->data;
                   val.dsize = strlen (p->data);
                 }
             else
                 { val.dptr = (char *) NULL;
                   val.dsize = 0;
                 }
             free (s);
             return (val);
         }

datum mydbm_firstkey (db)
         DBM *db;
         {
             Node *head, *p;
             datum key;
             head = db->dbm_list->list;
             p = head->next;
             if (p != head)
                 { key.dptr = p->key;
                   key.dsize = strlen (p->key);
                 }
             else
                 { key.dptr = (char *) NULL;
                   key.dsize = 0;
                 }
             db->dbm_next = p->next;
             return (key);
         }

datum mydbm_nextkey (db)
         DBM *db;
         {
             Node *head, *p;
             datum key;
             head = db->dbm_list->list;
             p = db->dbm_next;
             if (p != head)
                 { key.dptr = p->key;
                   key.dsize = strlen (p->key);
                 }
             else
                 { key.dptr = (char *) NULL;
                   key.dsize = 0;
                 }
             db->dbm_next = p->next;
             return (key);
         }

/* Note: only updates the in-memory copy, which is written out at 
   mydbm_close time. Note: Also differs from DBM in that on duplication,
   it gives a warning, rather than either DBM_INSERT or DBM_REPLACE 
   behavior. */
int mydbm_store (db, key, value, flags)
         DBM *db;
         datum key;
         datum value;
         int flags;
         {
             Node *node;
             node = getnode ();
             node->type = NDBMNODE;
         }
node->key = xmalloc (key.size + 1);
    strncpy (node->key, key.dptr, key.size);
    node->key[key.size] = '\0';

    node->data = xmalloc (value.size + 1);
    strncpy (node->data, value.dptr, value.size);
    node->data[value.size] = '\0';

    db->modified = 1;
    if (addnode (db->dbmlist, node) == -1)
    {
        error (0, 0, "attempt to insert duplicate key '%s'", node->key);
        freenode (node);
        return 0;
    }
    return 0;
}

static void
mydbm_load_file (fp, list)
    FILE *fp;
    List *list;
{
    char *line = NULL;
    size_t line_size;
    char *value;
    size_t value_allocated;
    char *vp, *vp;
    int cont;
    int line_length;

    value_allocated = 1;
    value = xmalloc (value_allocated);
    cont = 0;
    while ((line_length = getstr (&line, &line_size, fp, '\012', 0)) > 0)
    {
        if (line_length > 0 && line[line_length - 1] == '\012')
            /* Strip the newline. */
            --line_length;
        if (line_length > 0 && line[line_length - 1] == '\015')
            /* If the file (e.g. modules) was written on an NT box, it will
             contain CRLF at the end of lines. Strip them (we can't do
             this by opening the file in text mode because we might be
             running on unix). */
            --line_length;
        line[line_length] = '\0';
        /* Add the line to the value, at the end of this is a continuation
         * line, otherwise at the beginning, but only after any trailing
         * backslash is removed. */
        if (cont)
            value[0] = '\0';
        /*
         * See if the line we read is a continuation line, and strip the
         * backslash if so.
         *
         * if (line_length > 0)
         *     cp = &line[line_length - 1];
         * else
         *     cp = line;
         * if (*cp == '\\')
         *     cont = 1;
         *     *cp = '\0';
         * } else
         *     expand_string (&value,
         *     &value_allocated,
         *     strlen (value) + line_length + 5);
         * streal (value, line);
         * if (value[0] == '\#')
         *     continue; /* comment line */
         *     vp = value;
         * while (*vp && ispance (*vp))
         *     vp++;
         * if (*vp == '\0')
        */
continue;  // empty line */

/* If this was not a continuation line, add the entry to the database */
if (!cont) {
    Node *p = getnode();
    char *kp;
    kp = vp;
    while (*vp && isspace (*vp))
        vp++;
    /* NULL terminate the key */
    p->type = NDBMNODE;
    p->key = xstrdup (kp);
    while (*vp && isspace (*vp))
        vp++;
    /* skip whitespace to value */
    if (*vp == ' ')
        { /* notice: NULL value for key 'x' */
            error (0, 0, "warning: NULL value for key 'x'", p->key);
            freenode (p);
            continue;
        }
    p->data = xstrdup (vp);
    if (addnode (list, p) == -1)
        { /* notice: duplicate key found for 'x' */
            error (0, 0, "duplicate key found for 'x'", p->key);
            freenode (p);
        }
}

if (line_length < 0 || !feof (fp))
    /* FIXME: should give the name of the file. */
    free (line);
    free (value);
}
A.40 myndbm.h

```c
#ifdef MY_NDBM
#define DBLKSIZ 4096
typedef struct {
    List *dbm_list;  /* cached database */
    Node *dbm_next;  /* next key to return for nextkey() */
} DBM;

typedef struct {
    char *dbptr;
    int dsiz;
} datum;

/* So as not to conflict with other dbm_open, etc., routines that may
* be included by someone's libc, all of my emulation routines are prefixed
* by "my" and we define the "standard" ones to be "my" ones here. */
#define dbm_open mydbm_open
#define dbm_close mydbm_close
#define dbm_fetch mydbm_fetch
#define dbm_firstkey mydbm_firstkey
#define dbm_nextkey mydbm_nextkey
#define dbm_store mydbm_store
#define DBM_INSERT 0
#define DBM_REPLACE 1

void mydbm_open PROTO((char *file, int flags, int mode));
void mydbm_close PROTO((DBM *db));

#define MY_NDBM */
```
A.41  no_diff.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"

int

No_Difference (finfo, vers)
struct file_info *finfo;
struct TS vers;
{
    Node *p;
    int ret;
    char *ts, *options;
    int retcode = 0;
    char *tocvsPath;

    /* If ts_user is "Is-modified", we can only conclude the files are
different (since we don't have the file's contents). */
    if (vers->ts_user != NULL
        && (strcmp (vers->ts_user, "Is-modified") == 0)
        return −1;
    if (vers−>srcfile || vers−>srcfile−>path)
        return (−1);
    /* different since we couldn't tell */

# ifdef PRESERVE_PERMISSIONS_SUPPORT
/* If special files are in use, then any mismatch of file metadata
information also means that the files should be considered different. */
if (preserve_wperms && special_file_mismatch (finfo, vers->ts_user, NULL))
    return 1;
#endif

    options = xstrdup (vers−>entdata−>options);
    if (vers−>entdata && vers−>entdata−>options)
        options = xstrdup (vers−>entdata−>options);
    else
        options = xstrdup ("*");

    tocvsPath = wrap tocvs_process file (finfo−>file);
    retcode = RCS_imp_file (vers−>srcfile, vers−>ts_user, options,
                            tocvsPath == NULL ? info−>file : tocvsPath);
    if (retcode == 0)
        { /* no difference was found, so fix the entries file */
            ts = time stamp (finfo−>file);
            Register (finfo−>entries, info−>file,
                      vers−>ts_user ? vers−>ts_user : vers−>srcfile, ts,
                      options, vers−>tag, vers−>date, (char *) 0, CVSRoot, directory, info−>repository);
        }
#endif

# ifdef SERVER_SUPPORT
    if (server_active)
        { /* We need to update the entries line on the client side. */
            server_update_entries
                (finfo−>file, info−>update_dir, info−>repository, SERVER_UPDATED);
        }
#endif

    free (ts);

    /* update the entdata pointer in the vers.js structure */
    p = findnode (finfo−>entries, info−>file);
    vers−>entdata = (Entnode *) p−>data;
    ret = 0;
}

else
    ret = 1;  /* files were really different */

if (tocvsPath)
    { /* Need to call unlink myself because the noexec variable
        * has been set to 1. */
        if (trace)
            (void) fprintf (stderr, "#x-> unlink (2a)\n",
# ifdef SERVER_SUPPORT
                (server_active) ? '8' : ' ',
else
endif

tocvsPath);

if ( CVS_UNLINK (tocvsPath) < 0)
    error (0, errno, "could not remove %s", tocvsPath);
}

free (options);
return (ret);
}
A.42  parseinfo.c

/*
 * Copyright (c) 1992, Brian Birinder and Jeff Polk
 * Copyright (c) 1989-1992, Brian Birinder
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"
#include "getline.h"
#include <assert.h>

/*
 * Parse the INFOFILE file for the specified REPOSITORY. Invoke CALLPROC for
 * the first line in the file that matches the REPOSITORY, or if ALL != 0, any lines
 * matching "ALL", or if no lines match, the last line matching "DEFAULT".
 * Return 0 for success, -1 if there was not an INFOFILE, and >0 for failure.
 */

int Parse_Info (infofile, repository, callproc, all)
char  *infopath;
char  *repository;
CALLPROC callproc;
int all;
{
  int err = 0;
  FILE  *fpinfo;
  char  *infopath;
  char  *repository;
  CALLPROC callproc;
  int all;
  if (CVSroot_original == NULL)
  {
    /* XXX - should be error maybe? */
    error (0, 0, "CVSROOT variable not set");
    return (1);
  }

  /* find the info file and open it */
  infopath = xmalloc (strlen (CVSroot_directory) + strlen (infopath) + sizeof (CVSROOTADM) + 10);
  fp_info = CVS_FOPEN (infopath, "r");
  if (fp_info == NULL)
  {
    /* If no file, don't do anything special. */
    if (errno == EPERM)
      error (0, errno, "cannot open INFOPATH", infopath);
    free (infopath);
    return 0;
  }

  /* strip off the CVSROOT if repository was absolute */
  repository = ShortRepository (repository);
  if (trace)
    (void) fprintf (stderr, " -> ParseInfo(", 1, &repository);%s"\n", infofile, repository, all ? "ALL" : "not ALL");

  /* search the info file for lines that match */
  line_number = 0;
  while (getline (&line, &line_allocated, fp_info) >= 0)
  {
    line_number++;
    /* skip lines starting with # */
    if (line[0] == '#')
      continue;
    /* skip whitespace at beginning of line */
    while (isspace (*cp)) cp++;
    /* if *cp is null, the whole line was blank */
    if (*cp == '\n' )
      continue;
    /* the regular expression is everything up to the first space */
    for (exp = cp; *cp & isnospace (*cp); cp++)
  
    /* the regular expression is everything up to the first space */
    for (exp = cp; *cp & isnospace (*cp); cp++)
  ;
if (exp != NULL)
    free (expanded_value);
expanded_value = expand_path (value, infopath, line_number);
if (expanded_value)
    continue;

/* At this point, exp points to the regular expression, and value
   points to the value to call the callback routine with. Evaluate
   the regular expression against srepos and callback with the value
   if it matches. */
/* save the default value so we have it later if we need it */
if (strcmp (exp, "DEFAULT") == 0)
    /* Is it OK to silently ignore all but the last DEFAULT
       expression? */
if (default_value != NULL)
    free (default_value);
default_value = xstrdup (expanded_value);
continue;

/* For a regular expression of "ALL", do the callback always We may
   execute lots of ALL callbacks in addition to *one* regular matching
   callback or default */
if (strcmp (exp, "ALL") == 0)
    {  
    if (all)
        err += callproc (repository, expanded_value);
    else
        error(0, 0, "Keyword 'ALL' is ignored at line %d in %s file",
              line_number, infopath);
        continue;
    }
if (callbackDone)
    /* only first matching, plus "ALL"'s */
    continue;

/* see if the repository matched this regular expression */
if ((repos_err = match (exp)) != NULL)
    {  
    error(0, 0, "bad regular expression at line %d file %s",
          line_number, infopath, repos_err);
    continue;
    }
if (repos_exec (srepos) == 0)
    continue; /* no match */
/* std did, so do the callback and note that we did one */
err += callproc (repository, expanded_value);
callbackDone = 1;
    }
if (error (infopath))
    error (0, errno, "cannot read %s", infopath);
if (exec_info < 0)
    error (0, errno, "cannot close %s", infopath);

/* if we fell through and didn't callback at all, do the default */
if (callbackDone == 0 && default_value != NULL)
    err += callproc (repository, default_value);
/* free up space if necessary */
if (default_value != NULL) free (default_value);
if (expanded_value != NULL) free (expanded_value);
free (infopath);
if (line != NULL) free (line);
return (err);
}

/* Parse the CVS config file. The syntax right now is a bit ad hoc
but tries to draw on the best or more common features of the other
*info files and various unix (or non-unix) config file syntaxes.
Lines starting with # are comments. Settings are lines of the form
KEYWORD=VALUE. There is currently no way to have a multi-line
VALUE (would be nice if there was, probably).
CVSROOT is the $CVSROOT directory (CVSroot_directory might not be
set yet).

Returns 0 for success, negative value for failure. Call
error(0, ...) on errors in addition to the return value. */

int
parse_config (cvsroot)
char *cvsroot;
{
char *infopath;
FILE *fp_info;
char *line = NULL;

size_t line_allocated = 0;
size_t len;
char *p;

/* FIXME-reentrancy: If we do a multi-threaded server, this would need
to go to the per-connection data structures. */
static int parsed = 0;

/* Authentication code and serve_root might both want to call us.
Let this happen smoothly. */
if (parsed)
return 0;
parsed = 1;

infopath = malloc (strlen (cvsroot)
+ sizeof (CVSROOTADM_CONFIG)
+ sizeof (CVSROOTADM)
+ 10);
if (infopath == NULL)
{
error (0, 0, "out of memory; cannot allocate infopath");
go to error_return;
}

strcpy (infopath, cvsroot);
strcat (infopath, "/");
strcat (infopath, CVSROOTADM);
strcat (infopath, "/");
strcat (infopath, CVSROOTADM_CONFIG);

fp_info = CVS_FOPEN (infopath, "r");
if (fp_info == NULL)
{
/* If no file, don't do anything special. */
if (! existence (errno))
{
/* Just a warning message; doesn't affect return
value, currently at least. */
error (0, errno, "cannot open %s", infopath);
}
free (infopath);
return 0;
}

while (getline (&line, &line_allocated, fp_info) >= 0)
{
/* Skip comments. */
if (line[0] == '
') continue;

/* At least for the moment we don't skip whitespace at the start
of the line. Too picky? Maybe. But being insufficiently
picky leads to all sorts of confusion, and it is a lot easier
to start out picky and relax it than the other way around.

Is there any kind of written standard for the syntax of this
sort of config file? Anywhere in POSIX for example (I guess
makefiles are sort of close)? Red Hat Linux has a bunch of
these too (with some GUI tools which edit them)..."
Along the same lines, we might want a table of keywords, with various types (boolean, string, &c.), as a mechanism for making sure the syntax is consistent. Any good examples to follow there (Apache)? */

/* Strip the training newline. There will be one unless we read a partial line without a newline, and then got end of file (or error?). */

len = strlen(line) - 1;
if (line[len] == '\n')
    line[len] = '\0';
/* Skip blank lines. */
if (line[0] == '\0')
    continue;
/* The first '=' separates keyword from value. */
p = strchr(line, '=');
if (p == NULL)
    {
    /* Probably should be printing line number. */
    error (0, 0, "syntax error in %s: line '%s' is missing '='", infopath, line);
    goto error
    return;
    }
p++ = '\0';
if (strcmp(line, "RCSBIN") == 0)
    {
    /* This option used to specify the directory for RCS executables. But since we don't run them any more, this is a noop. Silently ignore it so that a repository can work with either new or old CVS. */
    ;
    }
else if (strcmp(line, "SystemAuth") == 0)
    {
    #ifdef AUTH_SERVER_SUPPORT
    system_auth = 0;
    #else
    /* Still parse the syntax but ignore the option. That way the same config file can be used for local and server. */
    ;
    #endif
    #ifdef AUTH_SERVER_SUPPORT
    system_auth = 1;
    #else
    warning: this CVS does not support SystemAuth;
    #endif
    }
else if (strcmp(line, "PreservePermissions") == 0)
    {
    if (strcmp(p, "no") == 0)
        preserve_perms = 0;
    else if (strcmp(p, "yes") == 0)
        
    #ifdef PRESERVE_PERMISSIONS_SUPPORT
    preserve_perms = 1;
    #else
    warning: this CVS does not support PreservePermissions;
    #endif
    }
    else
    {
        error (0, 0, "unrecognized value '"%s' for PreservePermissions", p);
        goto error;
    }
    }
else if (strcmp(line, "TopLevelAdmin") == 0)
    {
    if (strcmp(p, "no") == 0)
        top_level_admin = 0;
    else if (strcmp(p, "yes") == 0)
        top_level_admin = 1;
    else
        error (0, 0, "unrecognized value '"%s' for TopLevelAdmin", p);
        goto error;
    }
*/ We may be dealing with a keyword which was added in a subsequent version of CVS. In that case it is a good idea to complain, as (1) the keyword might enable a behavior like alternate locking behavior, in which it is dangerous and hard to detect if some CVS's have it one way and others have it the other way, (2) in general, having us not do what the user had in mind when they put in the keyword violates the principle of least surprise. Note that one corollary is adding new keywords to your CVSSROOT/config file is not particularly recommended unless you are planning on using the new features. */

error(0, 0, "In: unrecognized keyword 'Is', infopath, line);  goto error;  return;

360
}
else  
{
*/

370

380

390

400  }
A.43 patch.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Patch
 * Create a Larry Wall format "patch" file between a previous release and the
 * current head of a module, or between two releases. Can specify the
 * release as either a date or a revision number.
 */

#include "cvs.h"
#include "getline.h"

datatype patch gunshot PROTO((void));
datatype Dtype patch_dirproc PROTO ((void *callerdat, char *dir,
    char *repos, char *update_dir, List *entries));
datatype int patch_fileproc PROTO ((void *callerdat, struct file_info *info));
datatype int patch_proc PROTO (int *parge, char **argv, char *xwhere,
    char *mwhere, char *emfile, int shortc, int local_specified, char *mnname, char *msg));

datatype int forced_tag_match = 1;
datatype int patch_short = 0;
datatype int top_twodiffs = 0;
datatype int local = 0;
datatype int options = NULL;
datatype char *rev1 = NULL;
datatype int rev1_validated = 0;
datatype char *rev2 = NULL;
datatype int rev2_validated = 0;
datatype char *date1 = NULL;
datatype char *date2 = NULL;
datatype char *tmphfile1 = NULL;
datatype char *tmphfile2 = NULL;
datatype int tmphfile3 = NULL;
datatype int unidiff = 0;

static const char *const patch_usage[] = {
    "Usage: %s [ -ffil ] [ -c|-u ] [ -s|-t ] [ -V "vers" ] [ -r rev1 -0 date ] [ -r rev2 -0 date ] modules...",
    "-r rev1-0 date [ -r rev2 -0 date ] modules...",
    "-t|-u Force a head revision match if tag/date not found.",
    "-t Local directory only, not recursive.",
    "-t Process directories recursively.",
    "-c|-u Unidiff format.",
    "-s|-t command line arguments for diff1 --arg1 arg2 ...",
    "-V vers Use RCS Version "vers" for keyword expansion.",
    "Specify the --help global option for a list of other help options.",
    "-q or -Q must be specified before "%s"",
    "if (server_active)"
};

int patch (argc, argv)
int argc;
char **argv;
{
    register int i;
    int c;
    int err = 0;
    DBM *db;

    if (argc == -1)
        usage (patch_usage);

    optind = 0;
    while ((c = getopt(argc, argv, "*V:k:cuftsqQExr:")) != -1)
    {
        switch (c)
        {
            case 'Q':
            case 'q':
                if (server_support)
                    error (1, 0,
                        "-q or -Q must be specified before ""\%s"",
                        command_name);
                /* else */
                break;
            case 'c':
                /* Else */
            default:
                usage (patch_usage);
            break;
        }
    }
    return (optind);
}

endif

*/ The CVS 1.5 client sends these options (in addition to
Global option requests), so we must ignore them. */

if (server_active)
break;

case 'f':
    force_tag_match = 0;
    break;

case 'l':
    local = 1;
    break;

case 'R':
    local = 0;
    break;

case 't':
    toptwo_diffs = 1;
    break;

case 's':
    patch_short = 1;
    break;

case 'D':
    if (rev2 != NULL || date2 != NULL)
        error (1, 0, "no more than two revisions/dates can be specified");
    if (rev1 != NULL || date1 != NULL)
    
        date2 = Make_Date (optarg);
    else
        date1 = Make_Date (optarg);
    break;

case 'r':
    if (rev2 != NULL || date2 != NULL)
        error (1, 0, "no more than two revisions/dates can be specified");
    if (rev1 != NULL || date1 != NULL)
        rev2 = optarg;
    else
        rev1 = optarg;
    break;

case 'k':
    if (options)
        free (options);
    options = RCS_check_flag (optarg);
    break;

case 'V':
    /* This option is pretty seriously broken:
      1. It is not clear what it does (does it change keyword
         expansion behavior? If so, how? Or does it have
         something to do with what version of RCS we are using?
         Or the format we write RCS files in?).
      2. Because both it and -k use the options variable,
         specifying both -V and -k doesn't work.
      3. At least as of CVS 1.9, it doesn't work (failed
         assertion in RCS_checkout where it asserts that options
         starts with -k). Few people seem to be complaining.
     In the future (perhaps the near future), I have in mind
     removing it entirely, and updating NEWS and cvs.texinfo,
     but in case it is a good idea to give people more time
     to complain if they would miss it, I'll just add this
     quick and dirty error message for now. */
    error (1, 0, "the -V option is obsolete and should not be used");

    ifdef 0
    if (atoi (optarg) < 0)
        error (1, 0, "must specify a version number to -V");
    if (options)
        free (options);
    options = xmalloc (strlen (optarg) + 1 + 2); /* for the -V */
    (void) sprintf (options, "-V%s", optarg);
    endif
    break;

case 'u':
    unidiff = 1; /* Unidiff */
    break;

case 'c':
    /* Context diff */
    unidiff = 0;
    break;

case '?':
    default:
        usage (patch_usage);
        break;
}

argc -= optind;
argv += optind;

/* Sanity checks */
if (argc < 1)
    usage (patch_usage);

if (toptwo_diffs && patch_short)
    error (1, 0, "-t and -s options are mutually exclusive");
if (toptwo_diffs && (date1 != NULL || date2 != NULL ||
    rev1 != NULL || rev2 != NULL))
    error (1, 0, "must not specify revisions/dates with -t option");
if (toptwodiffs && (date1 == NULL && date2 == NULL &&
    rev1 == NULL && rev2 == NULL))
    error (1, 0, "must specify at least one revision/date");
if (date1 != NULL && date2 != NULL)
    if ( RCS_datecmp (date1, date2) >= 0 )
        error (1, 0, "second date must come after first date");
/* if options is NULL, make it a NULL string */
if (options == NULL)
    options = xstrdup ("");

sifdef CLIENT_SUPPORTED
    if (clean_active)
        /* We're the client side. Fire up the remote server. */
        start_server ();
        ignore_setup ();
        if (local)
            send_arg("-l");
            if (force_tag_match) send_arg("-f");
            if (toptwodiffs) send_arg("-t");
            if (patch_short) send_arg("-r");
            if (branch) send_arg("-u");

        if (rev1)
            option_with_arg ("-r", rev1);
        if (date1)
            client_senddate (date1);
        if (rev2)
            option_with_arg ("-r", rev2);
        if (date2)
            client_senddate (date2);
        if (options) send_arg (options);
        
        int i;
        for (i = 0; i < argc; ++i)
            send_arg(argv[i]);
        return get_responses send_close ();
    }
sendif

/* clean up if we get a signal */
sifdef SIGHUP
    (void) SIG_handler (SIGHUP, patch_cleanup);
sendif
sifdef SIGINT
    (void) SIG_handler (SIGINT, patch_cleanup);
sendif
sifdef SIGQUIT
    (void) SIG_handler (SIGQUIT, patch_cleanup);
sendif
sifdef SIGPIPE
    (void) SIG_handler (SIGPIPE, patch_cleanup);
sendif
sifdef SIGTERM
    (void) SIG_handler (SIGTERM, patch_cleanup);
sendif

db = open_module (");
for (i = 0; i < argc; ++i)
    err += do_module (db, argv[i], PATCH, "Patching", patchproc,
        (char *) NULL, 0, 0, (char *) NULL);
    close_module (db);
    free (options);
    patch_cleanup ();
return (err);
}
/*
 * callback proc for doing the real work of patching
 */
/* ARGUSED */
static int
patchproc (pargs, argv, xwhere, mwhere, mfile, shorten, localspecified, mname, msg)
    int *pargs;
    char **argv;
    char *xwhere;
    
    if (true)
        return 0;
    
    for (i = 0; i < argc; ++i)
        for (j = 0; j < argv[i]; ++j)
            if (argv[i][j] == '-')
                continue;
            }
char *mwhere;
char *mfile;
int shorten;
int local, specified;
char *name;
char *msg;
{
  int err = 0;
  int which;
  char *repository;
  char *where;

  repository = xmalloc (CVSroot, directory + strlen (argv[0])
  + (mfile == NULL ? 0 : strlen (mfile))) + 30;
  (void) sprintf (repository, "%s/%s", CVSroot, directory, argv[0]);
  where = xmalloc (strlen (argv[0]) + (mfile == NULL ? 0 : strlen (mfile))
  + 10);
  (void) strcpy (where, argv[0]);

  /* if mfile isn't null, we need to set up to do only part of the module */
  if (mfile != NULL)
    {
      char *cp;
      char *path;

      /* if the portion of the module is a path, put the dir part on repos */
      if ((cp = strrchr (mfile, '/')) != NULL)
        {
          *cp = '\0';
          (void) strcat (repository, '/');
          (void) strcat (repository, mfile);
          mfile = cp + 1;
        }
      /* take care of the rest */
      path = xmalloc (strlen (repository) + strlen (mfile) + 5);
      (void) sprintf (path, "%s/%s", repository, mfile);
      if (isdir (path))
        {
          /* directory means repository gets the dir tacked on */
          (void) strcpy (repository, path);
          (void) strcat (repository, '/');
          (void) strcat (repository, mfile);
        }
      else
        {
          int i;

          /* a file means muck argv */
          for (i = 1; i < *pargc; i++)
            free (argv[i]);
          argv[0] = xstrdup (mfile);
          *pargc = 2;
          free (path);
        }

      /* cd to the starting repository */
      if (CVS_CHDIR (repository) < 0)
        {
          error (0, errno, "cannot chdir to %s", repository);
          free (repository);
          return (1);
        }
      free (repository);

      if [force$tag_match]
        which = WREPOS | WATTIC;
      else
        which = WREPOS;

      if (rev1 != NULL && !rev1_validated)
        {
          tag_check_valid (rev1, *pargc - 1, argv + 1, local, 0, NULL);
          rev1_validated = 1;
        }
      if (rev2 != NULL && !rev2_validated)
        {
          tag_check_valid (rev2, *pargc - 1, argv + 1, local, 0, NULL);
          rev2_validated = 1;
        }

      /* start the recursion processor */
      err = start_recursion (patchfileproc, FILESDONEPROC NULL, patchdirproc,
                              (DIRLEAVEPROC) NULL, NULL,
                              *pargc - 1, argv + 1, local, 0, 1, where, 1);
      free (where);
    }
return (err);
}

/*
 * Called to examine a particular RCS file, as appropriate with the options
 * that were set above.
 */
/* ARGUSUSED */
static int
patchfileproc (callerdat, finfo)
{
struct utimbuf t1;
char *vers, *vers_head;
char *rcs = NULL, RCSnode = rcsfile;
FILE *fp1, *fp2, *fp3;
int ret = 0;
int isattic = 0;
int retcode = 0;
char *file1;
char *file2;
char *strippath;
char *line1, *line2;
size_t line1len = char_allocated;
size_t line2len = char_allocated;
char *cp1, *cp2;
FILE *fp;
int line_length;

line1 = NULL;
line2len = char_allocated = 0;
line1len = char_allocated = 0;

/* find the parsed rcs file */
if ((rcsfile = finfo->rcs) == NULL)
{
  ret = 1;
  goto out2;
}
if ((rcsfile->flags & VALID) && (rcsfile->flags & INATTIC))
  isattic = 1;
rcs = xmalloc ((strlen (finfo->file) + sizeof (RCSEXT) + 5));
(void) sprintf (rcs, "%s", finfo->file, RCSEXT);
/* if vers_head is NULL, may have been removed from the release */
if (isattic && rev2 == NULL && date2 == NULL)
  vers_head = NULL;
else
{
  vers_head = RCS_getversion (rcsfile, rev2, date2, force_tag_match,
                            (int *) NULL);
  if (vers_head != NULL && RCSdead (rcsfile, vers_head))
  {
    free (vers_head);
    vers_head = NULL;
  }
}
if (toptwolevels)
{
  if (vers_head == NULL)
  {
    ret = 1;
    goto out2;
  }

if (date1)
  date1 = xmalloc (MAXDATELEN);
  *date1 = '\0';
  if (RCS_getrettime (rcsfile, vers_head, date1, 1) == -1)
  {
    if (!really_quiet)
      error (0, 0, "cannot find date in rcs file %s revision %s",
            rcs, vers_head);
    ret = 1;
    goto out2;
  }
  vers_tag = RCS_getversion (rcsfile, rev1, date1, force_tag_match,
                             (int *) NULL);
  if (vers_tag != NULL && RCSdead (rcsfile, vers_tag))
  {
    free (vers_tag);
    vers_tag = NULL;
  }
if (vers_tag == NULL && vers_head == NULL) {  
    /* Nothing known about specified revs. */  
    ret = 0;  
    goto out2;  
}

if (vers_tag && vers_head && strcmp (vers_head, vers_tag) == 0) {  
    /* Not changed between releases. */  
    ret = 0;  
    goto out2;  
}

if (patch_short) {  
    cvs_output ("File ", 0);  
    cvs_output (info->fullname, 0);  
    if (vers_tag == NULL) {  
        cvs_output ("* is new; current revision ", 0);  
        cvs_output (vers_head, 0);  
        cvs_output ("\n", 1);  
    } else if (vers_head == NULL) {  
        cvs_output ("* is removed; not included in ", 0);  
        if (rev2 != NULL) {  
            cvs_output ("release tag ", 0);  
            cvs_output (rev2, 0);  
        } else if (date2 != NULL) {  
            cvs_output ("release date ", 0);  
            cvs_output (date2, 0);  
        } else if (date1 != NULL) {  
            cvs_output ("current release", 0);  
            cvs_output ("\n", 1);  
        } else {  
            cvs_output ("changed from revision ", 0);  
            cvs_output (vers_tag, 0);  
            cvs_output (" to ", 0);  
            cvs_output (vers_head, 0);  
            cvs_output ("\n", 1);  
        }  
        ret = 0;  
    } goto out2;  
  
} /* Create 3 empty files. I'm not really sure there is any advantage  
to doing so now rather than just waiting until later. */  
tmpfile1 = cvspopen (tmpfile1, "w+");  
if (fp1 == NULL) {  
    error (0, errno, "cannot create temporary file \%s", tmpfile1);  
    ret = 1;  
    goto out;  
} else if (fclose (fp1) < 0)  
    error (0, errno, "warning: cannot close \%s", tmpfile1);  
tmpfile2 = cvspopen (tmpfile2, "w+");  
if (fp2 == NULL) {  
    error (0, errno, "cannot create temporary file \%s", tmpfile2);  
    ret = 1;  
    goto out;  
} else if (fclose (fp2) < 0)  
    error (0, errno, "warning: cannot close \%s", tmpfile2);  
tmpfile3 = cvspopen (tmpfile3, "w+");  
if (fp3 == NULL) {  
    error (0, errno, "cannot create temporary file \%s", tmpfile3);  
    ret = 1;  
    goto out;  
} else if (fclose (fp3) < 0)  
    error (0, errno, "warning: cannot close \%s", tmpfile3);
if (vers_tag != NULL)
{
    retcode = RCS_checkout (rcsfile, (char *) NULL, vers_tag,
    rev1, options, tmpfile1,
    (RCSCHECKOUTPROC) NULL, (void *) NULL);
    if (retcode != 0)
    {
        error (0, 0,
            "cannot check out revision %s of %s", vers_tag, rcs);
        ret = 1;
        goto out;
    }
}
memset ((char *) &t, 0, sizeof (t));
if ((t.actime = t.modtime = RCS_getmtime (rcsfile, vers_tag,
    (char *) 0, 0) != -1)
    /* I believe this timestamp only affects the dates in our diffs,
    and therefore should be on the server, not the client. */
    (void) utime (tmpfile1, &t);
} else if (toptwo_diffs)
{
    ret = 1;
    goto out;
}
if (vers_head != NULL)
{
    retcode = RCS_checkout (rcsfile, (char *) NULL, vers_head,
    rev2, options, tmpfile2,
    (RCSCHECKOUTPROC) NULL, (void *) NULL);
    if (retcode != 0)
    {
        error (0, 0,
            "cannot check out revision %s of %s", vers_head, rcs);
        ret = 1;
        goto out;
    }
}
    /* I believe this timestamp only affects the dates in our diffs,
    and therefore should be on the server, not the client. */
    (void) utime (tmpfile2, &t);
}
switch (diff_exec (tmpfile1, tmpfile2, unidiff ? "-u" : "-e", tmpfile3))
{
    case -1:
        /* fork/wait failure */
        error (1, errno, "fork for diff failed on %s", rcs);
        break;
    case 0:
        /* nothing to do */
        break;
    case 1:
        /* The two revisions are really different, so read the first two
         * lines of the diff output file, and make them to include more
         * reasonable file names that "patch" will understand.
         */
        /* Output an "index" line for patch to use */
        cvsoutput ("Index: * 0;");
        cvsoutput ("Index: * 0;");
        cvsoutput ("Index: * 0;");
        fp = open_file (tmpfile1, "r");
        if (getline (&line1, &line1_char_allocated, fp) < 0 || |
            getline (&line2, &line2_char_allocated, fp) < 0)
            if (feof (fp))
                error (0, 0, "\n"
                    failed to read diff file header %s for %s: end of file", tmpfile3, rcs);
    }
    else
    {
        error (0, errno,
            "failed to read diff file header %s for %s",
            tmpfile3, rcs);
        ret = 1;
        if (fclose (fp) < 0)
            error (0, errno, "error closing %s", tmpfile3);
        goto out;
    }
}
if (unidiff)
{
    if (!strn cmp (line1, "*****", 4) != 0 || |
        strn cmp (line2, "----", 4) != 0 || |
        (cp1 = strchr (line1, '\t')) == NULL || |
        (cp2 = strchr (line2, '\t')) == NULL)
        if (fclose (fp) < 0)
            error (0, 0, "invalid diff header for %s", rcs);
        ret = 1;
        if (fclose (fp) < 0)
            error (0, errno, "error closing %s", tmpfile3);
        goto out;
else
{
    if (strncmp (line1, "--- ", 4) != 0 ||
        strncmp (line2, "+++ ", 4) != 0 ||
        (cp1 = strchr (line1, '\n')) == NULL ||
        (cp2 = strchr (line2, '\n')) == NULL)
    {
        error (0, 0, "invalid unidiff header for \%s", rcs);
        ret = 1;
    }
    if (fclose (fp) < 0)
        error (0, errno, "error closing \%s", tmpfile3);
    goto out;
}
else
{
    if (CVSroot_directory != NULL)
    {
        strippath = xmalloc (strlen (CVSroot_directory) + 10);
        (void) sprintf (strippath, "%s", CVSroot_directory);
    }
    else
    {
        strippath = xstrdup (REPOS_STRIP);
    }
    if (strncmp (rcs, strippath, strlen (strippath)) == 0)
    {
        rcs = strlen (strippath);
        free (strippath);
    }
    if (vers_tag != NULL)
    {
        file1 = xmalloc (strlen (info->fullname) +
                        strlen (vers_tag) + 10);
        (void) sprintf (file1, "%s-%s", info->fullname, vers_tag);
    }
    else
    {
        file1 = xstrdup (DEVNULL);
    }
    file2 = xmalloc (strlen (info->fullname) +
                    strlen (vers_tag) + strlen (vers_head) + 10);
    (void) sprintf (file2, "%s-%s", info->fullname,
                   vers_head ? vers_head : "removed");
    /* Note that the string "diff" is specified by POSIX (for -c)
       and is part of the diff output format, not the name of a
       program. */
    if (unidiff)
    {
        cvs_output ("diff -u ", 0);
        cvs_output (file1, 0);
        cvs_output (file2, 0);
        cvs_output ("-w", 1);
        cvs_output ("--- ", 0);
        cvs_output (file1, 0);
        cvs_output (file2, 0);
        cvs_output ("--- ", 0);
    }
    else
    {
        cvs_output ("diff -c ", 0);
        cvs_output (file1, 0);
        cvs_output (file2, 0);
        cvs_output ("-w", 1);
        cvs_output ("**** ", 0);
        cvs_output (file1, 0);
        cvs_output (file2, 0);
        cvs_output ("**** ", 0);
    }
}
(cvs_output (info->fullname, 0);
(cvs_output (fp), 0);
    /* spew the rest of the diff out */
    while (line_length >= 0)
        cvs_output (line1, 0);
    cvs_output (line2, 0);
    if (line_length < 0 & & !feof (fp))
        error (0, errno, "cannot read \%s", tmpfile3);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close \%s", tmpfile3);
    free (file1);
    free (file2);
break;
default:
error (0, 0, "diff failed for %s", finfo->fullname);
}
out:
if (line1)
  free (line1);
if (line2)
  free (line2);
if (CVS_UNLINK (tmpfile1) < 0)
  error (0, errno, "cannot unlink %s", tmpfile1);
if (CVS_UNLINK (tmpfile2) < 0)
  error (0, errno, "cannot unlink %s", tmpfile2);
if (CVS_UNLINK (tmpfile3) < 0)
  error (0, errno, "cannot unlink %s", tmpfile3);
free (tmpfile1);
free (tmpfile2);
free (tmpfile3);
tmpfile1 = tmpfile2 = tmpfile3 = NULL;
out:
if (rcs != NULL)
  free (rcs);
740  return (ret);
}

/* Print a warm fuzzy message */
static Dtype patch_dirproc (callerdat, dir, repos, update_dir, entries)
  void *callerdat;
char *dir;
char *repos;
char *update_dir;
List *entries;
{
  if (! quiet)
    error (0, 0, "Diffing %s", update_dir);
  return (R_PROCESS);
}

760  /* Clean up temporary files */
static RETSIGTYPE patch_cleanup ()
{
  if (tmpfile1 != NULL)
  {
    (void) unlink_file (tmpfile1);
    free (tmpfile1);
  }
  if (tmpfile2 != NULL)
  {
    (void) unlink_file (tmpfile2);
    free (tmpfile2);
  }
  if (tmpfile3 != NULL)
  {
    (void) unlink_file (tmpfile3);
    free (tmpfile3);
  }
  tmpfile1 = tmpfile2 = tmpfile3 = NULL;
}
A.44  rcs.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * The routines contained in this file do all the rcs file parsing and
 * manipulation
 */

#include <assert.h>
#include "cvs.h"
#include "edit.h"
#include "hardlink.h"

int preserve_perms = 0;

/* The RCS -k options, and a set of enums that must match the array.
 These come first so that we can use enum kflag in function
 prototypes. */
static const char +const kflags[] =
{ "av", "vrl", "k", "r", "o", "b", (char *) NULL};
enum kflag { KFLAG_KV = 0, KFLAG_KVL, KFLAG_K, KFLAG_V, KFLAG_O, KFLAG_B };
The RCS file reading functions are called a lot, and they do some
string comparisons. This macro speeds things up a bit by skipping
the function call when the first characters are different. It
evaluates its arguments multiple times. */
define STRREQ(a, b) \{ (a)[0] == (b)[0] && strcmp (a), (b) == 0 \}

/* We don't want to use isspace() from the C library because:
   * 1. The definition of "whitespace" in RCS files includes ASCII
      backspace, but the C locale doesn't.
   * 2. isspace is an expensive function call in some implementations
      due to the addition of wide character support.
   */
static const char spacetable[] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};
define whitespace(c) \{ spacetable[(unsigned char)c] != 0 \}

/* Parse an rcsfile given a user file name and a repository. If there
is an error, we print an error message and return NULL. If the file
does not exist, we return NULL without printing anything (I'm not
sure this allows the caller to do anything reasonable, but it is
the current behavior). */
RCSNode *
RCSparse (file, repos)
const char +file;
const char +repos;
{
   RCSNode +rcs;
   FILE +fp;
   RCSNode +rsvn;
   char +rcfile;
   /* We're creating a new RCSNode, so there is no hope of finding it
      in the cache. */
   rcsbuf_cache_close (fp);
}
resfile = xmalloc(strlen(repos) + strlen(file) + sizeof (RCSEX) + sizeof (CVSATTIC) + 10);
(void) sprintf (resfile, "%s/%s", repos, file, RCSEX);
if ((fp = CVS_POPEN (resfile, FOPEN_BINARY_READ)) != NULL) {
    rcs = RCS_parsersrcfile_l(fp, resfile);
    if (rcs != NULL) {
        rcs->flags |= VALID;
        retval = rcs;
        goto out;
    } else if (! exists (errno)) {
        error (0, errno, "cannot open %s", resfile);
        retval = NULL;
        goto out;
    }
    (void) sprintf (resfile, "%s/%s/%s", repos, CVSATTIC, file, RCSEX);
    if ((fp = CVS_POPEN (resfile, FOPEN_BINARY_READ)) != NULL) {
        rcs = RCS_parsersrcfile_l(fp, resfile);
        if (rcs != NULL) {
            rcs->flags |= INATTIC;
            rcs->flags |= VALID;
            retval = rcs;
            goto out;
        } else if (! exists (errno)) {
            error (0, errno, "cannot open %s", resfile);
            retval = NULL;
            goto out;
        }
    } else if (defined (SERVER_SUPPORT) & defined (FILENAME_CASE_INSENSITIVE)) {
        int status;
        char *found_path;
        /* The client might be asking for a file which we do have (which the client doesn't know about), but for which the filename case differs. We only consider this case if the regular CVS_POPENs fail, because fopen_case is such an expensive call. */
        (void) sprintf (resfile, "%s/%s", repos, file, RCSEX);
        status = fopen_case (resfile, "rb", &fp, &found_path);
        if (status == 0) {
            rcs = RCS_parsersrcfile_l(fp, resfile);
            if (rcs != NULL) {
                rcs->flags |= VALID;
                free (rcs->path);
                rcs->path = found_path;
                retval = rcs;
                goto out;
            }
        } else if (! exists (errno)) {
            error (0, status, "cannot open %s", resfile);
            retval = NULL;
            goto out;
        }
    } else if (defined (SERVER_SUPPORT) & defined (FILENAME_CASE_INSENSITIVE)) {
        int status;
        char *found_path;
        /* The client might be asking for a file which we do have (which the client doesn't know about), but for which the filename case differs. We only consider this case if the regular CVS_POPENs fail, because fopen_case is such an expensive call. */
        (void) sprintf (resfile, "%s/%s", repos, file, RCSEX);
        status = fopen_case (resfile, "rb", &fp, &found_path);
        if (status == 0) {
            rcs = RCS_parsersrcfile_l(fp, resfile);
            if (rcs != NULL) {
                rcs->flags |= INATTIC;
                rcs->flags |= VALID;
            }
            free (rcs->path);
            rcs->path = found_path;
            retval = rcs;
            goto out;
        } else if (! exists (errno)) {
            error (0, status, "cannot open %s", resfile);
            retval = NULL;
            goto out;
        }
endif

tretval = NULL;

out:
free (rcsfile);

return retval;
}

/* Parse a specific rcsfile. */
RCSNode *
RCS_parserscfile (rcsfile)
char *rcsfile;
{
FILE *fp;
RCSNode *rcs;

/* We're creating a new RCSNode, so there is no hope of finding it
in the cache. */
rcbuf_cache_close ();

/* open the rcsfile */
if ((fp = CVS_FOPEN (rcsfile, FOPEN_BINARY_READ)) == NULL)
{
error (0, errno, "Couldn't open rcs file '"", rcsfile);
return (NULL);
}

rcs = RCS_parserscfile_i (fp, rcsfile);
return (rcs);
}

/*
static RCSNode *
RCS_parserscfile_i (fp, rcsfile)
FILE *fp;
const char *rcsfile;
{
RCSNode *rdata;
struct rcsbufferrcsbuf;
char *key, *value;

/* make a node */
rdata = (RCSNode *) xmalloc (sizeof (RCSNode));
memset ((char *) rdata, 0, sizeof (RCSNode));
rdata->refcount = 1;
rdata->path = xstrdup (rcsfile);

/* Process HEAD, BRANCH, and EXPAND keywords from the RCS header.
Most cvs operations on the main branch don't need any more
information. Those that do call RCS_parserscfile to parse
the rest of the header and the deltas. */
rcbuf_open (&rcbuf, fp, rcsfile, 0);
if (! rcbuf_getkey (&rcbuf, &key, &value))
goto lerror;
if (STRREQ (key, RCSDESC))
goto lerror;
if (STRREQ (RCSHEAD, key) && value != NULL)
rddata->head = rcbuf_valcopy (&rcbuf, value, 0, (size_t *) NULL);
if (! rcbuf_getkey (&rcbuf, &key, &value))
goto lerror;
if (STRREQ (key, RCSDESC))
goto lerror;
if (STRREQ (RCSBRANCH, key) && value != NULL)
{
char *cp;
rddata->branch = rcbuf_valcopy (&rcbuf, value, 0, (size_t *) NULL);
if ((numdots (rddata->branch) & 1) != 0)
{
/* turn it into a branch if it's a revision */
 cp = strrchr (rddata->branch, '\');
 else = '\0';
}
}
/* Look ahead for expand, stopping when we see desc or a revision

number. */
while (1) {
    char *cp;
    if (STREQ (RCSEXAND, key)) {
        rdata->expand = rcsbuf->expand (rcsbuf, value, 0,
                        (size_t *) NULL);
        break;
    }
    for (cp = key; (isdigit (*cp) || *cp == '.' || *cp == '\0'; cp++)
        /* do nothing */ ;
    if (STREQ (RCSDESC, key)) break;
    if (! rcsbuf_getkey (rcsbuf, &key, &value)) break;
}
rdata->flags |= PARTIAL;
rcsbuf_cache (rdata, &rcsbuf);
return rdata;

error:
    error (1, 0, "'%s' does not appear to be a valid rcs file", rcsfile);
rcsbuf_close (&rcsbuf);
freevcsnode (&rdata);
fclose (fp);
return (NULL);

/* Do the real work of parsing an RCS file.
400 On error, die with a fatal error; if it returns at all it was successful.
If PFP is NULL, close the file when done. Otherwise, leave it open
and store the FILE * in *PFP. */
void RCS_reparsercsfile (rdata, pfp, rcsbufp)
RCSNode *rdata;
FILE **pfp;
struct rcsbuffer *rcsbufp;
410 {
    FILE *fp;
    char *rcsfile;
    struct rcsbuffer rcsbuf;
    Node eq, *kv;
    RCSVers *vnode;
    int gotkey;
    char *cp;
    char *key, *value;
    assert (rdata != NULL);
    rcsfile = rdata->path;
    rcsbuf_cache_open (rdata, 0, &fp, &rcsbuf);
    /* make a node */
    /* This probably shouldn't be done until later; if a file has an
empty revision tree (which is permissible), rdata->versions should
be NULL. -twp */
    rdata->versions = getlist ();
430
    /* process all the special header information, break out when we get to
the first revision delta */
    gotkey = 0;
    for (; ) {
    /* get the next key/value pair */
    if (gotkey)
440 {
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
            { error (1, 0, "'%s' does not appear to be a valid rcs file", rcsfile);
            }
        gotkey = 0;
    }
/* Skip head, branch and expand tags; we already have them. */
if (STREQ (key, RCSHEAD))
  || STREQ (key, RCSBRANCH)
  || STREQ (key, RCSEXPNAND))
  continue;
}

if (STREQ (key, "access"))
{
  /* We pass the POLISH parameter as 1 because
   RCSaddaccess expects nothing but spaces. FIXME:
   It would be easy and more efficient to change
   RCSaddaccess. */
  rdata->access = rcsbuf_valcopy (rcsbuf, value, 1,
      (size_t) NULL);
  continue;
}

/* We always save lock information, so that we can handle
   -kk correctly when checking out a file. */
if (STREQ (key, "locks"))
{
  if (value != NULL)
    {
      / We pass the POLISH parameter as 1 because
       RCSaddlocks expects nothing but spaces. FIXME:
       It would be easy and more efficient to change
       RCSaddlocks. */
      rdata->locks = rcsbuf_valcopy (rcsbuf, value, 0,
          (size_t) NULL);
      if (! rcsbuf_getkey (rcsbuf, &key, &value))
        {
          error (1, 0, "premature end of file reading %s", rcsfile);
          if (STREQ (key, "strict") && value == NULL)
            {
              rdata->strict_locks = 1;
              gotkey = 1;
            }
        }
  else
    gotkey = 1;
}
continue;

if (STREQ (RCSSYMBOLS, key))
{
  if (value != NULL)
    rdata->symbols = rcsbuf_valcopy (rcsbuf, value, 0,
        (size_t) NULL);
    continue;
}

/* check key for ".", and digits (probably a rev) if it is a
 * revision or 'desc', we are done with the headers and are down to the
 * revision delta, so we break out of the loop
 */
for (cp = key; !isdigit (*cp) || *cp == '.' && *cp == '0'; cp++)
  /* do nothing */;
/* Note that when comparing with RCSDATE, we are not massaging
   VALUE from the string found in the RCS file. This is OK
   since we know exactly what to expect. */
if (*cp == '\' && strncmp (RCSDATE, value, (size_t) RCSDATE) == 0)
  break;

if (STREQ (key, RCSDESC))
  break;
if (STREQ (key, "comment"))
  {
    rdata->comment = rcsbuf_valcopy (rcsbuf, value, 0,
        (size_t) NULL);
    continue;
  }
if (rdata->other != NULL)
  rdata->other = getlist ();
kv = getnode ();
kv->type = RCSFIELD;
kv->key = xstrdup (key);
kv->data = rcsbuf_valcopy (rcsbuf, value, 1, (size_t) NULL);
if (addnode (rdata->other, kv) != 0)
  {
    error (0, 0, "warning: duplicate key '%s' in RCS file '%s'",
        key, rcsfile);
    freenode (kv);
  }

/* if we haven't grabbed it yet, we didn't want it */

/* We got out of the loop, so we have the first part of the first
revision delta in KEY (the revision) and VALUE (the date key
and its value). This is what getdelta expects to receive. */

while ((vnode = getdelta (&rcsbuf, rcsfile, &key, &value)) != NULL)
{
  /* get the node */
  q = getnode ();
  q->type = RCSVERS;
  q->delproc = rcsvers_delproc;
  q->data = (char *) vnode;
  q->key = vnode->version;

  /* add the nodes to the list */
  if (addnode (rdata->versions, q) != 0)
  {
    if (0)
      purify_print("WARNING: Adding duplicate version: %s (%s)",
                  q->key, rcsfile);
    freenode (q);
  }
}

/* Here KEY and VALUE are whatever caused getdelta to return NULL. */
if (STREQ (key, RCSDESC))
{
  if (rdata->desc != NULL)
  {
    error (0, 0,
           "warning: duplicate key '%s' in RCS file '%s'",
           key, rcsfile);
    free (rdata->desc);
  }

  rdata->desc = xstrdup (value);
}

rdata->deletemos = rcsbuf->tell (&rcsbuf);

if (pfp == NULL)
  rcsbuf->cache (rdata, &rcsbuf);
else
{
  *pfp = fp;
  *rcsbufp = rcsbuf;
}

rdata->flags &= ~PARTIAL;
}

/*
* Fully parse the RCS file. Store all keyword/value pairs, fetch the
* log messages for each revision, and fetch add and delete counts for
* each revision (we could fetch the entire text for each revision,
* but the only caller, logfileproc, doesn't need that information,
* so we don't want to waste the memory required to store it). The add and
* delete counts are stored on the OTHER field of the RCSVERSNODE
* structure, under the names "add" and "delete", so that we don't
* waste the memory space of extra fields in RCSVERSNODE for code
* which doesn't need this information.
*
* void RCS_FullyParse (rcs)
* RCSNode *rcs;
* {
*   FILE *fp;
*   struct rcsbuffer rcsbuf;
*   RCS_reparsecsfile (rcs, &fp, &rcsbuf);
*   while (1)
*   {
*     char *key, *value;
*     Node *vers;
*     RCSVers *vnode;
*     /* Rather than try to keep track of how much information we
*        have read, just read to the end of the file. */
*     if (!rcsbuf->prevnum (&rcsbuf, &key))
*       break;
*     vers = findnode (rcs->versions, key);
*     if (vers == NULL)
*       error (1, 0,
*              "mismatch in rcs file %s between deltas and deltapats",
*              rcs->path);
*     vnode = (RCSVers *) vers->data;
* }
* */
*/
while (rcbuf_getkey (&rcbuf, &key, &value))
{
    if (!STREQ (key, "text"))
    {
        Node *kv;
        if (vnode->other == NULL)
            vnode->other = getlist ();
        kv = getnode ();
        if (vnode->other == NULL)
            vnode->other = getlist ();
        kv->type = RCSFIELD;
        kv->key = xstrdup (key);
        kv->data = rcsbuf_valcopy (&rcbuf, value, 1, (size_t *)&NULL);
        if (addnode (vnode->other, kv) != 0)
            error (0, 0,
                "warning: duplicate key '%s' in version '%s' of RCS file '%s'",
                key, vnode->version, rcs->path);
        freenode (kv);
    }
    continue;
}

if (!STREQ (vnode->version, rcs->head))
{
    unsigned long add, del;
    char buf[50];
    Node *kv;

    /* This is a change text. Store the add and delete counts. */
    add = 0;
    del = 0;
    if (value != NULL)
    {
        size_t vallen;
        const char *cp;
        rcsbuf_valpolish (&rcbuf, value, 0, &vallen);
        cp = value;
        while (cp < value + vallen)
        {
            char op;
            unsigned long count;
            op = *cp++;
            if (op == 'a' && &op != 'd')
                error (1, 0, "unrecognized operation \"%c\" in \"%s\",
                    op, rcs->path);
            else
            {
                add += count;
                while (count != 0)
                {
                    if (*cp == '\012')
                        --count;
                    else if (cp == value + vallen)
                    {
                        if (count != 1)
                            error (1, 0, \
                                "invalid rcs file \"%s\": premature end of value",
                                rcs->path);
                        else
                            break;
                    }
                    ++cp;
                }
                break;
            }
        }
        sprintf (buf, "%lu", add);
        kv = getnode ();
        kv->type = RCSFIELD;
        kv->key = xstrdup ("add");
        kv->data = xstrdup (buf);
        if (addnode (vnode->other, kv) != 0)
{ error (0, 0, \
  warning: duplicate key "%s" in version '%s' of RCS file "%s",
  key, vnode->version, rcs->path);  
  freenode (kv);  
}  
  sprintf (buf, "%s", del);  
  kv = getnode ();  
  kv->type = RCSFIELD;  
  kv->key = xstrdup (*delete*);  
  kv->data = xstrdup (buf);  
  if (addnode (vnode->other, kv) != 0)  
  { error (0, 0, \
  warning: duplicate key "%s" in version '%s' of RCS file "%s",
  key, vnode->version, rcs->path);  
  freenode (kv);  
  }  
  */ We have found the "text" key which ends the data for
  this revision. Break out of the loop and go on to the
  next revision. */  
  break;  
}  
  rcsbuf<cache (rcs, &rcsbuf);  
}  
/*  
 * freercsnode - free up the info for an RCSNode
 */  
void freercsnode (rnodep)  
RCNode **rnodep;  
{ if (rnodep == NULL || *rnodep == NULL)  
    return;  
  { (*rnodep)->refcount--;  
    if (!(*rnodep)->refcount != 0)  
    {  
      *rnodep = (RCNode *) NULL;  
      return;  
    }  
    free ((*rnodep)->path);  
    if (!(*rnodep)->head != (char *) NULL)  
    free ((rnodep)->head);  
    free ((rnodep)->branch != (char *) NULL)  
    free ((rnodep)->branch);  
    free (rnodep->other != (char *) NULL)  
    free (char *) *rnodep;  
    *rnodep = (RCNode *) NULL;  
  }  
}  
/*  
 * freercsnode_contents - free up the contents of an RCSNode without
 * freeing the node itself, or the file name, or the head, or the
 * path. This returns the RCSNode to the state it is in immediately
 * after a call to RCSparse.
 */  
static void freercsnode_contents (rnode)  
RCNode *rnode;  
{  
  dellist (&rnode->versions);  
  if (rnode->symbols != (List *) NULL)  
  dellist (&rnode->symbols);  
  if (rnode->symbols_data != (char *) NULL)  
  free (rnode->symbols_data);  
  if (rnode->expand != NULL)  
  free (rnode->expand);  
  if (rnode->other != (List *) NULL)  
  dellist (&rnode->other);  
  if (rnode->access != NULL)  
  free (rnode->access);  
  if (rnode->lock_data != NULL)  
  free (rnode->lock_data);  
  if (rnode->lock != (List *) NULL)  
  dellist (&rnode->lock);  
  if (rnode->comment != NULL)  
  free (rnode->comment);  
  if (rnode->desc != NULL)  
  free (rnode->desc);  
  }
static void
free_rcsvers_contents(rnode)
RCSVers *rnode;
{
    if (rnode->branches != (List *) NULL)
        dellist (&rnode->branches);
    if (rnode->date != (char *) NULL)
        free (rnode->date);
    if (rnode->next != (char *) NULL)
        free (rnode->next);
    if (rnode->author != (char *) NULL)
        free (rnode->author);
    if (rnode->state != (char *) NULL)
        free (rnode->state);
    if (rnode->other != (List *) NULL)
        dellist (&rnode->other);
    if (rnode->other_delta != NULL)
        dellist (&rnode->other_delta);
    if (rnode->text != NULL)
        freedeltatext (rnode->text);
    free ((char *) rnode);
}

/**
 * rcsvers_delproc - free up an RCSVers type node
 */

static void
rcsvers_delproc (p)

    Node *p;
{
    free_rcsvers_contents ((RCSVers *) p->data);
}

/**
 * These functions retrieve keys and values from an RCS file using a
 * buffer. We use this somewhat complex approach because it turns out
 * that for many common operations, CVS spends most of its time
 * reading keys, so it's worth doing some fairly hairy optimization. *
 */

#define RCSBUF_BUFSIZE (8192)

/**
 * The buffer we use to store data. This grows as needed. *
 */

static char *rcbufbuffer = NULL;

static size_t rcbufbuffer_size = 0;

/**
 * Whether rcbufbuffer is in use. This is used as a sanity check. *
 */

static int rcbufbufuse;

/**
 * Set up to start gathering keys and values from an RCS file. This
 * initializes RCSBUF. *
 */

static void
rcbuf_open (rcbuf, fp, filename, pos)

    struct rcbuffer *rcbuf;
    FILE *fp;
    const char *filename;
    unsigned long pos;
{
    if (rcbufbufuse)
        error (1, 0, "rcbuf_open: internal error");
    rcbufbufuse = 1;
    if (rcbufbuffer_size < RCSBUF_BUFSIZE)
        expand_string (&rcbufbuffer, &rcbufbuffer_size, RCSBUF_BUFSIZE);
    rcbuf->ptr = rcbufbuffer;
    rcbuf->ptrend = rcbufbuffer;
    rcbuf->fp = fp;
    rcbuf->filename = filename;
    rcbuf->pos = pos;
    rcbuf->vlen = 0;
    rcbuf->at_string = 0;
    rcbuf->embedded_at = 0;
}

/**
 * Stop gathering keys from an RCS file. *
 */

static void
rcbuf_close (rcbuf)

    struct rcbuffer *rcbuf;
{

if (!rcsbuf_inuse)
    error (1, 0, "rcsbuf_close: internal error");
rcsbuf_inuse = 0;
}

/* Read a key/value pair from an RCS file. This sets *KEYP to point to the key, and *VALUEP to point to the value. A missing or empty value is indicated by setting *VALUEP to NULL. This function returns 1 on success, or 0 on EOF. If there is an error reading the file, or an EOF on an unexpected location, it gives a fatal error. This sets *KEYP and *VALUEP to point to storage managed by rcsbuf_getkey. Moreover, *VALUEP has not been massaged from the RCS format: it may contain embedded whitespace and embedded ']' characters. Call rcsbuf_valcopy or rcsbuf_valpolish to do appropriate massaging. */

static int rcsbuf_getkey(rcsbuf, keyp, valp)
struct rcsbuffer *rcsbuf;
char **keyp;
char **valp;
{
    register const char * const my_spacetab = spacetab;
    register char *ptr, *ptrend;
    char c;
    #define my_whitespace(c) (my_spacetab[(unsigned char)c] != 0)

    rcsbuf->vlen = 0;
    rcsbuf->at_string = 0;
    rcsbuf->embedded_at = 0;
    ptr = rcsbuf->ptr;
    ptrend = rcsbuf->ptrend;
    /* Sanity check. */
    if (ptr < rcsbuf_buffer || ptr > rcsbuf_buffer + rcsbuf_buffer_size)
        abort();

    /* If the pointer is more than RCSBUF_BUFSIZE bytes into the buffer, move back to the start of the buffer. This keeps the buffer from growing indefinitely. */
    if (ptr - rcsbuf_buffer >= RCSBUF_BUFSIZE)
    {
        int len;
        len = ptrend - ptr;
        /* Sanity check: we don't read more than RCSBUF_BUFSIZE bytes */
        if (len > RCSBUF_BUFSIZE)
            abort();

        /* Update the POS field, which holds the file offset of the first byte in the RCSBUF_BUFFER buffer. */
        rcsbuf->pos += ptr - rcsbuf_buffer;
        memcpy (rcsbuf_buffer, ptr, len);
        ptr = rcsbuf_buffer;
        ptrend = ptr + len;
        rcsbuf->ptrend = ptrend;
    }

    /* Skip leading whitespace. */
    while (1)
    {
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
            if (ptr == NULL)
                return 0;
            ptrend = rcsbuf->ptrend;
        }
        c = *ptr;
        if (!my_whitespace(c))
            break;
        ++ptr;
    }

    /* We've found the start of the key. */
    *keyp = ptr;
    if (c != ';' ||}
{  
  while (1)  
  {  
    ++ptr;  
    if (ptr >= ptrend)  
    {  
      ptr = rcsbuf_fill (rcbuf, ptr, keyp, (char **) NULL);  
      if (ptr == NULL)  
        error (1, 0, "EOF in key in RCS file %s",  
            rcsbuf->filename);  
      ptrend = rcsbuf->ptrend;  
    }  
    c = *ptr;  
    if (c == ';' | my_whitespace (c))  
      break;  
  }  
  
  /* Here *KEYP points to the key in the buffer, C is the character  
  we found at the of the key, and PTR points to the location in  
  the buffer where we found C. We must set *PTR to  \0 in order  
  to terminate the key. If the key ended with ';', then there is  
  no value. */  
  *ptr = '\0';  
  ++ptr;  
  if (c == ';')  
  {  
    *valp = NULL;  
    rcsbuf->ptr = ptr;  
    return 1;  
  }  
  }  
  /* C must be whitespace. Skip whitespace between the key and the  
  value. If we find ';' now, there is no value. */  
  while (1)  
  {  
    if (ptr >= ptrend)  
    {  
      ptr = rcsbuf_fill (rcbuf, ptr, keyp, (char **) NULL);  
      if (ptr == NULL)  
        error (1, 0, "EOF while looking for value in RCS file %s",  
            rcsbuf->filename);  
      ptrend = rcsbuf->ptrend;  
    }  
    c = *ptr;  
    if (c == ';')  
    {  
      *valp = NULL;  
      rcsbuf->ptr = ptr + 1;  
      return 1;  
    }  
    if (! my_whitespace (c))  
      break;  
    ++ptr;  
  }  
  /* Now PTR points to the start of the value, and C is the first  
  character of the value. */  
  if (c == '@')  
    *valp = ptr;  
  else  
  {  
    char *pat;  
    size_t vlen;  
    /* Optimize the common case of a value composed of a single  
    */  
    rcsbuf->n_string = 1;  
    ++ptr;  
    *valp = ptr;  
  }  
  while (1)  
  {  
    while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)  
    {  
      /* Note that we pass PTREND as the PTR value to  
      rcsbuf_fill, so that we will wind up setting PTR to  
      the location corresponding to the old PTREND, so  
      that we don't search the same bytes again. */  
      ptr = rcsbuf_fill (rcbuf, ptr, keyp, valp);  
      if (ptr == NULL)  
        error (1, 0,  
            "EOF while looking for end of string in RCS file %s",  
            rcsbuf->filename);  
    }  
  }  
}
rcsbuf->filename);
ptrend = rcsbuf->ptrend;
}
/* Handle the special case of an \'\' right at the end of
the known bytes. */
if (pat + 1 >= ptrend)
{
    /* Note that we pass PAT, not PTR, here. */
    pat = rcsbuf_fill (rcsbuf, pat, keyp, valp);
    if (pat == NULL)
    {
        /* EOF here is OK; it just means that the last
character of the file was an \'\' terminating a
value for a key type which does not require a
trailing \'\'. */
        pat = rcsbuf->ptrend - 1;
    }
    ptrend = rcsbuf->ptrend;
}
/* Note that the value of PTR is bogus here. This is
OK, because we don't use it. */
if (pat + 1 >= ptrend || pat[1] != '@')
    break;
/* Here PAT points to the final \'\'. */
*pat = '\0';
 vlen = pat - *valp;
 if (vlen == 0)
    *valp = NULL;
 rcdbuf->vlen = vlen;
 ptr = pat + 1;
}
/* Certain keywords only have a \'\' string. If there is no \'\'
string, then the old getrcskey function assumed that they had
no value, and we do the same. */
{
    char *k;

    if (STREQ (k, RCSDESC) || STREQ (k, "text") || STREQ (k, "log"))
    {
        if (c != '@')
            *valp = NULL;
        rdbuf->ptr = ptr;
        return 1;
    }
    if (! my_whitespace (n))
        break;

while (1)
    {
    char n;

    if (ptr >= ptrend)
    {
        ptr = rcsbuf_fill (rcbuf, ptr, keyp, valp);
        if (ptr == NULL)
            error (1, 0, "EOF in value in RCS file %s",
                rcsbuf->filename);
        ptrend = rcsbuf->ptrend;
    }
    n = *ptr;
    if (n == '\;')
    {
        /* We're done. We already set everything up for this
case above. */
        rdbuf->ptr = ptr + 1;
        return 1;
    }
    if (! my_whitespace (n))
        break;
++ptr;
}

/* The value extends past the ' ' string. We need to undo the
closing of the ' ' done in the default case above. This
case never happens in a plain RCS file, but it can happen
if user defined phrases are used. */
if (rcbuf->vlen != 0)
  *(valp) = ' ';
else
  valp = ptr;
}

/* Here we have a value which is not a simple ' ' string. We need
to gather up everything until the next ' ', including any '@'
strings. VALP points to the start of the value. If RCSBUF->VLEN is not zero,
then we have already read an ' ' string, and PTR points to the data following the ' ' string.
Otherwise, PTR points to the start of the value. */
while (1)
{
  char *start, *psemi, *pat;
  /* Find the ' ' which must end the value. */
  start = ptr;
  while ((psemi = memchr (ptr, ' ', ptrend - ptr)) == NULL)
  {
    int slen;
    /* Note that we pass PTREND as the PTR value to
     rcsbuf_fill, so that we will wind up setting PTR to the
     location corresponding to the old PTREND, so that we
don't search the same bytes again. */
    slen = start - valp;
    ptr = rcsbuf_fill (rcbuf, ptrend, keyp, valp);
    if (ptr == NULL)
      error (1, 0, "ERR in value in RCS file %s", rcsbuf->filename);
    start = valp + slen;
    ptrend = rcsbuf->ptrend;
  }

  /* See if there are any '@'
  pat = memchr (start, '@', psemi - start);

  if (pat == NULL)
  {
    size_t vlen;
    /* We're done with the value. Trim any trailing
     whitespace. */
    rcsbuf->ptr = psemi + 1;
    start = valp;
    while (psemi > start && my_whitespace (psemi[-1]))
      --psemi;
    *psemi = '\0';
    vlen = psemi - start;
    if (vlen == 0)
      valp = NULL;
    rcsbuf->vlen = vlen;
    return 1;
  }

  /* We found an '@' string in the value. We set
  RCSBUF->AT_STRING, which means that we won't be able to
  compress whitespace correctly for this type of value.
  Since this type of value never arises in a normal RCS file,
  this should not be a big deal. It means that if anybody
  adds a phrase which can have both an '@' string and regular
  text, they will have to handle whitespace compression
  */
  rcsbuf->at_string = 1;
  *pat = ' ';
  ptr = pat + 1;

  while (1)
  {
    while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)
    {
      /* Note that we pass PTREND as the PTR value to
       rcsbuf_fill, so that we will wind up setting PTR
       to the location corresponding to the old PTREND, so
       that we don't search the same bytes again. */
    }
ptr = rcsbuf_fill (rcsbuf, ptrend, keyp, valp);
if (ptr == NULL)
    error (1, 0,
        "EOF while looking for end of string in RCS file %s",
        rcsbuf->filename);
ptrend = rcsbuf->ptrend;
}

/* Handle the special case of an ' ' right at the end of
the known bytes. */
if (pat[1] == ' ')
    break;

/* We found an ' ':
++rcsbuf->embedded_at;
ptr = pat + 2;
*/
/* Here PAT points to the final ' ':
*pat = ' ';
ptr = pat + 1;
*/

#define my_whitespace (c) (my_spacetab[(unsigned char)c] != 0)

/* double redundant code in rcsbuf_getkey, rcsbuf_getid, rcsbuf_getstring, rcsbuf_getword. These last three functions were
all created by hacking monstrous swaths of code from rcsbuf_getkey, and some engineering would make the code easier to read and
maintain.
Note that the extreme hair in rcsbuf_getkey is because profiling
statistics show that it was worth it.
We probably could be processing “hardlinks” by first calling
rcsbuf_getkey, and breaking up the value afterwards; the code to
break it up would not need to be hacked for speed. This would
remove the need for rcsbuf_getword, rcsbuf_getid, and
rcsbuf_getstring, as the other calls are easy to remove. */

/* Read an 'id' (in the sense of rcsfile(5)) from RCSBUF, and store in IDP. */

static int
rcsbuf_getid (rcsbuf, idp)
struct rcsbuffer *rcsbuf;
char **idp;
{
    register const char *const my_spacetab = spacetab;
    register char +ptr, +ptrend;
    char c;

#define my_whitespace (c) (my_spacetab[(unsigned char)c] != 0)

    rcsbuf->vlen = 0;
    rcsbuf->vstring = 0;
    rcsbuf->embedded_at = 0;
    ptr = rcsbuf->ptr;
    ptrend = rcsbuf->ptrend;

    /* Sanity check. */
    if (ptr < rcsbuf->buffer || ptr > rcsbuf->buffer + rcsbuf->buffer_size)
        abort ();

    /* If the pointer is more than RCSBUF_BUFSIZE bytes into the
    buffer, move back to the start of the buffer. This keeps the
    buffer from growing indefinitely. */
    if (ptr - rcsbuf->buffer >= RCSBUF_BUFSIZE)
    {
        int len;
        len = ptrend - ptr;
        /* Sanity check: we don’t read more than RCSBUF_BUFSIZE bytes
        at a time, so we can’t have more bytes than that past PTR. */
        if (len > RCSBUF_BUFSIZE)
            abort ();
/ Update the POS field, which holds the file offset of the first byte in the RCSBUF_BUFFER buffer. */
rcbuf->pos += ptr - rcbuf.buffer;
memcpy(rcbuf.buffer, ptr, len);
ptr = rcbuf.buffer;
ptrend = ptr + len;
rcbuf->ptrend = ptrend;
}

/* Skip leading whitespace. */
while (1)
{
    if (ptr >= ptrend)
    {
        ptr = rcbuf.fill(rcbuf, ptr, (char **) NULL, (char **) NULL);
        if (ptr == NULL)
            return 0;
        ptrend = rcbuf->ptrend;
    }
    c = *ptr;
    if (!my_whitespace(c))
        break;
    ++ptr;
}

/* We've found the start of the key. */
*idp = ptr;
if (c == ';')
{
    while (1)
    {
        ++ptr;
        if (ptr >= ptrend)
        {
            ptr = rcbuf.fill(rcbuf, ptr, idp, (char **) NULL);
            if (ptr == NULL)
                error(1, 0, "EOF in key in RCS file "
                      rcbuf->filename);
            ptrend = rcbuf->ptrend;
        }
        c = *ptr;
        if (c == ';' || my_whitespace(c))
            break;
    }
}

/* Here *IDP points to the id in the buffer, C is the character we found at the end of the key, and PTR points to the location in the buffer where we found C. We may not set *PTR to \0, because it may overwrite a terminating semicolon. The calling function must copy and terminate the id on its own. */
rcbuf->ptr = ptr;
return 1;

#undef my_whitespace

/* Read an RCS */

static int rcbuf_getstring(rcbuf, strp)
{
    register const char * const my_spacetab = spacetab;
    register char *ptr, *ptrend;
    char *pat;
    size_t vlen;
    char c;
#define my_whitespace(c) (my_spacetab[(unsigned char)c] != 0)
    rcbuf->vlen = 0;
    rcbuf->at_string = 0;
    rcbuf->embedded_at = 0;
    ptr = rcbuf->ptr;
    ptrend = rcbuf->ptrend;
    /* Sanity check. */
    if (ptr < rcbuf.buffer || ptr > rcbuf.buffer + rcbuf.buffer.size)
        abort();
        }
/* If the pointer is more than RCSBUF_BUFSIZE bytes into the
buffer, move back to the start of the buffer. This keeps the
buffer from growing indefinitely. */
if (ptr - rcsbuf.buffer >= RCSBUF_BUFSIZE)
{
  int len;
  len = ptrend - ptr;

  /* Sanity check; we don't read more than RCSBUF_BUFSIZE bytes
   at a time, so we can't have more bytes than that past PTR. */
  if (len > RCSBUF_BUFSIZE)
    abort();

  /* Update the POS field, which holds the file offset of the
   first byte in the RCSBUF_BUFFER buffer. */
  rcsbuf->pos += ptr - rcsbuf.buffer;
  memcpy (rcsbuf.buffer, ptr, len);
  ptr = rcsbuf.buffer;
  ptrend = ptr + len;
  rcsbuf->ptrend = ptrend;
}

/* Skip leading whitespace. */
while (1)
{
  if (ptr == ptrend)
  {
    ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
    if (ptr == NULL)
      error (1, 0, "unexected end of file reading %s",
           rcsbuf->filename);
    ptrend = rcsbuf->ptrend;
  }
  c = *ptr;
  if (! my_whitespace (c))
    break;
  ++ptr;
}

/* PTR should now point to the start of a string. */
if (c == ' ')
  error (1, 0, "expected @-string at '%c' in %s",
       c, rcsbuf->filename);

rcbuf->at_string = 1;
++ptr;

*strp = ptr;

while (1)
{
  while ((pat = memchr (ptr, ' ', ptrend - ptr)) == NULL)
  {
    /* Note that we pass PTREND as the PTR value to
      rcsbuf_fill, so that we will wind up setting PTR to
      the location corresponding to the old PTREND, so
      that we don't search the same bytes again. */
    ptr = rcsbuf_fill (rcsbuf, ptrend, NULL, strp);
    if (ptr == NULL)
      error (1, 0,
            "EOF while looking for end of string in RCS file %s",
            rcsbuf->filename);
    ptrend = rcsbuf->ptrend;
  }

  /* Handle the special case of an '@' right at the end of
   the known bytes. */
  if (pat + 1 >= ptrend)
  {
    /* Note that we pass PAT, not PTR, here. */
    pat = rcsbuf_fill (rcsbuf, pat, NULL, strp);
    if (pat == NULL)
    {
      /* EOF here is OK; it just means that the last
       character of the file was an '@' terminating a
       value for a key type which does not require a
       trailing '@'. */
      pat = rcsbuf->ptrend - 1;
    }
    ptrend = rcsbuf->ptrend;
  }
/* Note that the value of PTR is bogus here. This is OK, because we don't use it. */

if (pat + 1 >= ptrend || pat[1] != '\0')
    break;
/* We found an */
    +rcbuf->embedded_at;
    ptr = pat + 2;

/* Here PT points to the final */
    *pat = '\0';
    vlen = pat - *strp;
    if (vlen == 0)
        *strp = NULL;
    rcbuf->vlen = vlen;
    rcbuf->ptr = pat + 1;
    return 1;

#define my_whitespace(c) (my_spacetab[(unsigned char)c] != 0)

static int
rcbuf_getword (rcbuf, wordp)
struct rcsbuffer *rcbuf;
char **wordp;
{
    register const
        char *const my_spacetab = spacetab;
    register char *ptr, *ptrend;
    char c;

    rcsbuf->vlen = 0;
    rcsbuf->at_string = 0;
    rcsbuf->embedded_at = 0;
    ptr = rcsbuf->ptr;
    ptrend = rcsbuf->ptrend;
/* Sanity check. */
    if (ptr < rcsbuf->buffer || ptr > rcsbuf->buffer + rcsbuf->buffer_size)
        abort (1);
/* If the pointer is more than RCSBUF_BUFSIZE bytes into the
   buffer, move back to the start of the buffer. This keeps the
   buffer from growing indefinitely. */
    if (ptr - rcsbuf->buffer >= RCSBUF_BUFSIZE)
    {
        int len;

        len = ptrend - ptr;
/* Sanity check: we don't read more than RCSBUF_BUFSIZE bytes
   at a time, so we can't have more bytes than that past PTR. */
        if (len > RCSBUF_BUFSIZE)
            abort (1);
/* Update the POS field, which holds the file offset of the
   first byte in the RCSBUF_BUFFER buffer. */
        rcsbuf->pos += ptr - rcsbuf->buffer;
        memcpy (rcbuf->buffer, ptr, len);
        ptr = rcsbuf->buffer;
        ptrend = ptr + len;
        rcsbuf->ptrend = ptrend;
    }
/* Skip leading whitespace. */

    while (1)
    {
        if (ptr >= ptrend)
        {
            ptr = rcsbuf->fill (rcbuf, ptr, (char **) NULL, (char **) NULL);
            if (ptr == NULL)
                error (1, 0, "unexpected end of file reading %s",
                    rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
        }
c = *ptr;
    if (! my_whitespace (c))
        break;
    ++ptr;
}
/* If we have reached ';', there is no value. */
if (c == ';')
    {  
        wordp = NULL;
        *ptr++ = '\0';
        rcsbuf->ptr = ptr;
        rcsbuf->vlen = 0;
        return 1;
    }
/* PTR now points to the start of a value. Find out whether it is 
a num, an id, a string or a colon. */
if (c == ':')
    {  
        wordp = ptr++;
        rcsbuf->ptr = ptr;
        rcsbuf->vlen = 1;
        return 1;
    }
if (c == '@')
    {  
        char *pat;
        size_t vlen;
        /* Optimize the common case of a value composed of a single 
           * /
        rcsbuf->string = 1;
        ++ptr;
        *wordp = ptr;
        while (1)  
            {  
                while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)  
                    {  
                        /* Note that we pass PTREND as the PTR value to 
                            rcsbuf_fill, so that we will wind up setting PTR to 
                            the location corresponding to the old PTREND, so 
                            that we don't search the same bytes again. */
                            ptr = rcsbuf_fill (rcsbuf, ptrend, NULL, wordp);  
                        if (ptr == NULL)  
                            {  
                                error (1, 0,  
                                    "EOF while looking for end of string in RCS file %s",  
                                    rcsbuf->filename);  
                                ptrend = rcsbuf->ptrend;
                            }  
                        /* Handle the special case of an '@' right at the end of 
                           the known bytes. */
                        if (pat + 1 >= ptrend)  
                            {  
                                /* Note that we pass PAT, not PTR, here. */
                                pat = rcsbuf_fill (rcsbuf, pat, NULL, wordp);  
                                if (pat == NULL)  
                                    {  
                                        /* EOF here is OK; it just means that the last 
                                            character of the file was an '@' terminating a 
                                            value for a key type which does not require a 
                                            trailing '@'. */
                                        pat = rcsbuf->ptrend - 1;
                                    }
                                ptrend = rcsbuf->ptrend;
                            }  
                        /* Note that the value of PTR is bogus here. This is 
                           OK, because we don't use it. */
                        if (pat + 1 >= ptrend || pat[1] != '@')
                            break;
                    }  
                /* Here PAT points to the final '@' */
                *pat = '\0';
vlen = pat - *wordp;
    if (vlen == 0)
        *wordp = NULL;
    rcsbuf->vlen = vlen;
    rcsbuf->ptr = pat + 1;
    return 1;
} /* C is neither ':', ';' nor ',', so it should be the start of a num or an id. Make sure it is not another special character. */
    if (c == '$' || c == '.' || c == ',')
    {
        error (1, 0, "illegal special character in RCS field in %s",
               rcsbuf->filename);
    }
    *wordp = ptr;
    while (1)
    {
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, wordp);
            if (ptr == NULL)
                error (1, 0, "unexpected end of file reading %s",
                       rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
        }
        /* Legitimate ID characters are digits, dots and any "graphic printing character that is not a special." This test ought */
        c = ++ptr;
        if (isprint (c) && c != ';' && c != '.' && c != '@' && c != ':')
        {
            ++ptr;
            continue;
        }
        break;
    }
    /* PTR points to the last non-id character in this word, and C is the character in its memory cell. Check to make sure that it is a legitimate word delimiter – whitespace or semicolon. */
    if (c == ';' || my_whitespace (c))
    {
        rcsbuf->vlen = ptr - *wordp;
        rcsbuf->ptr = ptr;
        return 1;
    }
    error (1, 0, "illegal special character in RCS field in %s",
           rcsbuf->filename);
    /* Shut up compiler warnings. */
    return 0;
#undef my_whitespace

/* Read an RCS revision number from an RCS file. This sets *REVP to point to the revision number; it will point to space that is managed by the rcsbuf functions, and is only good until the next call to rcsbuf_getkey or rcsbuf_getrevnum. */

static int rcsbuf_getrevnum (rcsbuf, revp)
{
struct rcsbuffer *rcsbuf;
char **revp;

    char *ptr, *ptrend;
    char c;
    ptr = rcsbuf->ptr;
    ptrend = rcsbuf->ptrend;

    *revp = NULL;
    /* Skip leading whitespace. */
    while (1)
    {
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
        }
if (ptr == NULL)
    return 0;
    ptrend = rcsbuf->ptrend;
}
    c = *ptr;
    if (! whitespace(c))
        break;
    ++ptr;

if (! isdigit(c) && c != '0')
    error(1, 0, "unexpected '%c' reading revision number in RCS file %s", c, rcsbuf->filename);
    *revp = ptr;
    do {
        ++ptr;
        if ((ptr >= ptrend)
            { ptrend = rcsbuf_fill (rcsbuf, ptr, revp, (char **) NULL);
            if (ptr == NULL)
                error(1, 0, "unexpected EOF reading revision number in RCS file %s", rcsbuf->filename);
                ptrend = rcsbuf->ptrend;
            }

        c = *ptr;
        while (isdigit(c) || c == '0');
        if (! whitespace(c))
            error(1, 0, "unexpected 'E' reading revision number in RCS file %s", rcsbuf->filename);
            *ptr = '\0';

    } rcsbuf->ptr = ptr + 1;
    return 1;

    /* Fill RCSBUF_BUFFER with bytes from the file associated with RCSBUF, updating PTR and the PTREND field. If KEYP and *KEYP are not NULL, then *KEYP points into the buffer, and must be adjusted if the buffer is changed. Likewise for VALP. Returns the new value of PTR, or NULL on error. */

    static char *
    rcsbuf_fill (rcsbuf, ptr, keyp, valp)
    struct rcsbuf *rcsbuf;
    char *ptr;
    char **keyp;
    char **valp;
    { int got;

        if (rcsbuf->ptrend - rcsbuf_buffer + RCSBUF_BUFSIZE > rcsbuf_buffer_size)
            { int poff, koff, voff;
                poff = ptr - rcsbuf_buffer;
                koff = rcsbuf->ptrend - rcsbuf_buffer;
                if (keyp != NULL & & *keyp != NULL)
                    koff = *keyp - rcsbuf_buffer;
                if (valp != NULL & & *valp != NULL)
                    voff = *valp - rcsbuf_buffer;

                expand_string (&rcsbuf_buffer, rcsbuf_buffer_size,
                                rcsbuf_buffer_size + RCSBUF_BUFSIZE);

                ptr = rcsbuf_buffer + poff;
                rcsbuf->ptrend = rcsbuf_buffer + poff;
                if (keyp != NULL & & *keyp != NULL)
                    *keyp = rcsbuf_buffer + koff;
                if (valp != NULL & & *valp != NULL)
                    *valp = rcsbuf_buffer + voff;

            }

        got = fread(rcsbuf->ptrend, 1, RCSBUF_BUFSIZE, rcsbuf->fp);
        if (got == 0)
            { if (errno (rcsbuf->fp))
                error(1, errno, "cannot read %s", rcsbuf->filename);
                return NULL;
            }
1890
    rcsbuf->ptrend += got;

    return ptr;
}

/* Copy the value VAL returned by rcsbuf_getkey into a memory buffer, returning the memory
buffer. Polish the value like rcsbuf_valpolish, q.v. */

1900 static char *
rcsbuf_valcopy(rcsbuf, val, polish, lenp)
struct rcsbuffer *rcsbuf;
char *val;
int polish;
size_t *lenp;
{
    size_t vlen;
    int embedded_at;
    char *ret;
    if (val == NULL)
    {
        if (lenp != NULL)
            *lenp = 0;
        return NULL;
    }
    vlen = rcsbuf->vlen;
    embedded_at = rcsbuf->embedded_at;
    ret = xmalloc(vlen - embedded_at + 1);
    if (rcsbuf->at_string ? embedded_at == 0 : ! polish)
    {
        /* No special action to take. */
        memcpy(ret, val, vlen);
        if (lenp != NULL)
            *lenp = vlen;
        return ret;
    }
rcsbuf_valpolish_internal(rcsbuf, ret, val, lenp);
    return ret;
}

/* Polish the value VAL returned by rcsbuf_getkey. The POLISH parameter is non-zero if multiple
embedded whitespace characters should be compressed into a single whitespace character. Note that
leading and trailing whitespace was already removed by rcsbuf_getkey. Within an ’ ’ string, pairs of
’ ’ characters are compressed into a single ’ ’ character regardless of the value of
POLISH. If LENP is not NULL, set *LENP to the length of the value. */

1940 static void
rcsbuf_valpolish(rcsbuf, val, polish, lenp)
struct rcsbuffer *rcsbuf;
char *val;
int polish;
size_t *lenp;
{
    if (val == NULL)
    {
        if (lenp != NULL)
            *lenp = 0;
        return;
    }
    if (rcsbuf->at_string ? rcsbuf->embedded_at == 0 : ! polish)
    {
        /* No special action to take. */
        if (lenp != NULL)
            *lenp = rcsbuf->vlen;
        return;
    }
rcsbuf_valpolish_internal(rcsbuf, val, val, lenp);
}

/* Internal polishing routine, called from rcsbuf_valcopy and
rcsbuf_valpolish. */

1970 static void
rcsbuf_valpolish_internal(rcsbuf, to, from, lenp)
struct rcsbuffer *rcsbuf;
char *to;
const char *from;
size_t *lenp;
{
size_t len;

len = rcsbuf->vlen;

if (! rcsbuf->at_string)
{
    char *orig_to;
    size_t clen;
    orig_to = to;

for (clen = len; clen > 0; **from, --clen)
{
    char c;
    c = *from;
    if (isspace (c))
    {
        /* Note that we know that clen can not drop to zero
           while we have whitespace, because we know there is
           no trailing whitespace. */
        while (isspace (from[1]))
        {
            ++from;
            --clen;
        }
        c = ' ';  
    }
    *to++ = c;
}

*to = '\0';

if (lenp != NULL)
    *lenp = to - orig_to;
else
{
    const char *orig_from;
    char *orig_to;
    int embedded_at;

    orig_from = from;
    orig_to = to;
    embedded_at = rcsbuf->embedded_at;

    if (lenp != NULL)
        *lenp = len - embedded_at;

for (clen = len; clen > 0; ++from, --clen)
{
    char c;
    c = *from;
    *to++ = c;
    if (c == ' ')
    {
        ++from;
        while (isspace (from[1]))
        {
            ++from;
            --clen;
        }
        if (c == ' ')
        {
            ++from;
        }
        else
        {
            memcpy (to, from + 1, clen - 1);
            from += clen;
            to += clen - 1;
            break;
        }
    }
    
    /* Sanity check. */
    if ((from != orig_from)     ||
        (clen == 0))
    abort (1);

    clen = --clen;
    embedded_at = --embedded_at;
    if (embedded_at == 0)
    {
        /* We've found all the embedded ' ' characters.
           We can just memcpy the rest of the buffer after
           this ' ' character. */
        if (orig_to != orig_from)
            memcpy (to, from + 1, clen - 1);
        else
            memmove (to, from + 1, clen - 1);
        from += clen;
        to += clen - 1;
        break;
    }
}

    /* Sanity check. */
    if (from != orig_from + len)
        abort (1);
}
}
static void rcsbuf_fclose (rcsbuf)
{
    if (rcsbuf != NULL)
        close (rcsbuf->fp);
    return;
}

 RCSNode *rcsbuf_cache (rcs, rcsbuf)
2110
    RCSNode *rcs;
 struct rcsbuffer *rcsbuf;
2120
    if (rcsbuf != NULL)
        rcsbuf_cache_close ();
    if (rcsfail (rcsbuf) != 0)
        error (0, errno, "cannot close %s", rcsbuf->filename);
    rcsbuf_close (&cached.rcsbuf);
    free_rcsnode (&cached.rcs);
    cached.rcs = NULL;
    return;
}

/* Open an rcsbuffer for RCS, getting it from the cache if possible. Set *FPP to the file, and *RCSBUFP to the rcsbuf. The file should be put at position POS. */
2140
rcsbuf_cache_open (rcs, pos, pfp, prcsbuf)
    RCSNode *rcs;
    long pos;
    FILE **pfp;
 struct rcsbuffer *prcsbuf;
2150
    if (rcsfail (rcsbuf) != pos)
    {
        if (fseek (cached.rcsbuf->fp, pos, SEEK_SET) != 0)
            error (1, 0, "cannot fseek RCS file %s", cached.rcsbuf.filename);
        cached.rcsbuf.pfp = rcsbuf->fp;
        cached.rcsbuf.pend = rcsbuf->buffer;
        cached.rcsbuf.pos = pos;
        pfp = cached.rcsbuf.fp;
    }
When RCS parse opens a file using fopen case, it frees the
filename we cached in CACHED_RCS_BUF and stores a new
file name in RCS Tập. We avoid problems here by always
copying the filename over. FIXME: This is hackish. */
cached_rcsbuf.filename = rcs->path;

/* Removing RCS from the cache removes a reference to it. */
−−rcs->refcount;
if (rcs->refcount <= 0)
    error (1, 0, "rcsbuf_cache_open: internal error");
else
    if (cached_rcs != NULL)
        rcsbuf_cache_close();
    pfp = CVS_FOPEN (rcs->path, FOPEN_BINARY_READ);
if (pfp == NULL)
    error (1, 0, "unable to reopen '%s'", rcs->path);
    fseek (pfp, pos, SEEK_SET);
rcsbuf_open (prcsbuf, pfp, rcs->path, pos);

static void
do_symbols (list, val)
List *list;
char *val;
{
    Node *p;
    char *cp = val;
    char *tag, *rev;
    for (;
        /* skip leading whitespace */
        while (whitespace (*cp))
            cp++;
    /* split it up into tag and rev */
    tag = cp;
    cp = strchr (cp, '\');
        *cp++ = '\0';
    rev = cp;
    while (whitespace (*cp) & & *cp != '\0')
        cp++;
    if (*cp != '\0')
        *cp++ = '\0';
    /* make a new node and add it to the list */
    p = getnode ();
    p->key = xstrdup (tag);
    p->data = xstrdup (rev);
    (void) addnode (list, p);
}

static void
do_locks (list, val)
List *list;
char *val;
{
    Node *p;
    char *cp = val;
    char *user, *rev;
    for (;
        /* skip leading whitespace */
```c
while (whitespace (*cp))
    cp++;
/* if we got to the end, we are done */
if (*cp == '\0')
    break;
/* split it up into user and rev */
user = cp;
rev = strchr (cp, ':');
*cp++ = '\0';
while (whitespace (*cp) && *cp != '\0')
    *cp++ = '\0';
/* make a new node and add it to the list */
p = getnode ();
p->key = xstrdup (rev);
p->data = xstrdup (user);
( void ) addnode (list, p);
}
/* process the branches list of a revision delta */
static void
do_branches (list, val)
List *list;
char *val;
{
    Node *p;
    char *cp = val;
    char *branch;
    for (;
        /
        * skip leading whitespace */
        while (whitespace (*cp))
            cp++;
/* if we got to the end, we are done */
if (*cp == '\0')
    break;
/* find the end of this branch */
branch = cp;
while (whitespace (*cp) && *cp != '\0')
    *cp++ = '\0';
/* make a new node and add it to the list */
p = getnode ();
p->key = xstrdup (branch);
p->data = xstrdup (val);
( void ) addnode (list, p);
}
/* process the branches list of a revision delta */
static void
do_remote_branches (list, val)
List *list;
char *val;
{
    Node *p;
    char *cp = val;
    char *branch;
    if (val == NULL)
        return;
    for (;
        /
        * skip leading whitespace */
        while (whitespace (*cp))
            cp++;
/* if we got to the end, we are done */
if (*cp == '\0')
    break;
/* find the end of this branch */
branch = cp;
while (whitespace (*cp) && *cp != '\0')
    *cp++ = '\0';
```
if (*p != '\0') {
    *p++ = '\0';
    /* make a new node and add it to the list */
    p = getnode();
    p->key = xstrdup(branch);
    (void) addnode(list, p);
}
}

/*
 * Version Number
 *
 * Returns the requested version number of the RCS file, satisfying tags and/or
 * dates, and walking branches, if necessary.
 *
 * The result is returned; null-string if error.
 */
char *
RCS_getversion (rcs, tag, date, force_tag_match, simple_tag)
RCSNode *rcs;

char *tag;
char *date;
int force_tag_match;
int *simple_tag;
{
    if (simple_tag != NULL)
        *simple_tag = 0;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if (tag && date) {
        char *branch, *rev;
        if (! RCS_nodeisbranch (rcs, tag))
            {/* We can't get a particular date if the tag is not a
               branch. */
                return NULL;
            }
        /* Work out the branch. */
        if (! isdigit (tag[0]))
            branch = RCS_whatbranch (rcs, tag);
        else
            branch = xstrdup (tag);

        /* Fetch the revision of branch as of date. */
        rev = RCS_getdatebranch (rcs, date, branch);

        free (branch);
        return (rev);
    }
    else if (tag)
        return (RCS_gettag (rcs, tag, force_tag_match, simple_tag));
    else if (date)
        return (RCS_getdate (rcs, date, force_tag_match));
    else
        return (RCS_head (rcs));

}

typedef struct {
    char* number;
    char* branch;
} findremotebranchcl;

int findremotebranchin (Node* node, void* closure)
{
    findremotebranchcl* data = (findremotebranchcl*) closure;
    if (data->branch != NULL)
        return 0;
    if (strncmp (node->key, data->number, strlen (data->number)) != 0) {
        data->branch = xstrdup (node->key);
    }
    return 0;
}

char *
RCS_getremoteversion (info, rcs, tag, local_tag)
struct file_info* info;
RCSNode *rcs;
char *tag;
char* local_tag;
{
    *local_tag = NULL;
    RCS_fully_parse (rcs);
    assert (rcs != NULL);
    /*
if (isdigit(tag[0])) {
    /* If it's a digit, it has to be a symbolic tag. Verify that it's a remote tag,
     * to save us some errors later, and return it */
    if (strchr(tag, ':') != 0) {
        return xstrdup(tag);
    } else {
        return NULL;
    }
} else {
    /* If it's a digit, see if the tag is on one of the remote branches. Start by finding out its
     * branch magic number */

    if ((numdots(tag) % 2 == 0) {
        /* Can't be a valid tag unless it has an odd number of dots (maybe we should allow branch tags?) */
        return NULL;
    } else {
        char *remotetag = xstrdup(tag);
        char *base = NULL;
        char *nextdot = remotetag;
        char *lastdot = NULL;
        char *beforelastdot = remotetag;

        int branchnum;

        for (; nextdot != NULL; nextdot = strchr(nextdot, '.')) {
            beforelastdot = lastdot + 1;
            lastdot = nextdot;
            nextdot += 1;
        }

        /* Didn't find any dots? That's no good */

        if (lastdot == NULL) {
            return NULL;
        }

        /* Kill the last component (we don't care) */
        *lastdot = '\0';

        /* Read the branch number */
        branchnum = atoi(beforelastdot);

        /* Remove the branch number */
        *beforelastdot = '\0';

        base = strdup(remotetag);

        /* Put in the magic branch number - note that this never
         * overflows the buffer because magic branch is one char long */
        sprintf(remotetag + strlen(remotetag), "%d.%d", RCS_MAGIC_BRANCH, branchnum);

        /* Now we know the remote tag. Go down the list of remote
         * branches and see if any one of them matches... */

        
        Node *versionNode = NULL;
        RCSVers *version = NULL;
        find_remote_branch = { NULL, NULL };;
        remotetag = xrealloc(remotetag, strlen(remotetag) + 2);
        strcat(remotetag, "\0");
        closure.number = remotetag;
        base[strlen(base) - 1] = '\0';

        versionNode = findnode(rcs->versions, base);
        if (versionNode != NULL) {
            version = (RCSVers*) (versionNode -> data);
        }

        if (version != NULL) {
            walklist(version -> remote_branches, find_remote_branch, fn, &closure);
        } free(remotetag);
        free (base);

        /* According to the RCS file, this is a remote revision.
         * However, it's possible that we already have it. Check whether we do */

        if (closure.branch == NULL) {
            return NULL;
        } else {
            char *local_rev = revision_available_locally (finfo, rcs, tag);
            if (local_rev != NULL) {
                local_tag = local_rev;
                return NULL;
            } else {
                return closure.branch;
            }
        }
    }
}

char *revision_available_locally (struct file_info *finfo, RCSNode *rcs, char *remotRev)
2520  {
  char* result = 0;
  FILE* remotes_list;
  char* line;
  int line_length;
  int line_char_allocated;

  /* Try to find it in the Remotes file */
  remotes_list = fopen (CVSADM_REMOTES, "r");
  while ((line_length = getline (line, &line, &line_char_allocated, remotes_list)) > 0) {
    char* file = NULL;
    char* rev = NULL;
    char* data = NULL;

    file = line;
    rev = strchr (file, '\'/');
    if (rev == NULL) continue;
    *rev = '\0';
    rev++;

    data = strchr (rev, '\'/');
    if (data == NULL) continue;
    *data = '\0';
    data++;
    *strchr (data, '\"') = '\0';

    if ((strcmp (finfo->file, file) == 0) && (strcmp (rev, remote_rev) == 0)) {
      /* File and revision match */
      char* fullname = xmalloc (strlen (CVSADM) + strlen (data) + 5);
      sprintf (fullname, "%s/%s", CVSADM, data);
      RCS_checkin (rcs, fullname, "Remote revision", "1.1.3", RCSFLAGSKEEPFILE);
      return RCS_getbranch (rcs, xstrdup (CVSREMOTE_BRANCH), 1);
      /* We found where the revision data is stored... */
      break;
    }
    return result;
  }

  /* Get existing revision number corresponding to tag or revision.
   * Similar to RCS_gettag but less interpretation imposed.
  *
  * For example:
  * - If tag designates a magic branch, RCS_tag2rev
  * returns the magic branch number.
  * - If tag is a branch tag, returns the branch number, not
  * the revision of the head of the branch.
  * - If tag or revision is not valid or does not exist in file,
  * exit with error.
  */
  char * RCS_tag2rev (rcs, tag)
    RCSNode *rcs;
    char *tag;
  {
    char *rev, *pa, *pb;
    int i;
    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
      RCS_reparserecfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);

    /* If a valid revision, try to look it up */
    if (RCS_valid_rev (tag)) {
      /* Make a copy so we can scribble on it */
      rev = xstrdup (tag);
      /* If revision exists, return the copy */
      if (RCSexist_rev (rcs, tag))
        return rev;

      /* Nope, none such. If tag is not a branch we're done. */
      i = numdots (rev);
      if (i & 1) == 1 )
        {
          pa = strchr (rev, '\'.');
          if [i == 1] || *(pa-1) != RCS_MAGIC_BRANCH || *(pa-2) != '\."
            {
              free (rev);
            }
If tag is "HEAD", special case to get head RCS revision

If tag is "HEAD", special case to get head RCS revision

RCS\_check\_tag (tag); /* exit if not a valid tag */

/* If tag is "HEAD", special case to get head RCS revision */
if (tag && (strcmp (tag, Tag\_HEAD) == 0))
    return (RCS\_head (rcs));

/* If valid tag let translate\_symtag say yes or no. */
rev = translate\_symtag (rcs, tag);

if (rev)
    return rev;

error (1, 0, "revision \"%s\" does not exist", tag);
/* NOT REACHED – error (1 ... ) does not return here */
return 0;

/* Find the revision for a specific tag.
 * If force\_tag\_match is set, return NULL if an exact match is not
 * possible otherwise return RCS\_head (). We are careful to look for
 * and handle “magic” revisions specially.
 * If the matched tag is a branch tag, find the head of the branch.
 * Returns pointer to newly malloc’ed string, or NULL.
 */

char *
RCS\_gettag (rcs, symtag, force\_tag\_match, simple\_tag)
RCSNode *rcs;
char *symtag;
int force\_tag\_match;
int *simple\_tag;
{
    char *tag = symtag;
    int tag\_allocated = 0;

    if (simple\_tag != NULL)
        *simple\_tag = 0;

    /* make sure we have something to look at ... */
    assert (rcs != NULL);

    /* XXX this is probably not necessary, -jtc */
    if ((rcs->flags & PARTIAL)
        RCS\_parse\_cfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);

    /* If tag is "HEAD", special case to get head RCS revision */
    if (tag && (strcmp (tag, Tag\_HEAD) || *tag == \"\"))
        return ((char *) NULL); /* head request for removed file */
    else
        return (RCS\_head (rcs));

    /* If we got a symbolic tag, resolve it to a numeric */
    if (isdigit (tag[0]))
        {
            char *version;

            /* If version != NULL */
            version = translate\_symtag (rcs, tag);
            if (version != NULL)
                {
                    int dots;
                    char *magic, *branch, *cp;

                    printf ("\"%s\"", version);
                    return 0;
                }
tag = version;
tag_allocated = 1;

/* If this is a magic revision, we turn it into either its
 * physical branch equivalent (if one exists) or into
 * its base revision, which we assume exists.
 */
dots = numdots (tag);
if (dots > 2 & & (dots & 1) != 0)
{
    branch = strrchr (tag, '.');
    cp = branch+i-1;
    while (*cp != '.')
        cp--;
    /* see if we have .magic-branch. (".0.") */
    magic = xmalloc (strlen (tag)+1);
    (void) sprintf (magic, "%.d.", RCS_MAGIC_BRANCH);
    if (strncmp (magic, cp, strlen (magic)) == 0)
    {
        /* it's magic. See if the branch exists */
        *cp = '\'0'; /* turn it into a revision */
        (void) sprintf (magic, "%s.%s", tag, branch);
        branch = RCS_getbranch (rcs, magic, 1);
        free (magic);
        if (branch != NULL)
            { free (tag);
              return (branch);
            }    /* make sure it exists */
    } else
        /* return the head or NULL */
    if (force_tag_match)
        return (NULL);
    else
        return (RCS_head (rcs));
}

/* numeric tag processing:
 * 1) revision number - just return it
 * 2) branch number - find head of branch
 */
/* strip trailing dots */
while ([tag[strlen (tag) - 1] == '.']
    tag[strlen (tag) - 1] = '\0';
if ((numdots (tag) & 1) == 0)
{
    char *branch;
    /* we have a branch tag, so we need to walk the branch */
    branch = RCS_getbranch (rcs, tag, force_tag_match);
    if (tag_allocated)
        free (tag);
    return (branch);
} else
    { Node *p;

    /* we have a revision tag, so make sure it exists */
    p = findnode (rcs->versions, tag);
    if (p != NULL)
    { /* We have found a numeric revision for the revision tag.
      * To support expanding the RCS keyword Name, if
      * SIMPLE_TAG is not NULL, tell the the caller that this
      * is a simple tag which co will recognize. FIXME: Are
      * there other cases in which we should set this? In
      * particular, what if we expand RCS keywords internally
      * without calling co? */
        if (simple_tag != NULL)
            simple_tag = 1;
        if (! tag_allocated)
            tag = xstrdup (tag);
        return (tag);
    } else
    {
static int check_magic( RCSNode *rcs )
{
  int rnum = 2;
  int xnum;
  char *xrev = xmalloc( strlen( rcs->rev ) + 14 );
  /* enough for .0.number */
  check_xrev = xrev;

  /* only look at even numbered branches */
  for ( xnum = 2; rnum += 2 ) {
    /* see if the physical branch exists */
    (void) sprintf( xrev, "%s.%d", rcs->rev, xnum );
    xrev = xrealloc( xrev, strlen( xrev ) + 14 );
    test_branch = RCS_getbranch( rcs, xrev, 1 );
    if ( test_branch != NULL ) /* it did, so keep looking */
    {
      free( test_branch );
      continue;
    }

    /* now, create a "magic" revision */
    (void) sprintf( xrev, "%s.%d.%d", rcs->rev, xnum, 1 );
    /* walk the symbols list to see if a magic one already exists */
    if ( walklist( RCS_symbols( rcs ), check_magic_proc, NULL ) != 0 )
      continue;

    /* we found a free magic branch. Claim it as ours */
    return( xrev );
  }

  /* walklist proc to look for a match in the symbols list. */
  return( 0 );
}

/* Given an RCSNode, returns non-zero if the specified revision number */
/* or symbolic tag resolves to a "branch" within the rcs file. */
/* FIXME: this is the same as RCS_pnodeisbranch except for the special */
/* case for handling a null rcsnode. */
static int check_magic_proc( p, closure )
{
  Node *p;
  void *closure;
  if ( STREQ( check_xrev, p->data ) )
    return( 1 );
  else
    return( 0 );
}

/* Return a "magic" revision as a virtual branch off of REV for the RCS file. */
/* A "magic" revision is one which is unique in the RCS file. By unique, I */
/* mean we return a revision which: */
/* - has a branch of 0 (see rcs.h RCS_MAGICBRANCH) */
/* - has a revision component which is not an existing magic revision */
/* - is an even-numbered revision, to avoid conflicts with vendor branches */
/* The first point is what makes it "magic". */
/* */
/* As an example, if we pass in 1.37 as REV, we will look for an existing */
/* branch called 1.37.2. If it did not exist, we would look for an */
/* existing symbolic tag with a numeric part equal to 1.37.0.2. If that */
/* didn't exist, then we know that the 1.37.2 branch can be reserved by */
/* creating a symbolic tag with 1.37.0.2 as the numeric part. */
/* */
/* This allows us to fork development with very little overhead - just a */
/* symbolic tag is used in the RCS file. When a commit is done, a physical */
/* branch is dynamically created to hold the new revision. */

/* Note: We assume that REV is an RCS revision and not a branch number. */
/* */
static char *check_xrev;
char *RCS_magicrev( rcs, rev )
RCSNode *rcs;
char *rev;
{
  int rnum,
  char *xrev, *test_branch;

  xrev = xmalloc( strlen( rev ) + 14 );
  /* enough for .0.number */
  check_xrev = xrev;

  /* only look at even numbered branches */
  for ( rnum = 2; rnum += 2 )
  {
    /* see if the physical branch exists */
    (void) sprintf( xrev, "%s.%d", rev, rnum );
    test_branch = RCS_getbranch( rcs, xrev, 1 );
    if ( test_branch != NULL ) /* it did, so keep looking */
    {
      free( test_branch );
      continue;
    }

    /* now, create a "magic" revision */
    (void) sprintf( xrev, "%s.%d.%d", rev, rnum, 1 );
    /* walk the symbols list to see if a magic one already exists */
    if ( walklist( RCS_symbols( rcs ), check_magic_proc, NULL ) != 0 )
      continue;

    /* we found a free magic branch. Claim it as ours */
    return( xrev );
  }

  /* walklist proc to look for a match in the symbols list. */
  return( 0 );

  if ( tag_allocated )
  free( tag );
  if ( force_match )
    return( NULL );
  else
    return( RCS_head( rcs ) );
}
int RCSnodeisbranch (rcs, rev)
RCSNode *rcs;
const char *rev;
{
    /* numeric revisions are easy - even number of dots is a branch */
    if (isdigit (*rev))
        return ((numdots (rev) & 1) == 0);

    /* assume a revision if you can't find the RCS info */
    if (rcs == NULL)
        return (0);

    /* now, look for a match in the symbols list */
    return (RCSnodeisbranch (rcs, rev));
}

int RCSnodeisbranch (rcs, rev)
RCSNode *rcs;
const char *rev;
{
    int dots;
    char *version;

    assert (rcs != NULL);

    /* numeric revisions are easy - even number of dots is a branch */
    if (isdigit (*rev))
        return ((numdots (rev) & 1) == 0);

    version = translateSymtag (rcs, rev);
    if (version == NULL)
        return (0);
    dots = numdots (version);

    if ((dots & 1) == 0)
    {
        free (version);
        return (1);
    }

    /* got a symbolic tag match, but it's not a branch; see if it's magic */
    if (dots > 2)
    {
        char *magic;
        char *branch = strchr (version, '.');
        char *cp = branch - 1;
        while (*cp != '
')
            cp--;

        /* see if we have .magic-branch. (*.0 .) */
        magic = xmalloc (strlen (version) + 1);
        (void) sprintf (magic, "%s", RCS_MAGIC_BRANCH);
        if (strncmp (magic, cp, strlen (magic)) == 0)
        {
            free (magic);
            free (version);
            return (1);
        }
        free (magic);
        free (version);
    }
    return (0);
}

/* Returns a pointer to malloc'd memory which contains the branch 
   for the specified symbolic tag. Magic branches are handled correctly. */
char * RCSwhatbranch (rcs, rev)
RCSNode *rcs;
const char *rev;
{
    char *version;
    int dots;

    /* assume no branch if you can't find the RCS info */
    if (rcs == NULL)
        return ((char *) NULL);

    /* now, look for a match in the symbols list */
    version = translateSymtag (rcs, rev);
    if (version == NULL)
```c
2970 return ((char *) NULL);
dots = numdots (version);
if ((dots & 1) == 0)
    return (version);

/* get a symbolic tag match, but it's not a branch; see if it's magic */
if (dots > 2)
    {
    char *magic;
    char *branch = strchr (version, ',');
    char *cp = branch + 1;

    while (*cp != ',', ')
        cp -= 1;

    /* see if we have .magic-branch. (*.0.*) */
    magic = xmalloc (strlen (version) + 1);
    if (strcmp (magic, cp, strlen (magic)) == 0)
        {
        /* yep. it's magic. now, construct the real branch */
        *cp = '\0';    /* turn it into a revision */

        (void) sprintf (magic, "%s.%s", version, branch);
        free (version);
        return (magic);
        }
    free (magic);
    free (version);
    }

    return ((char *) NULL);
}

3000 /*
 * Get the head of the specified branch. If the branch does not exist,
 * return NULL or RCS_NULL or depend on force_tag_match.
 * Returns NULL or a newly malloc'd string.
 */
char *RCS_getbranch (rcs, tag, force_tag_match)
RCSNode *rcs;
char *tag;
int force_tag_match;
{
    Node *p, *head;
    RCSVers *vn;
    char *xtag;
    char *nextvers;
    char *cp;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

3020 if (rcs->flags & PARTIAL)
    RCS_parsersortfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);

    /* find out if the tag contains a dot, or is on the trunk */
    cp = strchr (tag, '.');

    /* trunk processing is the special case */
    if (cp == NULL)
        {
        xtag = xmalloc (strlen (tag) + 1 + 1); /* +1 for an extra */

        (void) strcpy (xtag, tag);

        (void) strcat (xtag, '*');
        for (cp = rcs->head; cp != NULL)
            {
            if (strncmp (xtag, cp, strlen (xtag)) == 0)
                break;
            p = findnode (rcs->versions, cp);
            if (p == NULL)
                {
                free (xtag);
                return (RCS_NULL);
            }
            break;
            }
        vn = (RCSVers *) p->data;
        cp = vn->next;
        }
    free (xtag);
}

3040 if (force_tag_match)
    return (NULL);
else
    return (RCS_head (rcs));

3050 {
    if (force_tag_match)
        return (NULL);
    else
        return (RCS_head (rcs));
    }

return (xstrdup (cp));
```
/* if it had a ' ', terminate the string so we have the base revision */
  *sp = '\0';

  /* look up the revision this branch is based on */
  p = findnode (rcs->versions, tag);

  /* put the . back so we have the branch again */
  *sp = '.';

  if (p == NULL)
    {
      /* if the base revision didn't exist, return head or NULL */
      if (forewtagmatch)
        return (NULL);
      else
        return (RCShead (rcs));
    }

  /* find the first element of the branch we are looking for */
  vn = (RCSVers *) p->data;
  if (vn->branches == NULL)
    return (NULL);

  xtag = xmalloc (strlen (tag) + 1 + 1); /* 1 for the extra '.' */
  (void) strncpy (xtag, tag);
  (void) strcat (xtag, ".");
  head = vn->branches->list;
  for (p = head->next; p != head; p = p->next)
    if (strcmp (p->key, xtag, strlen (xtag)) == 0)
      break;
  free (xtag);

  if (p == head)
    {
      /* we didn't find a match so return head or NULL */
      if (forewtagmatch)
        return (NULL);
      else
        return (RCShead (rcs));
    }

  /* now walk the next pointers of the branch */
  nextvers = p->key;
  do
    {
      p = findnode (rcs->versions, nextvers);
      if (p == NULL)
        {
          /* a link in the chain is missing - return head or NULL */
          if (forewtagmatch)
            return (NULL);
          else
            return (RCShead (rcs));
        }
      vn = (RCSVers *) p->data;
      nextvers = vn->next;
    } while (nextvers != NULL);

  /* we have the version in our hand, so go for it */
  return (xstrdup (vn->version));

  /* Returns the head of the branch which REV is on. REV can be a branch tag
   or non-branch tag; symbolic or numeric.

   Returns a newly malloc'd string. Returns NULL if a symbolic name
   isn't found. */

  char *
  RCSbranchhead (rcs, rev)
  RCSNode *rcs;
  char *rev;

  { char *num;
    char *br;
    char *retval;
    assert (rcs != NULL);

    if (RCSnodeisbranch (rcs, rev))
      return RCSgetbranch (rcs, rev, 1);

    if (isdigit (*rev))
      num = xstrdup (rev);
    else
      {
        num = translate (rcs, rev);
        if (num == NULL)
          return NULL;
      }
    br = truncate (num);
```c
3150 retval = RCS_getbranch (rcs, br, 1);
    free (br);
    free (num);
    return retval;
}

3160 /* Get the branch point for a particular branch, that is the first
   revision on that branch.  For example, RCS_getbranchpoint (rcs,
   "1.3.2") will normally return "1.3.2.1".  TARGET may be either a
   branch number or a revision number; if a revnum, find the
   branchpoint of the branch to which TARGET belongs.

   Return RCS_head if TARGET is on the trunk or if the root node could
   not be found (this is sort of backwards from our behavior on a branch;
   the rationale is that the return value is a revision from which you
   can start walking the next fields and end up at TARGET).
   Return NULL on error. */

3170 static char * RCS_getbranchpoint (rcs, target)
    RCSNode *rcs;
    char *target;
    {
        char *branch, *bp;
        Node *vp;
        RCSVers *rev;
        int dots, isrevnum, brlen;

        dots = numdots (target);
        isrevnum = dots & 1;

        3180 if (dots == 1) /* TARGET is a trunk revision; return rcs->head. */
            return (RCS_head (rcs));

        /* Get the revision number of the node at which TARGET's branch is
         rooted.  If TARGET is a branch number, lop off the last field;
         if it's a revision number, lop off the last two fields. */
        branch = xstrdup (target);
        bp = strrchr (branch, '.');
        if (bp == NULL)
            error (1, 0, "%s: confused revision number %s",
                   rcs->path, target);
        if (isrevnum)
            while (*--bp != '.');
        *bp = '\0';
        vp = findnode (rcs->versions, branch);
        if (vp == NULL)
            {
                error (0, 0, "%s: can't find branch point %s",
                       rcs->path, target);
                return NULL;
            }
        rev = (RCSVers *) vp->data;

        3190 *bp++ = '1';
        while (*bp & & *bp != '.')
            ++bp;
        brlen = bp - branch;

        3200 vp = rev->branches->list->next;
        while (vp != rev->branches->list)
            {
                /* BRANCH may be a genuine branch number, e.g. "1.1.3", or
                 maybe a full revision number, e.g. "1.1.3.6".  We have
                 found our branch point if the first BRANCHLEN characters
                 of the revision number match, *and* if the following
                 character is a dot. */
                if (strncmp (vp->key, branch, brlen) == 0 & & vp->key[brlen] == '.')
                    break;

            }
        vp = vp->next;
    }

    free (branch);
    if (vp == rev->branches->list)
        {
            error (0, 0, "%s: can't find branch point %s",
                   rcs->path, target);
            return NULL;
        }
    else
        return (xstrdup (vp->key));
}

3220 /* Get the head of the RCS file.  If branch is set, this is the head of the
   branch, otherwise the real head.
   * Returns NULL or a newly malloc'd string. */
    char *
```
RCS\_head (rcs)

RCSNode *rcs;
{
    /* make sure we have something to look at... */
    assert (rcs != NULL);
    /*
     * NOTE: we call getbranch with force\_tag\_match set to avoid any
     * possibility of recursion
     */
    if (rcs->branch)
        return (RCS\_getbranch (rcs, rcs->branch, 1));
    else
        return (xstrdup (rcs->head));
}

/*
 * Get the most recent revision, based on the supplied date, but use some
 * funky stuff and follow the vendor branch maybe
 */

char *
RCS\_getdate (rcs, date, force\_tag\_match)
RCSNode *rcs;
char date;
int force\_tag\_match;
{
    char *cur\_rev = NULL;
    char *retval = NULL;
    Node *p;
    RCSVers *vers = NULL;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS\_parserscfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
    /* if the head is on a branch, try the branch first */
    if (rcs->branch != NULL)
        retval = RCS\_getdatebranch (rcs, date, rcs->branch);
    /* if we found a match, we are done */
    if (retval != NULL)
        return (retval);
    /* otherwise if we have a trunk, try it */
    if (rcs->head)
    {
        p = findnode (rcs->versions, rcs->head);
        while (p != NULL)
        {
            /* if the date of this one is before date, take it */
            vers = (RCSVers *) p->data;
            if (RCS\_datecmp (vers->date, date) <= 0)
            {
                cur\_rev = vers->version;
                break;
            }
            /* if there is a next version, find the node */
            if (vers->next != NULL)
            {
                p = findnode (rcs->versions, vers->next);
            }
            else
                p = (Node *) NULL;
        }
    }
    /* at this point, either we have the revision we want, or we have the
     * first revision on the trunk (1.1?) in our hands
     */

    /* if we found what we're looking for, and it's not 1.1 return it */
    if (cur\_rev != NULL && ! STREQ (cur\_rev, "1.1")
        return (xstrdup (cur\_rev));
    /* look on the vendor branch */
    retval = RCS\_getdatebranch (rcs, date, CVSBRANCH);
    /* if we found a match, return it; otherwise, we return the first
     * revision on the trunk or NULL depending on force\_tag\_match and the
     * date of the first rev */
    if (retval != NULL)
        return (retval);
    if (force\_tag\_match || RCS\_datecmp (vers->date, date) <= 0)
        return (xstrdup (vers->version));
    else
        return (NULL);
}
```c
return (NULL);
}

/*@ Look up the last element on a branch that was put in before the specified
* date (return the rev or NULL)
*/
static char *
RCSgetdatebranch (rcs, date, branch)
  RCSNode *rcs;
  char *date;
  char *branch;
{
  char *currev = NULL;
  char *cp;
  char xbranch, *xrev;
  Node *p;
  RCSVers *vers;
  /* look up the first revision on the branch */
  xrev = xstrdup (branch);
  if (cp == NULL)
  {
    free (xrev);
    return (NULL);
  }
  *cp = '\0';                         /* turn it into a revision */
  assert (rcs != NULL);
  if (rcs->flags & PARTIAL)
    RCSreparse_file (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
  p = findnode (rcs->versions, xrev);
  free (xrev);
  if (p == NULL)
    return (NULL);
  vers = (RCSVers *) p->data;
  /* Tentatively use this revision, if it is early enough. */
  if (RCSdatecmp (vers->date, date) <= 0)
    currev = vers->version;
  /* If no branches list, return now.  This is what happens if the branch
   * is a (magic) branch with no revisions yet.  */
  if (vers->branches == NULL)
    return xstrdup (currev);
  /* walk the branches list looking for the branch number */
  xbranch = xmalloc (strlen (branch) + 1 + 1); /* +1 for the extra dot */
  (void) strcpy (xbranch, branch);
  (void) strcat (xbranch, ".");
  for (p = vers->branches->list->next; p != vers->branches->list; p = p->next)
    if (strncmp (p->key, xbranch, strlen (xbranch)) == 0)
      break;
  free (xbranch);
  if (p == vers->branches->list)
  {
    /* This is what happens if the branch is a (magic) branch with
     * no revisions yet.  Similar to the case where vers->branches ==
     * NULL, except here there was a another branch off the same
     * branchpoint.  */
    return xstrdup (currev);
  }
  p = findnode (rcs->versions, p->key);
  /* walk the next pointers until you find the end, or the date is too late */
  while (p != NULL)
  {
    vers = (RCSVers *) p->data;
    if (RCSdatecmp (vers->date, date) <= 0)
      currev = vers->version;
    else
      break;
  /* if there is a next version, find the node */
  if (vers->next != NULL)
    p = findnode (rcs->versions, vers->next);
  else
    p = (Node *) NULL;
}
/* Return whatever we found, which may be NULL. */
return xstrdup (currev);
}

/*@ Compare two dates in RCS format.  Beware the change in format on January 1,
* ...
*/
```
/* 2000, when years go from 2-digit to full format.  */
int
RCS_datecmp (date1, date2)
char *date1, *date2;
{
    int length_diff = strlen (date1) - strlen (date2);
    return (length_diff ? length_diff : strcmp (date1, date2));
}

/* Look up revision REV in RCS and return the date specified for the
revision minus FUDGE seconds (FUDGE will generally be one, so that the
logically previous revision will be found later, or zero, if we want
the exact date).  The return value is the date being returned as a time
on error (previously was documented as zero on error; I haven't checked
then it must point to MAXDATELEN characters, and we store the same
return value there in DATEFORM format.  */

int
RCS_gettime (rcs, rev, date, fudge)
RCSNode *rcs;
char *rev;
char *date;
int fudge;
{
    char tdate[MAXDATELEN];
    struct tm *xtm = NULL;
    time_t revdate = 0;
    Node *p;
    RCSVers *vers;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if ((rcs->flags & PARTIAL) & (rcs->file) NULL, (struct rcsbuffer *) NULL);

    /* look up the revision */
    p = findnode (rcs->versions, rev);
    if (p == NULL)
        return (-1);
    vers = (RCSVers *) p->data;

    /* split up the date */
    ftm = &tm;
    (void) sscanf (vers->date, SDATEFORM, &ftm->tm_year, &ftm->tm_mon,
         &ftm->tm_mday, &ftm->tm_hour, &ftm->tm_min,
        &ftm->tm_sec);

    /* If the year is from 1900 to 1999, RCS files contain only two
digits, and sscanf gives us a year from 0-99. If the year is
2000+, RCS files contain all four digits and we subtract 1900,
because the tm_year field should contain years since 1900.  */
    if (ftm->tm_year > 1900)
        ftm->tm_year -- 1900;

    /* put the date in a form getdate can grok */
    (void) sprintf (tdate, "%d/%d/%d 00:00:00", ftm->tm_mon,
          ftm->tm_mday, ftm->tm_year + 1900, ftm->tm_hour,
          ftm->tm_min, ftm->tm_sec);

    /* turn it into seconds since the epoch */
    if (revdate != (time_t) -1)
    {
        revdate -= fudge; /* remove "fudge" seconds */
    }
    if (date)
    {
        /* put an appropriate string into "date" if we were given one */
        ftm = gmtime (&revdate);
        (void) sprintf (date, DATEFORM,
           ftm->tm_year + (ftm->tm_year < 1900 ? 0 : 1900),
           ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
           ftm->tm_min, ftm->tm_sec);
    }

    return (revdate);
}

List *RCS_getblocks (rcs)
RCSNode *rcs;
{
    assert(rcs != NULL);
if (rcs->flags & PARTIAL)
    RCSparseonsfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
if (rcs->lockspdata) {
    rcs->locks = getlist ();
    do_locks (rcs->locks, rcs->lockspdata);
    free(rcs->lockspdata);
    rcs->lockspdata = NULL;
}
return rcs->locks;
}

List = RCSsymbols(rcs)
RCSNode rcs;
{
    assert(rcs != NULL);
if (rcs->flags & PARTIAL)
    RCSparseonsfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
if (rcs->symbols_pdata) {
    rcs->symbols = getlist ();
    do_symbols (rcs->symbols, rcs->symbols_pdata);
    free(rcs->symbols_pdata);
    rcs->symbols_pdata = NULL;
}
return rcs->symbols;
}

/*
 * Returns the version associated with a particular symbolic tag.
 * Returns NULL or a newly malloc'd string.
 */
static char *
translate_rsymtag (rcs, tag)
RCSNode rcs;
const char *tag;
{
    if (rcs->flags & PARTIAL)
        RCSparseonsfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
    if (rcs->symbols != NULL)
        {
            Node *p;
            /* The symbols have already been converted into a list. */
            p = findnode (rcs->symbols, tag);
            if (p == NULL)
                return NULL;
            return xstrdup (p->pdata);
        }
    if (rcs->symbols_sdata != NULL)
        {
            size_t len;
            char *cp;
            /* Look through the RCS symbols information. This is like
do_symbols, but we don’t add the information to a list. In
most cases, we will only be called once for this file, so
generating the list is unnecessary overhead. */
            len = strlen (tag);
            cp = rcs->symbols_sdata;
            while ((cp = strchr (cp, tag[0])) != NULL)
            {
                if ((cp == rcs->symbols_sdata || whitespace (cp[-1]))
                    && strncmp (cp, tag, len) == 0
                    && cp[len] == ‘:’)
                    {
                        char *v, *r;
                        /* We found the tag. Return the version number. */
                        cp += len + 1;
                        *v = cp;
                        while (! whitespace (*cp) && *cp != ‘\0’)
                            cp++; *
                        t = xmalloc (cp – v + 1);
                        strncpy (t, v, cp – v);
                        if (cp[cp] != ‘\0’)
                            return t;
                        }
            while (! whitespace (*cp) && *cp != ‘\0’)
                cp++; *
*/
* The argument ARG is the getopt remainder of the *k* option specified on the
* command line. This function returns malloc-ed space that can be used
directly in calls to RCS V5, with the *k* flag munged correctly.
*/

3610 char *
RCS_check_kflag (arg)
const char *arg;
{
static const char *const keyword_usage[] =
{ /*%ls: invalid RCS keyword expansion mode %n*/
  "%s %s: invalid RCS keyword expansion mode\n",
  "Valid expansion modes include:\n",
  " -kkv\tGenerate keywords using the default form.\n",
  " -kkvl\tLike -kkv, except locker's name inserted.\n",
  " -k\tGenerate only keyword names in keyword strings.\n",
  " -k\tGenerate only keyword values in keyword strings\n",
  " -ko\tGenerate the old keyword string (no changes from checked in file)\n",
  " -kb\tGenerate binary file unmodified (merges not allowed) (RCS 5.7)\n",
  "(Specify the --help global option for a list of other help options)\n",
  NULL,
}; /* Big enough to hold any of the strings from kflags. */
char karg[10];
char const *const *cpp = NULL;

if (arg)
{
  for (cpp = kflags; *cpp != NULL; cpp++)
  {
    if (STREQ (arg, *cpp))
      break;
  }
}

if (arg == NULL || *cpp == NULL)
{
  usage (keyword_usage);
  return (malloc (xstrdup (karg)));
}

/* Do some consistency checks on the symbolic tag... These should equate
 pretty close to what RCS checks, though I don't know for certain.
*/
void
RCS_check_tag (tag)
const char *tag;
{
  char *invalid = "$,:;@"; /* invalid RCS tag characters */
  const char *cp = NULL;

  /* Check for remote tags first */
  if (tag[0] == ':')
    {
      const char * realtag = tag;
      realtag = strchr (realtag + 1, ':');
      if (realtag != NULL) {
        realtag = strchr (realtag + 1, ':');
        if (realtag != NULL) {
          return RCS_check_tag (realtag + 1);
        }
      }
    }

  /* The first character must be an alphabetic letter. The remaining
   * characters cannot be non-visible graphic characters, and must not be
   * in the set of "invalid" RCS identifier characters.
   */
  if (isalpha (*tag))
    {
      for (cp = tag; *cp; cp++)
        {
          if (isgraph (*cp))
            error (1, 0, "tag \"%s\" has non-visible graphic characters",
                   tag);
          if (strchr (invalid, *cp))
            error (1, 0, "tag \"%s\" must not contain the characters \"%s\",
                   tag, invalid);
        }
    }
}
else  
    error (1, 0, "tag 'Zs' must start with a letter", tag);
}
/
* TRUE if argument has valid syntax for an RCS revision or branch number. All characters
* must be digits or dots, first and last characters must be digits, and no two consecutive
* characters may be dots.
* Intended for classifying things, so this function doesn't
* call error.
*/
int RCS_validRev (rev)
char *rev;
{
    char last, c;
    last = *rev++;
    if (!isdigit (last))
        return 0;
    while (((c = *rev++)) /* Extra parens placate -Wall gcc option */
        {
            if (c == '.')
                {
                    if (last == '.')
                        return 0;
                    continue;
                }
            last = c;
            if (!isdigit (c))
                return 0;
        }
    return 1;
}
/
* Return true if RCS revision with TAG is a dead revision.
*/
int RCS_isdead (rcs, tag)
RCSNode *rcs;
const char *tag;
{
    Node *p;
    RCSVers *version;
    if (rcs->flags & PARTIAL)
        RCS_reparse RCSfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
    p = findnode (rcs->versions, tag);
    if (p == NULL)
        return (0);
    version = (RCSVers *) p->data;
    return (version->dead);
}
/
* Return the RCS keyword expansion mode. For example "b" for binary. 
* Returns a pointer into storage which is allocated and freed along with 
* the rest of the RCS information; the caller should not modify this 
* storage. Returns NULL if the RCS file does not specify a keyword 
* expansion mode; for all other errors, die with a fatal error. */
char *
RCS_getexpand (rcs)
RCSNode *rcs;
{
    assert (rcs != NULL);
    return rcs->expand;
}
/
* RCS keywords, and a matching enum. */
struct RCS_keyword
{
    const char *string;
    size_t len;
};
#define KEYWORD_INIT(s)  (s), sizeof (s) - 1
static const struct RCS_keyword keywords[] =
{
    { KEYWORD_INIT("Author") },
    { KEYWORD_INIT("Date") },
    { KEYWORD_INIT("Header") },
    { KEYWORD_INIT("Id") },
    { KEYWORD_INIT("Locker") },
    { KEYWORD_INIT("Log") },
    { KEYWORD_INIT("Name") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },
    { KEYWORD_INIT("RCSfile") },

static char * 
printable_date (char *rdate) 
{ 
    char *rdate; 
    int year, mon, mday, hour, min, sec; 
    char buf[105];

    (void) sscanf (rdate, SDATEFORM, &year, &month, &mday, &hour, &min, &sec); 

    if (year < 1900) 
        year += 1900; 
    sprintf (buf, "%04d/%02d/%02d %02d:%02d:%02d", year, month, mday, hour, min, sec); 
    return xstrdup (buf); 
}

/* Convert an RCS date string into a readable string. This is like 
the RCS date2str function. */

static char * 
escape_keyword (char *value, int free_value) 
{ 
    const char *value; 
    int *free_value; 

    char *ret, *t; 
    const char *s; 

    for (s = value; *s != '\0'; s++) 

    { 
        char c; 

        c = *s; 
        if (c == 't') 
            | c == '\n' 
            | c == '\' 
            | c == ' ' 
            | c == '$') 
            
            { 
                break; 
            } 

        if (*s == '\0') 
            *free_value = 0; 
        return (char *) value; 
    } 

    ret = xmalloc (strlen (value) * 4 + 1); 
    *free_value = 1; 

    for (s = value, t = ret; *s != '\0'; s++, t++) 

    { 
        switch (*s) 
        { 
            default: 
                *t = *s; 
            break; 

            case 't': 
                *t++ = '\n'; 
                *t = 't'; 
            break; 

            case 'b': 
                *t++ = '\\'; 
                *t = 'n'; 
            break; 

            case 'v': 
                *t++ = '\\'; 
                *t = 'v'; 
            break; 

            case 'n': 
                *t++ = '\\'; 
                *t = 'n'; 
            break; 

            case 's': 
                *t++ = '\\'; 
                *t = 's'; 
            break; 

            case 'd': 
                *t++ = '\\'; 
                *t = 'd'; 
            break; 

            case 'w': 
                *t++ = '\\'; 
                *t = 'w'; 
            break; 

            case 'l': 
                *t++ = '\\'; 
                *t = 'l'; 
            break; 

            case 'r': 
                *t++ = '\\'; 
                *t = 'r'; 
            break; 

            case 'x': 
                *t++ = '\\'; 
                *t = 'x'; 
            break; 

            case '\': 
                *t++ = '\\'; 
                *t = '\'; 
            break; 

            case '\": 
                *t++ = '\\'; 
                *t = '"'; 
            break; 

            case '0': 
                *t++ = '\\'; 
                *t = '0'; 
            break; 

            case '1': 
                *t++ = '\\'; 
                *t = '1'; 
            break; 

            case '2': 
                *t++ = '\\'; 
                *t = '2'; 
            break; 

            case '3': 
                *t++ = '\\'; 
                *t = '3'; 
            break; 

            case '4': 
                *t++ = '\\'; 
                *t = '4'; 
            break; 

            case '5': 
                *t++ = '\\'; 
                *t = '5'; 
            break; 

            case '6': 
                *t++ = '\\'; 
                *t = '6'; 
            break; 

            case '7': 
                *t++ = '\\'; 
                *t = '7'; 
            break; 

            case '8': 
                *t++ = '\\'; 
                *t = '8'; 
            break; 

            case '9': 
                *t++ = '\\'; 
                *t = '9'; 
            break; 

            case '\n': 
                *t++ = '\\'; 
                *t = '\n'; 
            break; 

            case '\t': 
                *t++ = '\\'; 
                *t = '\t'; 
            break; 

            case '\': 
                *t++ = '\\'; 
                *t = '\'; 
            break; 

            case '0': 
                *t++ = '\\'; 
                *t = '0'; 
            break; 

            case '1': 
                *t++ = '\\'; 
                *t = '1'; 
            break; 

            case '2': 
                *t++ = '\\'; 
                *t = '2'; 
            break; 

            case '3': 
                *t++ = '\\'; 
                *t = '3'; 
            break; 

            case '4': 
                *t++ = '\\'; 
                *t = '4'; 
            break; 

            case '5': 
                *t++ = '\\'; 
                *t = '5'; 
            break; 

            case '6': 
                *t++ = '\\'; 
                *t = '6'; 
            break; 

            case '7': 
                *t++ = '\\'; 
                *t = '7'; 
            break; 

            case '8': 
                *t++ = '\\'; 
                *t = '8'; 
            break; 

            case '9': 
                *t++ = '\\'; 
                *t = '9'; 
            break; 

            case '\": 
                *t++ = '\\'; 
                *t = '"'; 
            break; 

            case '0': 
                *t++ = '\\'; 
                *t = '0'; 
            break; 

            case '1': 
                *t++ = '\\'; 
                *t = '1'; 
            break; 

            case '2': 
                *t++ = '\\'; 
                *t = '2'; 
            break; 

            case '3': 
                *t++ = '\\'; 
                *t = '3'; 
            break; 

            case '4': 
                *t++ = '\\'; 
                *t = '4'; 
            break; 

            case '5': 
                *t++ = '\\'; 
                *t = '5'; 
            break; 

            case '6': 
                *t++ = '\\'; 
                *t = '6'; 
            break; 

            case '7': 
                *t++ = '\\'; 
                *t = '7'; 
            break; 

            case '8': 
                *t++ = '\\'; 
                *t = '8'; 
            break; 

            case '9': 
                *t++ = '\\'; 
                *t = '9'; 
            break; 

            case '\":
3870  
    
    case ‘\':
    *t++ = '\';
    *t = '\';
    break;
    
    case '0':
    *t++ = '0';
    *t = '0';
    break;
    
    case '4':
    *t++ = '4';
    *t = '4';
    break;

3880  
    case '$':
    *t++ = '$';
    *t = '$';
    break;
    
}  

    *t = '"0';

    return ret;

3890  

/\ Expand RCS keywords in the memory buffer BUF of length LEN. This
\ applies to file RCS and version VERS. If NAME is not NULL, and is
\ not a numeric revision, then it is the symbolic tag used for the
\ checkout. EXPAND indicates how to expand the keywords. This
\ function sets RETBUF and RETLEN to the new buffer and length.
\ This function may modify the buffer BUF. If BUF != RETBUF, then
\ RETBUF is a newly allocated buffer. */

3900  

static void
expand_keywords (rcs, ver, name, log, loglen, expand, buf, len, retbuf, retlen)
    RCSNode *rcs;
    RCSVer *ver;
    const char *name;
    const char *log;
    size_t loglen;
    enum kflag expand;
    char *buf;
    size_t len;

3910  
    char **retbuf;
    size_t *retlen;
    
    {  
        struct expand_buffer
        {  
            struct expand_buffer *next;
            char *data;
            size_t len;
            int free_data;
        } *ebufs = NULL;
        struct expand_buffer *ebuf_last = NULL;
        size_t ebuf_last_len = 0;
        char *locker;
        char *srch, *srch_next;
        size_t archlen;

3920  
        if (expand == KFLAG_K0 || expand == KFLAG_KB)
        {  
            retbuf = buf;
            *retlen = len;
            return;
        }

4030  
        /\ If we are using -kkvl, dig out the locker information if any. */
        locker = NULL;
        if (expand == KFLAG_KV)
        {  
            Node *lock;
            lock = findnode (RCS_getlocks(rcs), ver->version);
            if (lock != NULL)

3940  
            locker = xstrdup (lock->data);
        }

4050  
        /\ RCS keywords look like $STRING or STRING: VALUES. */
        arch = buf;
        arch_next = len;
        while ((arch_next = memchr (arch, ' ', arch_next)) != NULL)
        {  
            char *s, *send;
            size_t slen;

3950  
            const struct rcs_keyword *keyword;
            enum keyword kw;
            char *value;
            int free_value;
            char *sub;
            size_t sublen;

            arch_next = (arch_next + 1) - arch;
            arch = arch_next + 1;
/* Look for the first non alphabetic character after the '$'. */
send = src + src_len;
for (s = src; s < send; s++)
  if (!isalpha(*s))
    break;
/* If the first non alphabetic character is not '$' or ':',
  then this is not an RCS keyword. */
if (s == send || (*s == '$' && *s != ':'))
  continue;
/* See if this is one of the keywords. */
slen = s - src;
for (keyword = keywords; keyword->string != NULL; keyword++)
  if (keyword->len == slen
      && strncmp(keyword->string, src, slen) == 0)
    break;
if (keyword->string == NULL)
  continue;
kw = (enum keyword) (keyword - keywords);
/* If the keyword ends with a ':', then the old value consists
  of the characters up to the next '$'. If there is no '$'
  before the end of the line, though, then this wasn't an RCS
  keyword after all. */
if (*s == ':')
  for (; s < send; s++)
    if (*s == '$' || *s == '\n')
      break;
if (s == send || *s != '$')
  continue;
/* At this point we must replace the string from SRCH to S
  with the expansion of the keyword KW. */
/* Get the value to use. */
free_value = 0;
if (expand == KFLAG_K)
  value = NULL;
else
  switch (kw)
  {
  default:
    abort ();
    case KEYWORD_AUTHOR:
      value = ver->author;
      break;
    case KEYWORD_DATE:
      value = printable_date (ver->date);
      break;
    case KEYWORD_HEADER:
    case KEYWORD_ID:
      { char *path;
        int free_path;
        char *date;

        if (kw == KEYWORD_HEADER)
          path = rcs->path;
        else
          path = last_component (rcs->path);
        path = escape_keyword_value (path, &free_path);
        date = printable_date (ver->date);
        value = xmalloc (strlen (path)
                        + strlen (ver->version)
                        + strlen (date)
                        + strlen (ver->author)
                        + strlen (ver->state)
                        + strlen (locker)
                        + 20);
        sprintf (value, "%s %s %s %s %s%08x",
                  path, ver->version, date, ver->author,
                  ver->state, locker != NULL ? locker : "")
        if (free_path)
free (path);
    free (date);
    free_value = 1;
}
break;

case KEYWORD_LOCKER:
    value = locker;
    break;
case KEYWORD_LOG:
    case KEYWORD_RCSFILE:
        value = escape_keyword_value (last_component (rcs->path), &free_value);
        break;
case KEYWORD_NAME:
        if (name != NULL && !isdigit (*name))
            value = (char *) name;
        else
            value = NULL;
        break;
    case KEYWORD_REVISION:
        value = ver->version;
        break;
    case KEYWORD_SOURCE:
        value = escape_keyword_value (rcs->path, &free_value);
        break;
case KEYWORD_STATE:
    value = ver->state;
    break;
}

sub = xmalloc (keyword->len
    + (value == NULL ? 0 : strlen (value))
    + 10);
if (expand == KFLAG_V)
    {
        /* Decrement SRCH and increment S to remove the $ characters. */
        --srch;
        ++arch_len;
        s = new;
        sublen = 0;
    }
else
    {
        strcpy (sub, keyword->string);
        sublen = strlen (keyword->string);
        if (expand != KFLAG_K)
            {
                sub[sublen] = ':';
                sub[sublen + 1] = ' '; 
                sublen += 2;
            }
    }
if (value != NULL)
    {
        strcpy (sub + sublen, value);
        sublen += strlen (value);
    }
if (expand != KFLAG_V && expand != KFLAG_K)
    {
        sub[sublen] = ' '; 
        ++sublen;
        sub[sublen] = '\0';
    }
if (free_value)
    free (value);

/* The Log keyword requires special handling. This behaviour is taken from RCS 5.7. The special log message is what RCS uses for ci -k. */
if (kw == KEYWORD_LOG 
    && sizeof "checked in with -k by" <= loglen
    && strncmp (log, "checked in with -k by ",
                sizeof "checked in with -k by " - 1) != 0))
    {
char *start;
char *leader;
size_t leader_len, leader_p_len;
const char *logend;
const char *snl;
int cnl;
char *date;

...
const char *sl;

/* We are going to insert the trailing $ ourselves, before
the log message, so we must remove it from S, if we
haven't done so already. */
if (expand != KFLAG_Y)
++s;

/* Find the start of the line. */
start = srch;
while (*start > buf && start[-1] != 'n')
  --start;

/* Copy the start of the line to use as a comment leader. */
leader_len = start - srch;
if (expand != KFLAG_Y)
  leader_len = xmalloc(leader_len);
memcpy (leader, start, leader_len);
leader_p_len = leader_len;
while ((leader_p_len > 0 && leader[leader_p_len - 1] == ' ')
  --leader_p_len;

/* RCS does some checking for an old style of Log here,
but we don't bother. RCS issues a warning if it
changes anything. */

/* Count the number of newlines in the log message so that
we know how many copies of the leader we will need. */
cnl = 0;
logend = log + loglen;
for (sl = log; sl < logend; sl++)
  if (*sl == '
')
    ++cnl;

date = printable(ver->date);
sub = xmalloc(sublen
  + snprintf "Revision"
  + strlen(ver->version)
  + strlen(date)
  + strlen(ver->author)
  + loglen
  + (cnl + 2) + leader_len
  + 20);
if (expand != KFLAG_Y)
{
  sub[sublen] = '$';
  ++sublen;
}
sub[sublen] = '\n';
++sublen;
memcpy (sub + sublen, leader, leader_len);
sublen += leader_len;
sprintf (sub + sublen, "Revision %s %s %s
", ver->version, date, ver->author);
sublen += strlen (sub + sublen);
free (date);

sl = log;
while (sl < logend)
{
  if (*sl == '\n')
  {
    memcpy (sub + sublen, leader, leader_len);
    sublen += leader_p_len;
    sub[sublen] = '\n';
    ++sublen;
  } else
{
    const char *sln;
    memcpy (sub + sublen, leader, leader_len);
    for (sln = sl; sln < logend && *sln != 'n'; ++sln)
    {
      if (*sln == $)
        ++sln;
    }
    memcpy (sub + sublen, sl, sln - sl);
    sublen += sln - sl;
    sl = sln;
  }
}
memcpy (sub + sublen, leader, leader_p_len);
sublen += leader_p_len;
free (leader);
if (srch + sublen == s) {
    memcpy (arch, sub, sublen);
    free (sub);
} else {
    struct expand_buffer *ebuf;
    /* We need to change the size of the buffer. We build a list of expand_buffer structures. Each expand_buffer structure represents a portion of the final output. We concatenate them back into a single buffer when we are done. This minimizes the number of potentially large buffer copies we must do. */
    if (ebufs == NULL) {
        ebuf = (struct expand_buffer *) xmalloc (sizeof *ebuf);
        ebuf->next = NULL;
        ebuf->data = buf;
        ebuf->free_data = 0;
        ebuf->len = arch->buf;
        ebuf->last = ebuf;
    } else {
        assert (srch >= ebuf->last->data);
        assert (arch <= ebuf->last->data + ebuf->last->len);
        ebuf->len = ebuf->last->len - (arch - ebuf->last->data);
        ebuf->last->len = arch->buf - ebuf->last->data;
    }
    ebuf = (struct expand_buffer *) xmalloc (sizeof *ebuf);
    ebuf->data = arch;
    ebuf->len = sublen;
    ebuf->free_data = 1;
    ebuf->next = NULL;
    if (ebufs != NULL) ebuf->last = ebuf;
    ebuf->last = ebuf;
    ebuf->len += sublen;
    ebuf = (struct expand_buffer *) xmalloc (sizeof *ebuf);
    ebuf->len = arch->buf - (s - arch);
    ebuf->free_data = 0;
    ebuf->next = NULL;
    ebuf->last = ebuf;
    ebuf->last->len = arch->buf - (s - arch);
    arch->len -= (s - arch);
    arch = s;
}

if (locker != NULL) free (locker);
if (ebufs == NULL) {
    *retbuf = buf;
    *retlen = len;
} else {
    char *ret;
    ret = xmalloc (ebuf->len);
    *retbuf = ret;
    *retlen = ebuf->len;
    while (ebufs != NULL) {
        struct expand_buffer *next;
    }
    memcpy (ret, ebuf->data, ebuf->len);
    ret += ebuf->len;
    if (ebufs->free_data)
        free (ebufs->data);
    next = ebuf->next;
    free (ebufs);
    ebuf = next;
}
/* Check out a revision from an RCS file. 
If PFN is not NULL, then ignore WORKFILE and SOUT. Call PFN zero 
or more times with the contents of the file. CALLERDAT is passed, 
uninterpreted, to PFN. (The current code will always call PFN 
exactly once for a non empty file; however, the current code 
assumes that it can hold the entire file contents in memory, which 
isa good assumption, and might change in the future). 

Otherwise, if WORKFILE is not NULL, check out the revision to 
WORKFILE. However, if WORKFILE is not NULL, and noexec is set, 
then don’t do anything. 
Otherwise, if WORKFILE is NULL, check out the revision to SOUT. If 
SOUT is RUN_TTY, then write the contents of the revision to 
standard output. When using SOUT, the output is generally a 
temporary file; don’t bother to get the file modes correct. 
REV is the numeric revision to check out. It may be NULL, which 
means to check out the head of the default branch. 
If NAMETAG is not NULL, and is not a numeric revision, then it is 
the tag that should be used when expanding the RCS Name keyword. 
OPTIONS is a string such as “-kb” or “-kv” for keyword expansion 
options. It may be NULL to use the default expansion mode of the 
file, typically “-kkv”. 
On an error which prevented checking out the file, either print a 
nonfatal error and return 1, or give a fatal error. On success, 
return 0. */ 

/* This function mimics the behavior of ‘rcs co’ almost exactly. The 
chief difference is in its support for preserving file ownership, 
permissions, and special files across checkin and checkout — see 
comments in RCS_checkin for some issues about this. -twp */ 

int RCS_checkout (rcs, workfile, rev, nametag, options, sout, pfn, callerdat) 
int free_rev = 0; 
enum kflag expand; 
FILE *fp, *ofp; 
struct stat sb; 
struct rcbufferrcsbuff; 
char *key; 
char *value; 
size_t len; 
int free_value = 0; 
char *log = NULL; 
size_t loglen; 
Node *vp = NULL; 
#define PRESERVE_PERMISSIONS_SUPPORT 
uid_t rcs_owner = (uid_t) -1; 
gid_t rcs_group = (gid_t) -1; 
mode_t rcs_mode; 
int change_rcsowner = 0; 
int change_rcsgroup = 0; 
int special_file = 0; 
unsigned long devnum_long; 
dev_t devnum = 0; 
#endif 
if (trace) 
{ 
(void) fprintf (stderr, "#s> checkout (%s, %s, %s, %s)\n", 
#define SERVER_SUPPORT 
srvactive ? "S" : " ", 
#else 
" ", 
#endif 
rcs->path, 
rev != NULL ? rev : "", 
options != NULL ? options : "", 
(pfn != NULL ? "(function)" : "") 
? (workfile != NULL 
? workfile : "") 
? (sout != RUN_TTY ? sout : "(stderr)"))); 
}
assert (rev == NULL || isdigit (*rev));

if (noexec & & workfile != NULL)
    return 0;

assert (sout == RUNTTY || workfile == NULL);
assert (pfn == NULL || (sout == RUNTTY & & workfile == NULL));

/* Some callers, such as Checkin or removefile, will pass us a branch. */
if (rev != NULL & & (numdots (rev) & 1) == 0)
{
    rev = RCSgetbranch (rcs, rev, 1);
    if (rev == NULL)
        error (1, 0, "internal error: bad tag in checkout");
    free_rev = 1;
}

if (rev == NULL || STREQ (rev, rcs->head))
{
    int gothead;

    /* We want the head revision. Try to read it directly. */
    if (rcs->flags & PARTIAL)
        RCS_reparsersfile (rcs, &fp, &rcsbuf);
    else
        rcsbuf_valcopy (rcsbuf, &rcsbuf, &rcsbuf);

    gothead = 0;
    if (! rcsbuf_getrevnum (&rcsbuf, &key))
        error (1, 0, "unexpected EOF reading ")*, rcs->path");
    while (rcsbuf_getkey (&rcsbuf, &key, &value))
    {
        if (STREQ (key, "text"))
            log = rcsbuf_valcopy (&rcsbuf, value, 0, &loglen);
        else if (STREQ (key, "log"))
        {
            gothead = 1;
            break;
        }
    }

    if (! gothead)
    {
        error (0, 0, "internal error: cannot find head text");
        if (free_rev)
            free (rev);
        return 1;
    }

    rcsbuf_valpolish (&rcsbuf, value, 0, &len);
    if (fstat (fileno (fp), &sb) < 0)
        error (1, errno, "cannot stat ")*, rcs->path");

    rcsbuf_cache (rcs, &rcsbuf);
    else
    {
        struct rcsbuf *rcsbufp;

        /* It isn't the head revision of the trunk. We'll need to walk through the deltas. */
        fp = NULL;
        if (rcs->flags & PARTIAL)
            RCS_reparsersfile (rcs, &fp, &rcsbuf);

        if (fp == NULL)
        {
            /* If RCS_deltas didn't close the file, we could use fstat here too. Probably should change it thusly... */
            if (fstat (rcs->path, &sb) < 0)
                error (1, errno, "cannot stat ")*, rcs->path");
            rcsbufp = NULL;
        }
        else
        {
            if (fstat (fileno (fp), &sb) < 0)
                error (1, errno, "cannot stat ")*, rcs->path");
            rcsbufp = &rcsbuf;
        }

        RCS_deltas (rcs, fp, rcsbufp, rev, RCS_PETCH, &value, &len, &log, &loglen);
        free value = 1;
    }

    /* If OPTIONS is NULL or the empty string, then the old code would... */
invoke the RCS co program with no -k option, which means that
co would use the string we have stored in rcs->expand. */
if (options != NULL || options[0] == '"' || rcs->expand == NULL)
    expand = KFLAG_KV;
else {
    const char *ouroptions;
    const char *const *cpp;
    if (options != NULL && options[0] != '"')
        { /* "xxx", "yy", "zz" */
            assert (options[0] == '"' && options[1] == 'x');
           ouroptions = options + 2;
        }
    else
       ouroptions = rcs->expand;
    for (cpp = kflags; *cpp != NULL; cpp++)
        if (!STREQ (*cpp,ouroptions))
            break;

    if (*cpp != NULL)
        expand = (enum kflag) (cpp - kflags);
    else {
        error (0,0,
            "internal error: unsupported substitution string -k%c",
           ouroptions);
        expand = KFLAG_KV;
    }
}

#define PRESERVE_PERMISSIONS_SUPPORT
/* Handle special files and permissions, if that is desired. */
if (preserve_perms) {
    RCSVers *vers;
    Node *info;
    vp = findnode (rcs->versions, rev == NULL ? rcs->head : rev);
    if (vp == NULL)
        error (1, 0, "internal error: no revision information for %s",
            rev == NULL ? rcs->head : rev);
    vers = (RCSVers *) vp->data;
    /* First we look for symlinks, which are simplest to handle. */
    info = findnode (vers->other_delta, "symlink");
    if (info != NULL) {
        char *dest;
        if (pfn != NULL || (workfile != NULL && sout == RUN_TTY))
            error (1, 0, "symbolic link %s to %s cannot be piped",
                rcs->path, vers->version);
        if (workfile == NULL)
            dest = sout;
        else
            dest = workfile;
        /* Remove 'dest': just in case. It's okay to get ENOENT here,
           since we just want the file not to be there. (TODO: decide
           whether it should be considered an error for 'dest' to exist
           at this point. If so, the unlink call should be removed and
           'symlink' should signal the error. -twp) */
        if (unlink (dest) < 0 && existance_error (errno))
            error (1, errno, "cannot remove %s", dest);
        if (symlink (info->data, dest) < 0)
            error (1, errno, "cannot create symbolic link from %s to %s",
                dest, info->data);
        if (free_value)
            free (value);
    }
    if (free_rev)
        free (rev);
    return 0;
}
/* Next, we look at this file's hardlinks field, and see whether
it is linked to any other file that has been checked out.
If so, we don't do anything else — just link it to that file.

If we are checking out a file to a pipe or temporary storage,
none of this should matter. Hence the 'workfile != NULL' wrapper
around the whole thing. -twp */
if (workfile != NULL) {
    List *links = vers->hardlinks;
    if (links != NULL) {
        Node *uptodate_link;
/* For each file in the hardlinks field, check to see 
   if it exists, and if so, if it has been checked out 
   this iteration. When walklist returns, updatelink 
   should point to a hardlist node representing a file 
   in 'links' which has recently been checked out, or
   NULL if no file in 'links' has yet been checked out. */

updatelink = NULL;
(void) walklist (links, findcheckedout, &updatelink);
dellist (&links);

/* If we've found a file that 'workfile' is supposed to be 
   linked to, and it has been checked out since CVS was 
   invoked, then simply link workfile to that file and return.

If one of these conditions is not met, then 
workfile is the first one in its hardlink group to 
be checked out, and we must continue with a full 
check out. */

if (updatelink != NULL) 
{ 
   struct hardlink_info *hlinfo = 
   (struct hardlink_info *) updatelink->data;

   if (link (updatelink->key, workfile) < 0) 
      error (1, errno, "cannot link %s to %s");

   hlinfo->checkedout = 1; /* probably unnecessary */

   if (free (value))
      free (value);
   if (free (rev))
      free (rev);
   return 0;
}

info = findnode (vers->other_delta, "owner");
if (info != NULL) 
{
   change->owner = (uid_t) strtoul (info->data, NULL, 10);
}

info = findnode (vers->other_delta, "group");
if (info != NULL) 
{
   change->group = (gid_t) strtoul (info->data, NULL, 10);
}

info = findnode (vers->other_delta, "permissions");
if (info != NULL) 
{
   change->mode = 4;
   revnum = (mode_t) strtoul (info->data, NULL, 8);
}

info = findnode (vers->other_delta, "special");
if (info != NULL) 
{
   /* If the size of 'devtype' changes, fix the scanf call also */
   char devtype[10];

   if (scanf (info->data, "%Hs %lu", 
      devtype, &devnum) < 2) 
      error (1, 0, "%s has bad 'special' newphrase 's'",
      workfile, vers->version, info->data);

   devnum = devnum; /* condif */
   if (strcmp (devtype, "character") == 0) 
      special_file = SI_FCHR;
   else if (strcmp (devtype, "block") == 0) 
      special_file = SI_FBLK;
   else 
      error (0, 0, "%s is a special file of unsupported type '%s'",
      workfile, info->data);
}

if (expand != KFLAG_O && expand != KFLAG_S) 
{
   char *newvalue;

   /* Don't fetch the delta node again if we already have it. */
   if (vp == NULL) 
   {
      vp = findnode (rcs->versions, rev == NULL ? rcs->head : rev);
   }
   else 
      error (1, 0, "internal error: no revision information for %s",
      rev == NULL ? rcs->head : rev);
}
expand_keywords (rcs, (RCSVers *) vp->data, nametag, log, loglen, 
expand, value, len, &newvalue, &len);
if (newvalue != value)
{
  if (free_value)
    free (value);
  value = newvalue;
  free_value = 1;
}
}
if (free_rev)
  free (rev);
if (log != NULL)
{
  free (log);
  log = NULL;
}
if (pn != NULL)
{
  ifdef PRESERVE_PERMISSIONS_SUPPORT
    if (special_file)
      error (1, 0, "special file $a cannot be piped to anything", 
rcs->path);
  endif
  /* The PPN interface is very simple to implement right now, as 
   * we always have the entire file in memory. */
  if (len != 0)
    pfn (callerdat, value, len);
  ifdef PRESERVE_PERMISSIONS_SUPPORT
  endif
  else if (special_file)
  {}  /* Can send either to WORKFILE or to SOUT, as long as SOUT is 
   * not RUNTTY. */
  dest = workfile;
  if (dest == NULL)
  {
    if (sout == RUNTTY)
      error (1, 0, "special file $a cannot be written to stdout", 
rcs->path);
    dest = sout;
  }

  /* Unlink 'dest', just in case. It's okay if this provokes a 
   * ENOENT error. */
  if (unlink (dest) < 0 && existence_error (errno))
    error (1, errno, "cannot remove $a", dest);
  if (mknod (dest, special_file, devnum) < 0)
    error (1, errno, "could not create special file $a", 
dest);
  endif
  else
  {
    /* Not a special file: write to WORKFILE or SOUT. */
    if (workfile == NULL)
      ofp = stdout;
    else
      /* Symbolic links should be removed before replacement, so that 
       * 'fopen' doesn't follow the link and open the wrong file. */
      if (unlink (sout))
        if (unlink_file (sout) < 0)
          error (1, errno, "cannot remove $a", sout);
        ofp = CVS_FOPEN (sout, expand == KFLAG_B ? "wb" : "w");
      if (ofp == NULL)
        error (1, errno, "cannot open $a", sout);
      else
        /* Output is supposed to go to WORKFILE, so we should open that 
         * file. Symbolic links should be removed first (see above). */
        if (unlink (workfile))
          if (unlink_file (workfile) < 0)
            error (1, errno, "cannot remove $a", workfile);
          ofp = CVS_FOPEN (workfile, expand == KFLAG_B ? "wb" : "w");
        /* If the open failed because the existing workfile was not
if (fwrite (p, 1, nstep, ofp) != nstep)
    { error (0, errno, "cannot write \%s", workfile);
        return 1;
    }
    p += nstep;
    nleft -= nstep;
    if (nleft < nstep)
        nstep = nleft;
}

if (free_value)
    free (value);

if (workfile != NULL && sout == RUNTTY)
    { if (expand == KFLAG_B)
        cv_output_binary (value, len);
    }
    else
    { /* cv_output requires the caller to check for zero
        length. */
        if (len > 0)
            cv_output (value, len);
    }
    else
    { /* NT 4.0 is said to have trouble writing 2099999 bytes
        (for example) in a single fwrite. So break it down
        (there is no need to be writing that much at once
        anyway; it is possible that LARGEST_FWRITE should be
        somewhat larger for good performance, but for testing I
        want to start with a small value until/unless a bigger
        one proves useful). */

define LARGEST_FWRITE 8192

size_t nleft = len;
size_t nstep = (len < LARGEST_FWRITE ? len : LARGEST_FWRITE);

char *p = value;

while (nleft > 0)
    { if (fwrite (p, 1, nstep, ofp) != nstep)
        { error (0, errno, "cannot write \%s", workfile:
            workfile == NULL
            ? (sout == RUNTTY ? sout : "stdout"));
            if (free_value)
                free (value);
            return 1;
        }
        p += nstep;
        nleft -= nstep;
        if (nleft < nstep)
            nstep = nleft;
    }

if (free_value)
    free (value);

if (workfile == NULL)
    { int ret;

define PRESERVE_PERMISSIONS_SUPPORT

if (specialfile && fclose (ofp) < 0)
    { error (0, errno, "cannot close \%s", workfile);
        return 1;
    }

if (change_rwxowner_rwxgroup)
    { if (change_rwxowner_rwxgroup)
        { error (0, errno, "could not change owner or group of \%s",
            workfile);
            return 1;
        }

    ret = chmod (workfile,
            change_rwxmode
            \%s
            mode
            : sb stm
            mode & ~ [S_1W | S_1WGRP | S_1WOTH])
    }

else
    if (fclose (ofp) < 0)
4860     { error (0, errno, "cannot close %s", workfile);
4861       return 1;
4862     }
4863
4864     ret = chmod (workfile,
4865                sb.st_mode & ^S_IFWRITE | S_IFGRP | S_IFOTH);
4866     #endif
4867     if (ret < 0)
4868     { error (0, errno, "cannot change mode of file %s",
4869              workfile);
4870     }
4871   else if (sout != RUN_TTY)
4872     { if (#ifdef PRESERVE_PERMISSIONS_SUPPORT
4873          !specialfile &&
4874      #endif fclose (ofp) < 0)
4875     { error (0, errno, "cannot close %s",
4876           sout);
4877       return 1;
4878     }
4879   #ifdef PRESERVE_PERMISSIONS_SUPPORT
4880     /* If we are in the business of preserving hardlinks, then
4881      mark this file as having been checked out. */
4882     if (preserve_perms && workfile != NULL)
4883     { update_hardlink_info (workfile);
4884      #endif
4885     return 0;
4886   }
4887
4888 static RCSVers *RCS_findlockor_tipPROTO (RCSNode *rcs);
4889
4890 /* Find the delta currently locked by the user. From the 'ci' man page:
4891   "If rev is omitted, ci tries to derive the new revision number from the
4892    caller's last lock. If the caller has locked the tip revision of a branch,
4893    the new revision is appended to that branch. The new revision number is
4894    obtained by incrementing the tip revision number. If the caller locked a
4895    non-tip revision, a new branch is started at that revision by incrementing
4896    the highest branch number at that revision. The default initial branch
4897    and level numbers are 1. If rev is omitted and the caller has no lock, but
4898    owns the file and locking is not set to strict, then the revision is
4899    appended to the default branch (normally the trunk; see the -b option of
4900    rcs(1))."
4901
4902 RCS_findlockor_tip finds the unique revision locked by the caller
4903 and returns its delta node. If the caller has not locked any
4904 revisions (and is permitted to commit to an unlocked delta, as
described above), return the tip of the default branch. */
4905
4906 static RCSVers *
4907 RCS_findlockor_tip (rcs)
4908     RCSNode *rcs;
4909     {
4910       char *user = getcaller();
4911       Node *lock, *p;
4912       List *locklist;
4913
4914       /* Find unique delta locked by caller. This code is very similar
4915        to the code in RCS_unlock – perhaps it could be abstracted
4916        into a RCS_findlock function. */
4917       locklist = RCS_getlocks (rcs);
4918       lock = NULL;
4919       for (p = locklist->list->next; p != locklist->list; p = p->next)
4920       { if (STREQ (p->data, user))
4921         { if (lock != NULL)
4922           { error (0, 0, \\n4923             cleanup
4924             multiple revisions locked by %s; please specify one",
4925             rcs->path, user);
4926             return NULL;
4927           }
4928           lock = p;
4929         }
4930       }
4931   if (lock != NULL)
4932     {
/ * Found an old lock, but check that the revision still exists. */
4950 p = findnode (rcs->versions, lock->key);
4951 if (p == NULL) {
4952    error (0, 0, "%s: can’t unlock nonexistent revision %s",
4953           rcs->path, lock->key);
4954    return NULL;
4955 } return (RCSVers *) p->data;
4960
4960 /* No existing lock. The RCS rule is that this is an error unless locking is nonstrict AND the file is owned by the current user. Trying to determine the latter is a portability nightmare in the face of NT, VMS, AFS, and other systems with non-unix-like ideas of users and owners. In the case of CVZ, we should never get here (as long as the traditional behavior of making sure to call RCS_lock persists). Anyway, we skip the RCS error checks and just return the default branch or head. The reasoning is that those error checks are to make users lock before a checkin, and we do that in other ways if at all anyway (e.g. rcunlock.pl). */
4970 p = findnode (rcs->versions, RCS_getbranch (rcs, rcs->branch, 0));
4971 return (RCSVers *) p->data;
4980
4980 /* Revision number string, R, must contain a ’.’.
4980 * Return a newly-malloc’d copy of the prefix of R up
4980 * to but not including the final ’.’. */
4990 static char *
4990 truncate_revnum (r)
4991 const char *r;
4992 { size_t len;
4993 char *new;
4994 char *dot = strchr (r, ‘.’);
4995 assert (dot);
4996 len = dot - r;
4997 new = xmalloc (len + 1);
4998 memcpy (new, r, len);
4999 *(new + len) = ‘\0’;
5000 return new;
5010
5010 /* Revision number strings, R and S, must each contain a ’.’.
5010 * R and S must be writable and must have the same number of dots.
5010 * Truncate R and S for the comparison, then restored them to their
5010 * original state. Return the result (see compare_revnums) of comparing R and S
5010 * ignoring differences in any component after the rightmost ’.’. */
5020 static int
5020 compare_truncated_revnums (r, s)
5021 char *r;
5022 char *s;
5023 { char *s0dot = truncate_revnum_in_place (r);
5024 char *s1dot = truncate_revnum_in_place (s);
5025 int cmp;
5026    assert (numdots (r) == numdots (s));
5027    cmp = compare_revnums (r, s);
5028    *s0dot = ‘.’;
5029    *s1dot = ‘.’;
5030 return cmp;
5030
5030 /* Return a malloc’d copy of the string representing the highest branch
5030 * number on BRANCHNODE. If there are no branches on BRANCHNODE, return NULL.
5030 * FIXME: isn’t the max rev always the last one? */
If so, we don’t even need a loop. */

static char *maxrev PROTO ((const RCSVers *));

static char *
maxrev (branchnode)
const RCSVers *branchnode;
{
    Node *head;
    Node *bp;
    char *max;

    if (branchnode->branches == NULL)
    {
        return NULL;
    }

    max = NULL;
    head = branchnode->branches->list;
    for (bp = head->next; bp != head; bp = bp->next)
    {
        if (max == NULL || compare_truncated_revnums (max, bp->key) < 0)
        {
            max = bp->key;
        }
    }
    assert (max);

    return truncate_revnum (max);
}

/*@ Create BRANCH in RCS’s delta tree. BRANCH may be either a branch
number or a revision number. In the former case, create the branch
with the specified number; in the latter case, create a new branch
rooted at node BRANCH with a higher branch number than any others.
Return the number of the tip node on the new branch. */

static char *
RCSaddbranch (rcs, branch)
RCSNode *rcs;
const char *branch;
{
    char *branchpoint, *newrevnum;
    Node *nodep, *bp;
    Node *marker;
    RCSVers *branchnode;

    /* Append to end by default. */
    marker = NULL;
    branchpoint = xstrdup (branch);

    if ((numdots (branchpoint) & 1) == 0)
    {
        truncate_revnum_in_place (branchpoint);
    }

    /* Find the branch rooted at BRANCHPOINT. */
    nodep = findnode (rcs->versions, branchpoint);
    if (nodep == NULL)
    {
        error (0, 0, "%s: can’t find branch point %s", rcs->path, branchpoint);
        return NULL;
    }
    branchnode = (RCSVers *) nodep->data;

    /* If BRANCH was a full branch number, make sure it is higher than MAX. */
    if ((numdots (branch) & 1) == 1)
    {
        if (branchnode->branches == NULL)
        {
            /* We have to create the first branch on this node, which means
               appending “.2” to the revision number. */
            newrevnum = (char *) xmalloc (strlen (branch) + 3);
            strcpy (newrevnum, branch);
            strcat (newrevnum, ".2”);
        }
        else
        {
            char *max = maxrev (branchnode);
            assert (max);
            newrevnum = increment_revnum (max);
        }
    }
    else
    {
        newrevnum = xstrdup (branch);
        if (branchnode->branches != NULL)
Node *head;
Node *bp;

/* Find the position of this new branch in the sorted list of branches. */
head = branchnode->branches->list;
for (bp = head->next; bp != head; bp = bp->next)
{
    char *dot;
    int found = 0;

    /* The existing list must be sorted on increasing revnum. */
    assert (bp->next == head
          || compare_truncated_revnnums (bp->key,
                                         bp->next->key) < 0);
    dot = truncate_revnnum_place (bp->key);
    found = (compare_revnnums (branch, bp->key) < 0);
    *dot = '.';

    if (found)
    {
        break;
    }
    marker = bp;
}
newrevnum = (char *) xrealloc (newrevnum, strlen (newrevnum) + 3);
strcat (newrevnum, ".1");

/* Add this new revision number to BRANCHPOINT's branches list. */
if (branchnode->branches == NULL)
    branchnode->branches = getlist();
bp = getnode();
bp->key = xstrdup (newrevnum);

/* Append to the end of the list by default, that is, just before the header node, 'list'. */
if (marker == NULL)
    marker = branchnode->branches->list;

int fail = insert_before (branchnode->branches, marker, bp);
assert (!fail);

return newrevnum;

/* Check in to RCSFILE with revision REV (which must be greater than the largest revision) and message MESSAGE (which is checked for legality). If FLAGS & RCS_FLAGS_DEAD, check in a dead revision. If FLAGS & RCS_FLAGS_QUIET, tell ci to be quiet. If FLAGS & RCS_FLAGS_MODTIME, use the working file's modification time for the checkin time. WORKFILE is the working file to check in from, or NULL to use the usual RCS rules for deriving it from the RCSFILE. If FLAGS & RCS_FLAGS_KEEPFILE, don't unlink the working file; unlinking the working file is standard RCS behavior, but is rarely appropriate for CVS. */

int RCS_checkin (rcs, workfile, message, rev, flags)
    RCSNode *rcs;
    char *workfile;
    char *message;
    char *rev;
    int flags;
{
    RCSVers *delta, *commitpt;
    Deltatext *dtext;
    Node *nodep;
    char *tmpfile, *changefile, *chtext;
    char *deltas;
    size_t buffsize;
    int bufen, chtexplen;
    int status, checkin, allocated_workfile;
    struct tm *ftm;
    time_t modtime;

7069 This function should almost exactly mimic the behavior of 'rcs ci'. The principal point of difference is the support here for preserving file ownership and permissions in the delta nodes. This is not a clean solution - precisely because it diverges from RCS's behavior - but it doesn't seem feasible to do this anywhere else in the code. [twp]
int adding_branch = 0;

commitpt = NULL;

if (rcs->flags & PARTIAL)
    RCS_reparsecfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);

/* Get basename of working file. Is there a library function to
do this? I couldn't find one. -twp */
if (workfile == NULL)
    return;

char *p;
int extlen = strlen (RCSEXT);
workfile = xstrdup (last_component (rcs->path));
p = workfile + (strlen (workfile) - extlen);
assert (strcmp (p, RCSEXT, extlen) == 0);
*p = '0';
allocated_workfile = 1;

/* Is the backend file a symbolic link? Follow it and replace the
filename with the destination of the link. */
while (islink (rcs->path))
{
    char *newname;

    #ifdef HAVE_READLINK
    /* The clean thing to do is probably to have each filesub.c
    implement this (with an error if not supported by the
    platform, in which case islink would presumably return 0).
    But that would require editing each filesub.c and so the
    expedient hack seems to be looking at HAVE_READLINK. */
    newname = xreadlink (rcs->path);
    #else
    error (1, 0, "internal error: islink doesn't like readlink");
    #endif

    if (isabsolute (newname))
    {
        free (rcs->path);
        rcs->path = newname;
    }
    else
    {
        char *oldname = last_component (rcs->path);
        int dirlen = oldname - rcs->path;
        char *fullnewname = xmalloc (dirlen + strlen (newname) + 1);
        strcpy (fullnewname, rcs->path, dirlen);
        strcpy (fullnewname + dirlen, newname);
        free (newname);
        dirlen = strlen (dirlen);
        free (rcs->path);
        rcs->path = fullnewname;
    }

    checking_quiet = flags & RCS_FLAGS_QUIET;
    if (checking_quiet)
    {
        cvs_output (rcs->path, 0);
        cvs_output ("* < - -", 7);
        cvs_output ("\"", 1);
    }

    /* Create new delta node. */
    delta = (RCSVers **) xmalloc (sizeof (RCSVers));
    memset (delta, 0, sizeof (RCSVers));
    delta->author = xstrdup (getcaller ());
    if (flags & RCS_FLAGS_MODTIME)
    {
        struct stat ws;
        if (stat (workfile, &ws) < 0)
        {
            error (1, errno, "cannot stat %s", workfile);
        } else
        {
            modtime = ws.st_mtime;
        }
    } else
    {
        (void) time (&modtime);
        ftm = gmtime (&modtime);
    }

    delta->date = (char *)__xmalloc (MAXDATELEN);
    (void) sprintf (delta->date, DATEFORM,
        ftm->tm_year + (ftm->tm_year < 1900 ? 1900 : 0 : 1900),
        ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
        ftm->tm_min, ftm->tm_sec);
    if (flags & RCS_FLAGS_DEAD)
    {
        delta->state = xstrdup (RCSDEAD);
        delta->dead = 1;
    } else
delta->state = xstrdup("Exp");

#define PRESERVE_PERMISSIONS
/* If permissions should be preserved on this project, then
save the permission info. */
if (preserve_perms)
{
    Node enp;
    struct stat sb;
    char buf[64]; /* static buffer should be safe: see usage. -tup */
    delta->other_delta = getlist();
    
    if (CVS_STAT(workfile, &sb) < 0)
        error (0, 1, "cannot liststat %s", workfile);

    if (S_ISLNK (sb_st_mode))
    {
        np = getnode();
        np->key = xstrdup("*symlink*");
        np->data = xreadlink (workfile);
        addnode (delta->other_delta, np);
    }
    else
    {
        (void) strncpy (buf, "%s", sb_st_uid);
        np = getnode();
        np->key = xstrdup("*owner*");
        np->data = xstrdup (buf);
        addnode (delta->other_delta, np);
    }

    (void) strncpy (buf, "%s", sb_st_gid);
    np = getnode();
    np->key = xstrdup("*group*");
    np->data = xstrdup (buf);
    addnode (delta->other_delta, np);

    (void) strncpy (buf, "%s", sb_st_mode & S_IFMT);
    np = getnode();
    np->key = xstrdup("*permissions*");
    np->data = xstrdup (buf);
    addnode (delta->other_delta, np);

    /* Save device number. */
    switch (sb_st_mode & S_IFMT)
    {
    case S_IFREG: break;
    case S_IFCHR: break;
    case S_IFBLK: break;

    np = getnode();
    np->key = xstrdup("*special*");
    strncpy (buf, "%s", sb_st_mode & S_IFMT);
    if (sb_st_mode & S_IFMT) == S_IFCHR
        "*character": "Block",
    (unsigned long) sb_st_dev);
    np->data = xstrdup (buf);
    addnode (delta->other_delta, np);
    } } break;

default:
    error (0, 0, "special file %s has unknown type", workfile);
}

    /* Save hardlinks. */
    delta->hardlinks = listlinkedfilelist (workfile);
}
#endif

/* Create a new deltatext node. */
dtext = (Deltatext *) xmalloc (sizeof (Deltatext));
memset (dtext, 0, sizeof (Deltatext));

if (delta->head == NULL)
{
    char *newrev;
    FILE *fout;

    /* Figure out what the first revision number should be. */
    if (rev == NULL || *rev == '0')
        newrev = xstrdup ("1.1");
    else if (numdots (rev) == 0)
    {

/* If the delta tree is empty, then there's nothing to link the
new delta into. So make a new delta tree, snarf the working
file contents, and just write the new RCS file. */
}
newrev = (char *) malloc(strlen (rev) + 3);
strcpy (newrev, rev);
strcpy (newrev, "+1");
}

else
newrev = xstrdup (rev);

/* Don't need to xstrdup NEWREV because it's already dynamic, and
not used for anything else. (Don't need to free it, either.) */
rcs->head = newrev;
delta->version = xstrdup (newrev);
	nodep = getnode();
nodep->type = RCSVERS;
nodep->key = xstrdup (newrev);
nodep->data = (char *) delta;
(void) addnode (rcs->versions, nodep);

dtext->version = xstrdup (newrev);
bufsize = 0;
getfile (workfile, workfile,
rcs->expand != NULL && STREQ (rcs->expand, "b") ? "b" : "a",
&dtext->text, &bufsize, &dtext->len);

if (checkin_quiet)
{

cvs_output ("initial revision: ", 0);

cvs_output (rcs->head, 0);

cvs_output ("a", 1);
}

/* We are probably about to invalidate any cached file. */
rebuild_cache_close ();

fout = rcs_internal_lockfile (rcs->path);
RCScoutmin (rcs, fout);
RCScouttree (rcs, rcs->head, fout);
RCScouttree (rcs, fout);
rcs->delnumos = ftell (fout);

if (rcs->delnumos == -1)
error (1, errno, "cannot ftell for %s", rcs->path);
putdeltatext (fout, dtext);
rcs_internal_unlockfile (fout, rcs->path);
freedeltatext (dtext);

if ((flags & RCS_FLAGS_KEEPFILE) == 0)
{
	if (unlink_file (workfile) < 0)
		/* FIXME-update-dir: message does not include update_dir. */
		error (0, errno, "cannot remove %s", workfile);
}

if (checkin_quiet)
cvs_output ("Done/a", 5);

return 0;

/* Derive a new revision number. From the 'ci' man page:

"If rev is a revision number, it must be higher than the
latest one on the branch to which rev belongs, or must
start a new branch.

If rev is a branch rather than a revision number, the new
revision is appended to that branch. The level number is
obtained by incrementing the tip revision number of that
branch. If rev indicates a non-existing branch, that
branch is created with the initial revision numbered
rev.1."

RCS_findlock_or_tip handles the case where REV is omitted.
RCS 5.7 also permits REV to be "$" or to begin with a dot, but
we do not address those cases -- every routine that calls
RCS_checkin passes it a numeric revision. */

if (rev == NULL || *rev == '0')
{
	/* Figure out where the commit point is by looking for locks.
If the commit point is at the tip of a branch (or is the
head of the delta tree), then increment its revision number
to obtain the new revision. Otherwise, start a new
branch. */
	commitpt = RCS_findlock_or_tip (rcs);
	if (commitpt == NULL)
	{
		status = 1;
	
goto checking_done;
	}
	else if (commitpt->next == NULL
		|| STREQ (commitpt->version, rcs->head))
/* REV is either a revision number or a branch number. Find the tip of the target branch. */

int dots, isrevnum;

newrev = xstrdup (rev);
dots = numdots (newrev);

nodep = findnode (rcs->versions, tip);
getlocks

name

temp
temp
exec
branch
done
checkout
name

commitpt

5650
5660
5670
5680
5690
5700
5710
5720
5730
5740
5750
5760
5770
5780
5790
5800
5810
5820
5830
5840
5850
5860
5870
5880
5890
5900
5910
5920
5930
5940
5950
5960
5970
5980
5990
6000
6010
6020
6030
6040
6050
6060
6070
6080
6090
6100
6110
6120

commitpt = (RCSVers *) nodep->data;

free (branch);
free (newrev);
free (tip);
}

assert (delta->version != NULL);

/* If COMMITPT is locked by us, break the lock. If it's locked
by someone else, signal an error. */
nodep = findnode (RCS_getlocks (rcs), commitpt->version);
if (nodep != NULL)
{
  if (! STREQ (nodep->data, delta->author))
  {
    /* If we are adding a branch, then leave the old lock around.
That is sensible in the sense that when adding a branch,
we don't need to use the lock to tell us where to check
in. It is fishy in the sense that if it is our own lock,
we break it. However, this is the RCS 5.7 behavior (at
the end of addbranch in ci.c in RCS 5.7, it calls
removevlock only if it is our own lock, not someone
else's). */
    if (ladding_branch)
    {
      error (0, 0, "Ja: revision %s locked by %s",
            rcs->path, nodep->key, nodep->data);
      status = 1;
      goto checkin_done;
    }
    else
delnode (nodep);
}
dtext->version = xstrdup (delta->version);

/* Obtain the change text for the new delta. If DELTA is to be the
new head of the tree, then its change text should be the contents
of the working file, and LEAFNODE's change text should be a diff.
Else, DELTA's change text should be a diff between LEAFNODE and
the working file. */

tmpfile = cvstemp_name();
status = RCS_checkout (rcs, NULL, commitpt->version, NULL,
((rcs->expand != NULL && STREQ (rcs->expand, "b"))
  ? "-a -b"
  : "-a"),
tmpfile,
(RCSCHECKOUTPROC)0, NULL);
if (status != 0)
  error (1, 0, "could not check out revision %s of %s",
        commitpt->version, rcs->path);
bufsize = buflen = 0;
chtext = NULL;
chtextlen = 0;
changefile = cvstemp_name();

/* Diff options should include --binary if the RCS file has-kb set
in its 'expand' field. */
diffopts = (rcs->expand != NULL && STREQ (rcs->expand, "b")
  ? "-a -b --binary"
  : "-a --binary");
if (STREQ (commitpt->version, rcs->head) &&
numdots (delta->version) == 1)
{
  /* If this revision is being inserted on the trunk, the change text
  for the new delta should be the contents of the working file... */
  bufsize = 0;
  get_file (workfile, workfile,
            rcs->expand != NULL && STREQ (rcs->expand, "b") ? "rb"
            : "r",
            &dtext->text, &bufsize, &dtext->len);
  /* ... and the change text for the old delta should be a diff. */
  commitpt->text = (Deltatext *) xmalloc (sizeof (Deltatext));
  memset (commitpt->text, 0, sizeof (Deltatext));
  bufsize = 0;
  switch (difexec (workfile, tempfile, diffopts, changefile))
  {
    case 0:
    case 1:
      break;
case -1:
  /* FIXME-update-dir: message does not include update_dir. */
  error (1, errno, "error diffing %s", workfile);
  break;
default:
  /* FIXME-update-dir: message does not include update_dir. */
  error (1, 0, "error diffing %s", workfile);
  break;
}

/* OK, the text file case here is really dumb. Logically
speaking we want diff to read the files in text mode,
convert them to the canonical form found in RCS files
(which, we hope at least, is independent of OS–always
bare linefeeds), and then work with change texts in that
format. However, diff–see both generates change
texts and produces output for user purposes (e.g. patch.c),
and there is no way to distinguish between the two cases.
So we actually implement the text file case by reading the
change text as a text file, then reading it as a text file.
This should cause no harm, but doesn't strike me as
immensely clean. */

get_file (changefile, changefile,
  rcs->expand != NULL &amp; STREQ (rcs->expand, "b") ? "rb" : "r",
  &commitpt->text-&gt;text, &bufsize, &commitpt-&gt;text-&gt;len);

/* If COMMITPT-&gt;TEXT-&gt;TEXT is NULL, it means that CHANGEFILE
was empty and that there are no differences between revisions.
In that event, we want to force RCS–write to write an empty
string for COMMITPT’s change text. Leaving the change text
field set NULL won’t work, since that means “preserve the original
change text for this delta.” */
if (commitpt-&gt;text-&gt;text == NULL)
{
  commitpt-&gt;text-&gt;text = xstrdup (**);
  commitpt-&gt;text-&gt;len = 0;
}
else
{
/* This file is not being inserted at the head, but on a side
branch somewhere. Make a diff from the previous revision
to the working file. */
switch (diffexec (tmpfile, workfile, diffopts, changefile))
{
  case 0:
  case 1:
    break;
  case -1:
    /* FIXME-update-dir: message does not include update_dir. */
    error (1, errno, "error diffing %s", workfile);
  break;
default:
  /* FIXME-update-dir: message does not include update_dir. */
  error (1, 0, "error diffing %s", workfile);
  break;
}
/* See the comment above, at the other get_file invocation,
regarding binary vs. text. */
get_file (changefile, changefile,
  rcs-&gt;expand != NULL &amp; STREQ (rcs-&gt;expand, "b") ? "rb" : "r",
  &text-&gt;text, &bufsize, &text-&gt;len);
if (dtext-&gt;text == NULL)
{
  dtext-&gt;text = xstrdup (**);
  dtext-&gt;len = 0;
}
}

/* Update DELTA linkage. It is important not to do this before
the very end of RCS-checkin; if an error arises that forces
us to abort checking in, we must not have malformed deltas
partially linked into the tree.
If DELTA and COMMITPT are on different branches, do nothing.
DELTA is linked to the tree through COMMITPT-BRANCHES, and we
don’t want to change ‘next’ pointers.
Otherwise, if the nodes are both on the trunk, link DELTA to
COMMITPT; otherwise, link COMMITPT to DELTA. */
if (numdots (commitpt-&gt;version) == numdots (delta-&gt;version))
{
  if (STREQ (commitpt-&gt;version, rcs-&gt;head))
  {
    delta-&gt;next = rcs-&gt;head;
    rcs-&gt;head = xstrdup (delta-&gt;version);
  }
else

commitpt->next = xstrdup (delta->version);

5760 }

/* Add DELTA to RCS>VERSIONS. */
if (rcs->versions == NULL)
    rcs->versions = getlist();
nodep = getnode();
nodep->type = RCSVERS;
nodep->key = xstrdup (delta->version);
nodep->data = (char *) delta;
(void) addnode (rcs->versions, nodep);

5770 */ Write the new RCS file, inserting the new delta at COMMITPT. */
if (checkinquiet)
    {
        cvsoutput ("new revision: ", 14);
        cvsoutput (delta->version, 0);
        cvsoutput ("; previous revision: ", 21);
        cvsoutput (commitpt->version, 0);
        cvsoutput ("\n", 1);
    }

5780 RCS_rewrite (rcs, dtext, commitpt->version);
if ((flags & RCS_FLAGS_KEEPFILE) == 0)
    {
        if (!unlink_file (workfile))
            /* FIXME-update-dir: message does not include update-dir. */
            error (1, errno, "cannot remove %s", workfile);
        if (!unlink_file (tmpfile))
            error (0, errno, "cannot remove %s", tmpfile);
        if (!unlink_file (changedfile))
            error (0, errno, "cannot remove %s", changedfile);
        if (checkinquiet)
            cvsoutput ("done\n", 5);
        checkin_done;
        if (!allocated_workfile)
            free (workfile);

5800 if (commitpt != NULL && commitpt->text != NULL)
    {
        freedeltatext (commitpt->text);
        commitpt->text = NULL;
    }
    freedeltatext (dtext);
    if (status != 0)
        free_rcsvers_contents (delta);

5810 return status;

    */ This structure is passed between RCS_cmpfile and cmp_file_buffer. */
struct cmp_file_data
    {
        const char *filename;
        FILE *fp;
        int different;
    };

    */ Compare the contents of revision REV of RCS file RCS with the
    contents of the file FILENAME. OPTIONS is a string for the keyword
    expansion options. Return 0 if the contents of the revision are
    the same as the contents of the file, 1 if they are different. */
int:
    RCS_cmp_file (rcs, rev, options, filename)
      RCSNode *rcs;
      char *rev;
      char *options;
      const char *filename;
      { int binary;
        FILE *fp;
        struct cmp_file_data data;
        int retcode;

5840 if (options != NULL && options[0] != '\0')
    binary = STREQ (options, "-b");
else
    { char *expand;
      expand = RCS_getexpand (rcs);
      if (expand != NULL && STREQ (expand, "b"))
        binary = 1;
else
    binary = 0;
}

#define PRESERVE_PERMISSIONS_SUPPORT
/* If CVS is to deal properly with special files (when
PreservePermissions is on), the best way is to check out the
revision to a temporary file and call `xcmp' on the two disk
files. `xcmp' needs to handle non-regular files properly anyway,
so calling it simplifies RCS's cmp_file. We could just yank the
delta node out of the version tree and look for device
numbers, but writing to disk and calling xcmp is a better
abstraction (therefore probably more robust). */
if (preserve_perms)
{
    char *tmp;

    tmp = cvs_cmp_file();
    retcode = RCS_checkout(rcs, NULL, NULL, options, tmp, NULL, NULL);
    if (retcode != 0)
        return 1;

    retcode = xcmp (tmp, filename);
    if (CVS_UNLINK (tmp) < 0)
        error (1, errno, "cannot remove %s", tmp);
    return retcode;
}
else
#endif

fp = CVS_FOPEN (filename, binary ? FOPEN_BINARY_READ : "r");
if (fp == NULL)
    /* FIXME: update-dir: should include update_dir in message. */
    error (1, errno, "cannot open file %s for comparing", filename);

    data.filename = filename;
    data.fp = fp;
    data.different = 0;

    retcode = RCS_checkout (rcs, (char *) NULL, (char *) NULL, NULL,
                            options, RUN_TTY, cmp_file_buffer,
                            (void *) &data);

    /* If we have not yet found a difference, make sure that we are at
    the end of the file. */
    if (! data.different)
    {
        if (fgets (fp) == EOF)
            data.different = 1;
    }

fclose (fp);
if (retcode != 0)
    return 1;
return data.different;

/* This is a subroutine of RCS_cmp_file. It is passed to
RCS_checkout. */
#define CMP_BUF_SIZE (8 * 1024)

static void
cmp_file_buffer (callerdat, buffer, len)
    void *callerdat;
    const char *buffer;
    size_t len;
{
    struct cmp_file_data *data = (struct cmp_file_data *) callerdat;
    char *filebuf;
    /* If we've already found a difference, we don't need to check
    further. */
    if (data->different)
        return;

    filebuf = xmalloc (len > CMP_BUF_SIZE ? CMP_BUF_SIZE : len);
    while (len > 0)
    {
        size_t checklen;
        checklen = len > CMP_BUF_SIZE ? CMP_BUF_SIZE : len;
        if (fread (filebuf, 1, checklen, data->fp) != checklen)
        {
            if (ferror (data->fp))
error (1, errno, "cannot read file %s for comparing",
      data->filename);
data->different = 1;
free (filebuf);
return;
}
if (memcmp (filebuf, buffer, checklen) != 0)
{
data->different = 1;
free (filebuf);
return;
}
buffer += checklen;
len -= checklen;
}
free (filebuf);
}
/* For RCS file RCS, make symbolic tag TAG point to revision REV. 
This validates that TAG is OK for a user to use. Return value is
-1 for error (and errno is set to indicate the error), positive for
error (and an error message has been printed), or zero for success. */
int
RCS_settag (rcs, tag, rev)
RCSNode *rcs;
const char *tag;
const char *rev;
{
List *symbols;
Node *node;
if (rcs->flags & PARTIAL)
RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuf *&) NULL);
/* FIXME: This check should be moved to RCS_check_tag. There is no
reason for it to be here. */
if (STREQ (tag, RCS_BASE) || STREQ (tag, RCS_HEAD))
{
/* Print the name of the tag might be considered redundant
with the caller, which also prints it. Perhaps this helps
clarify why the tag name is considered reserved, I don't
know. */
error (0, 0, "Attempt to add reserved tag name %s", tag);
return 1;
}
/* A revision number of NULL means use the head or default branch.
If rev is not NULL, it may be a symbolic tag or branch number;
expand it to the correct numeric revision or branch head. */
if (rev == NULL)
rev = rcs->branch ? rcs->branch : rcs->head;
/* At this point rcs->symbol->data may not have been parsed.
Calling RCS_symbols will force it to be parsed into a list
which we can easily manipulate. */
symbols = RCS_symbols (rcs);
if (symbols == NULL)
{
symbols = getlist ();
rcs->symbols = symbols;
}
node = findnode (symbols, tag);
if (node != NULL)
{
free (node->data);
node->data = xstrdup (rev);
}
else
{
node = getnode ();
node->key = xstrdup (tag);
node->data = xstrdup (rev);
(void) addnode_atfront (symbols, node);
}
return 0;
/* For RCS file RCS, make symbolic tag TAG point to revision REV. 
This validates that TAG is OK for a user to use. Return value is
-1 for error (and errno is set to indicate the error), positive for
error (and an error message has been printed), or zero for success. */
int
RCS_setremotetag (rcs, tag, rev, remote)
RCSNode *rcs;
const char *tag;
const char *rev;
const char *remote;

List* remote_branches;
Node* node;
Node* versionnode;
RCSVers* version;
char* branchpoint;
char* p;

int c = 0;

/* Set up the association between the tag and the branch as normally */
int retval = RCS_settag (rcs, tag, rev);
if (retval != 0) { return retval; }

branchpoint = xstrdup (rev);
for (p = branchpoint + strlen (branchpoint); p > branchpoint; p -= 2) {
    if (*p == ' ' || *p == '\

    key = xmalloc (strlen (rev) + strlen (remote) + 3);
    sprintf (node->key, "RCS\n
    node = getnode ();
    node->key = xmalloc (strlen (rev) + strlen (remote) + 3);
    sprintf (node->key, "RCS\n
    node->data = xstrdup (**);
    addnode_at_front (remote_branches, node);
    return 0;

    } /* Add the branch to the list of remote branches */

versionnode = findnode (rcs -> versions, branchpoint);
version = (RCSVers *) (versionnode -> data);
remote_branches = version -> remote_branches;
if (remote_branches == NULL) {
    remote_branches = version -> remote_branches = getlist ();
}

node = getnode ();
node->key = xmalloc (strlen (rev) + strlen (remote) + 3);
sprintf (node->key, "RCS\n
node->data = xstrdup (**);
addnode_at_front (remote_branches, node);
return 0;

    } /* Delete the symbolic tag TAG from the RCS file RCS. Return 0 if */
    /* the tag was found (and removed), or 1 if it was not present. (In */
    /* other case, the tag will no longer be in RCS->SYMBOLS) */

int
RCS_deltag (rcs, tag)
RCSNode *rcs;
const char *tag;
{
    List *symbols;
    Node *node;
    if (!rcs->flags & PARTIAL)
        RCS_parsersfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);
    symbols = RCS_symbols (rcs);
    if (symbols == NULL)
        return 1;
    node = findnode (symbols, tag);
    if (node == NULL)
        return 1;
    delnode (node);

    return 0;

    } /* Set the default branch of RCS to REV. */

int
RCS_setbranch (rcs, rev)
RCSNode *rcs;
const char *rev;
{
    if (!rcs->flags & PARTIAL)
        RCS_parsersfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);
    if (!rev & rcs->branch)
        rev = NULL;
    if (!rev && rcs->branch) return 0;
    if (!rev && !rcs->branch) return NULL && STREQ (rev, rcs->branch))
return 0;

if (rcs->branch != NULL)
    free (rcs->branch);
rcs->branch = xstrdup (rev);
return 0;
}

/* Lock revision REV. LOCKQUIET is 1 to suppress output. FIXME:
   Most of the callers only call us because RCScheckin still tends to
   like a lock (a relic of old behavior inherited from the RCS cs
   program). If we clean this up, only "cvs admin -l" will still need
   to call RCSlock. */
/* FIXME-twp: if a lock owned by someone else is broken, should this
   send mail to the lock owner? Prompt user? It seems like such an
   obscure situation for CVS as almost not worth worrying much
   about. */

int
RCSlock (rcs, rev, lockquiet)
RCSNode *rcs;
const char *rev;
int lockquiet;
{
    List *locks;
    Node *p;
    char *user;
    char *xrev = NULL;

    if (rcs->flags & PARTIAL)
        RCSreparsercsfile (rcs, (FILE **)NULL, (struct rcsbuffer *)NULL);
    locks = RCSgetlocks (rcs);
    if (locks == NULL)
        locks = rcs->locks = getlist();
    user = getcaller();
    /* A revision number of NULL means lock the head or default branch. */
    if (rev == NULL)
        xrev = RCShead (rcs);
    /* If rev is a branch number, lock the latest revision on that
       branch. I think that if the branch doesn’t exist, it’s
       okay to return 0 — that just means that the branch is new,
       so we don’t need to lock it anyway. -twp */
    else if (RCSnodeisbranch (rcs, rev))
    { xrev = RCSgetbranch (rcs, (char *)rev, 1);
        if (xrev == NULL)
            { if (!lockquiet)
                error (0, 0, "%s: branch %s absent", rcs->path, rev);
                    free (xrev);
                return 1;
            }  // else if (xrev == NULL)
        xrev = RCShead (rcs);
    }
    /* Make sure that the desired revision exists. Technically,
       we can update the locks list without even checking this,
       but RCS 5.7 did this. And it can’t hurt. */
    if (findnode (rcs->versions, xrev) == NULL)
    { if (!lockquiet)
        error (0, 0, "%s: revision %s absent", rcs->path, xrev);
            free (xrev);
        return 1;
    }
    /* Is this rev already locked? */
    p = findnode (locks, xrev);
    if (p != NULL)
    { if (strcmp (p->data, user))
        { /* We already own the lock on this revision, so do nothing. */
            free (xrev);
            return 0;
        }  // else if (strcmp (p->data, user))
    }
    /* Well, first of all, “rev” below should be “xrev” to avoid
       core dumps. But more importantly, should we really be
       breaking the lock unconditionally? What CVS 1.9 does (via
       RCS) is to prompt “Revision 1.1 is already locked by fred.
       Do you want to break the lock? [ny]”. -twp: we don’t want to interact
       with the user (certainly not at the
/* Break the lock. */
if (!lockquiet)
    delnode(p);
else
delnode_error(1, 0, "Revision %s is already locked by %s", xrev, p->data);
dendif

/* Create a new lock. */
p = getnode();
p->key = xrev; /* already xstrdupped */
p->data = xstrdup(getcaller());
(void)addnode_at_front locks, p;

/* Unlock revision REV. UNLOCKQUIET is 1 to suppress output. FIXME:
   Like RCSlock, this can become a no-op if we do the checkin ourselves.
   If REV is not null and is locked by someone else, break their lock
   and notify them. It is an open issue whether RCSunlock queries the user
   about whether or not to break the lock. */
int RCSunlock(rcs, rev, unlockquiet)
    RCSNode *rcs;
    const char *rev;
    int unlockquiet;
{
    Node *lock;
    List *locks;
    char *user;
    char *xrev = NULL;

user = getcaller();
if (rcs->flags & PARTIAL)
    RCSreparsersfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);

/* If rev is NULL, unlock the latest revision (first in
rcs->locks) held by the caller. */
if (rev == NULL)
    {
    Node *p;

    /* No-ops: attempts to unlock an empty tree or an unlocked file. */
    if (rcs->head == NULL)
        {
        if (!unlockquiet)
            cvsoutput ("can't unlock an empty tree\n", 0);
        return 0;
        }
    locks = RCSgetlocks (rcs);
    if (locks == NULL)
        {
        if (!unlockquiet)
            cvsoutput ("No locks are set.\n", 0);
        return 0;
        }
    lock = NULL;
    for (p = locks->list->next; p != locks->list; p = p->next)
        {
        if (STREQ (p->data, user))
            {
            if (lock != NULL)
                {
                if (!unlockquiet)
                    error (0, 0, "; %s: multiple revisions locked by %s; please specify one", rcs->path, user);
                return 1;
                }
            lock = p;
        }
if (lock == NULL)
    return 0; /* no lock found, ergo nothing to do */
xrev = x_strdup(lock->key);
}
else if (RCS_nodeisbranch(rcs, rev))
    /* If rev is a branch number, unlock the latest revision on that branch. */
xrev = RCS_getbranch(rcs, (char *) rev, 1);
if (xrev == NULL)
    error(0, 0, "%s: branch %s absent", rcs->path, rev);
    return 1;
}
else
    /* REV is an exact revision number. */
xrev = x_strdup(rev);
lock = findnode(RCS_getlocks(rcs), xrev);
if (lock == NULL)
    /* This revision isn’t locked. */
    free(xrev);
    return 0;
else
    if (!STREQ(lock->data, user))
        /* If the revision is locked by someone else, notify them. Note that this shouldn’t ever happen if RCS_unlock is called with a NULL revision, since that means “whatever revision is currently locked by the caller.” */
        char *repos, *workfile;
        repos = x_strdup(rcs->path);
        workfile = strchr(repos, '/');
        workfile += 1;
        notify_do(“C”, workfile, user, NULL, NULL, repos);
        free(repos);
    delnode(lock);
    if (!unlock_quiet)
        { cva_output(xrev, 0);
          cva_output(" unlocked
", 0);
        }
    free(xrev);
    return 0;
}
/* Add USER to the access list of RCS. Do nothing if already present. FIXME-twp: check syntax of USER to make sure it’s a valid id. */
void
RCS_addaccess(rcs, user)
RCSNode *rcs;
char *user;
{
    char *access, *a;
    if (rcs->flags & PARTIAL)
        RCS_parsersfcfile(rcs, FILE **) NULL, (struct rebuffer *) NULL);
    if (rcs->access == NULL)
        rcs->access = x_strdup(user);
    else
        { access = x_strdup(rcs->access);
          for (a = strtok(access, " "); a != NULL; a = strtok(NULL, " "))
            if (STREQ(a, user))
                { free(access);
                  return;
                }
            rcs->access = (char *) xrealloc(rcs->access, strlen(rcs->access) + strlen(user) + 2);
            strcat(rcs->access, " ");
            strcat(rcs->access, user);
        }
    /* Remove USER from the access list of RCS. */
    void
RCS_setaccess (rcs, user)

RCSNode *rcs;
char *user;
{
char *p, *s;
int ulen;

if (rcs->flags & PARTIAL)
    RCS_parsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

if (rcs->access == NULL)
    return;

p = rcs->access;
ulen = strlen (user);
while (p != NULL)
    {
    if (p[ulen] == '0' || p[ulen] == ' ')
        break;
    if (strncmp (p, user, ulen) == 0)
        break;
    p = strchr (p, ' ');
    if (p == NULL)
        return;
    s = p + uen;
    while (**s != '0')
        {++p;
        }
    if (p == NULL)
        return;
    p = s + uen;

    char *
    RCS_getaccess (rcs)
    RCSNode *rcs;
    {
    if (rcs->flags & PARTIAL)
        RCS_parsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);
    }

static int findtagPROTO ((Node *, void *));
/

static int
findtag (node, arg)
Node *node;
void *arg;
{
char *rev = (char *)arg;
    if (STREQ (node->data, rev))
        return 1;
    else
        return 0;
}

int
RCS_delete_revs (rcs, tag1, tag2, inclusive)
RCSNode *rcs;
char *tag1;
char *tag2;
int inclusive;
{
char *next;
Node *nodep;
RCSVers *revp = NULL;

/* Delete revisions between REV1 and REV2. The changes between the two
revisions must be collapsed, and the result stored in the revision
immediately preceding the lower one. Return 0 for successful completion,
1 otherwise.

Solution: check out the revision preceding REV1 and the revision
following REV2. Use call_diff to find aggregate diffs between
these two revisions, and replace the delta text for the latter one
with the new aggregate diff. Alternatively, we could write a
function that takes two change texts and combines them to produce a
new change text, without checking out any revs or calling diff. It
would be hairy, but so, so cool.

If INCLUSIVE is set, then TAG1 and TAG2, if non-NULL, tell us to
delete that revision as well (cvs admin -o tag1:tag2). If clear,
delete up to but not including that revision (cvs admin -o tag1::tag2).
This does not affect TAG1 or TAG2 being NULL, the meaning of the start
point in ::tag2 and :tag2 is the same and likewise for end points. */

int
RCS_delete_revs (rcs, tag1, tag2, inclusive)
RCVers *beforep;
#define RCSVERS
int status, found;
int save_noexec;
char *branchpoint = NULL;
char *rev1 = NULL;
int rev1_inclusive = inclusive;
int rev2_inclusive = inclusive;
char *before = NULL;
char *after = NULL;
char *beforefile = NULL;
char *afterfile = NULL;
char *soutfile = NULL;

if (tag1 == NULL && tag2 == NULL)
  return 0;

/* Make sure both revisions exist. */
if (tag1 != NULL)
    {
      rev1 = RCS_gettag (rcs, tag1, 1, NULL);
      if (rev1 == NULL | |
          (nodep = findnode (rcs->versions, rev1)) == NULL)
          {
            error (0, 0, "%s: Revision %s doesn't exist.", 
                  rcs->path, tag1);
            goto delrev_file;
          }
    }

if (tag2 != NULL)
    {
      rev2 = RCS_gettag (rcs, tag2, 1, NULL);
      if (rev2 == NULL | |
          (nodep = findnode (rcs->versions, rev2)) == NULL)
          {
            error (0, 0, "%s: Revision %s doesn't exist.", 
                  rcs->path, tag2);
            goto delrev_file;
          }
    }

/* If rev1 is on the trunk and rev2 is NULL, rev2 should be 
   RCS->HEAD. (Note RCS_head(rcs), which may return rcs->branch 
   instead.) We need to check this special case early, in order 
   to make sure that rev1 and rev2 get ordered correctly. */
if (rev2 == NULL && numdots (rev1) == 1)
    {
      rev2 = xstrdup (rcs->head);
      rev2_inclusive = 1;
    }

if (rev2 == NULL)
  rev2_inclusive = 1;

if (rev1 != NULL && rev2 != NULL)
    {
      /* A range consisting of a branch number means the latest revision 
         on that branch. */
      if (RCS_isbranch (rcs, rev1) && STREQ (rev1, rev2))
        rev1 = rev2 = RCS_getbranch (rcs, rev1, 0);
      else
        {
          /* Make sure REV1 and REV2 are ordered correctly (in the 
             same order as the next field). For revisions on the 
             trunk, REV1 should be higher than REV2; for branches, 
             REV1 should be lower. */
          /* Shouldn't we just be giving an error in the case where 
             the user specifies the revisions in the wrong order 
             (that is, always swap on the trunk, never swap on a 
             branch, in the non-error cases)? It is not at all 
             clear to me that users who specify -o 1.4:1.2 really 
             meant to type -o 1.2:1.4, and the out of order usage 
             has never been documented, either by cvs.texinfo or 
             rcs(1). */
          char *temp;
          int temp_inclusive;
          if (numdots (rev1) == 1)
            {
              if (compare_revnurns (rev1, rev2) <= 0)
                {
                  temp = rev2;
                  rev2 = rev1;
                  rev1 = temp;
                  temp_inclusive = rev2_inclusive;
                  rev2_inclusive = rev1_inclusive;
                  rev1_inclusive = temp_inclusive;
                }
            }
          else if (compare_revnurns (rev1, rev2) > 0)
```c
{
    temp = rev2;
    rev2 = rev1;
    rev1 = temp;
    temp_inclusive = rev2_inclusive;
    rev2_inclusive = rev1_inclusive;
    rev1_inclusive = temp_inclusive;
}
}

/* Basically the same thing; make sure that the ordering is what we
need. */
if (rev1 == NULL)
{
    assert (rev2 != NULL);
    if (numdots (rev2) == 1)
    {
        /* Swap rev1 and rev2. */
        int temp_inclusive;
        rev1 = rev2;
        rev2 = NULL;
        temp_inclusive = rev2_inclusive;
        rev2_inclusive = rev1_inclusive;
        rev1_inclusive = temp_inclusive;
    }
}

/* Put the revision number preceding the first one to delete into
BEFORE (where “preceding” means according to the next field).
If the first revision to delete is the first revision on its
branch (e.g. 1.3.2.1), BEFORE should be the node on the trunk
at which the branch is rooted. If the first revision to delete
is the head revision of the trunk, set BEFORE to NULL.

Note that because BEFORE may not be on the same branch as REV1,
it is not very handy for navigating the revision tree. It’s
most useful just for checking out the revision preceding REV1. */
if (rev1 == NULL)
{
    if (findnode (rcs, rev1 != NULL ? rev1 : rev2) == NULL)
    {
        rev1 = xstrdup (branchpoint);
        if (numdots (branchpoint) > 1)
        {
            char *bp;
            bp = strrchr (branchpoint, '\');
            while (--bp != '\')
            {
                *bp = '\0';
                /* Note that this is exclusive, always, because the inclusive
flag doesn’t affect the meaning when rev1 == NULL. */
                before = xstrdup (branchpoint);
                *bp = '\';
            }
        }
    }
    else if (! STREQ (rev1, branchpoint))
    {
        /* Walk deltas from BRANCHPOINT on, looking for REV1. */
        nodep = findnode (rcs->versions, branchpoint);
        revp = (RCSVers *) nodep->data;
        while (revp->next != NULL && ! STREQ (revp->next, rev1))
        {
            revp = (RCSVers *) nodep->data;
            nodep = findnode (rcs->versions, revp->next);
        }
        if (revp->next == NULL)
        {
            error (0, 0, "%s: Revision %s doesn’t exist.", rcs->path, rev1);
            goto delrevalone;
        }
        if (rev1_inclusive)
        {
            before = xstrdup (revp->version);
        }
        else
        {
            before = rev1;
            nodep = findnode (rcs->versions, before);
            rev1 = xstrdup ((RCSVers *)nodep->data)->next;
        }
    }
    else if (rev1_inclusive)
    {
        before = rev1;
        nodep = findnode (rcs->versions, before);
        rev1 = xstrdup ((RCSVers *)nodep->data)->next;
    }
    else if (numdots (branchpoint) > 1)
```
char *bp;
bp = strchr (branchpoint, '.');
while (*bp != NULL)
    *bp = '\0';
bp = '>'; 
/* If any revision between REV1 and REV2 is locked or is a branch point,
we can't delete that revision and must abort. */
next = rev1;
while (!found & next != NULL)
{
    nodep = findnode (rcs->versions, next);
    revp = (RCSVern *) nodep->data;
    if (rev2 != NULL)
        found = STREQ (revp->version, rev2);
    next = revp->next;
    if ( (!found & next != NULL) || rev2.inclusive || rev2 == NULL)
    {
        if (findnode (RCS_setlocks (rcs), revp->version))
            error (0, 0, "%s: can't remove locked revision %s",
                   rcs->path, revp->version);
        goto delrevalone;
    }
    if (revp->branches != NULL)
        error (0, 0, "%s: can't remove branch point %s",
               rcs->path, revp->version);
    goto delrevalone;
}
/* Doing this only for the : syntax is for compatibility.
See cvs.texinfo for somewhat more discussion. */
if (inclusive
    & walklist (RCS_symbols (rcs), findtag, revp->version))
{
    /* We don't print which file this happens to on the theory
that the caller will print the name of the file in a
more useful fashion (fullname not rcs->path). */
    error (0, 0, "cannot remove revision %s because it has tags",
           revp->version);
    goto delrevalone;
}
/* It's misleading to print the 'deleting revision' output
here, since we may not actually delete these revisions.
But that's how RCS does it. Bleah. Someday this should be
moved to the point where the revs are actually marked for
deletion. -twp */
cvs_output ("deleting revision ", 0);
cvs_output (revp->version, 0);
cvs_output ("\n", 1);
}
if (rev2 == NULL)
    else if (found)
    {
        if (rev2.inclusive)
            after = xstrdup (next);
        else
            after = xstrdup (revp->version);
    }
else if (inclusive)
{
    /* In the case of an empty range, for example 1.2:1.2 or
1.2:1.3, we want to just do nothing. */
    status = 0;
    goto delrevalone;
}
else
{
    /* This looks fishy in the cases where tag1 == NULL or tag2 == NULL.
Are those cases really impossible? */
    assert (tag1 != NULL);
    assert (tag2 != NULL);
error (0, 0, "%s: invalid revision range %s-%s", rcs->path, tag1, tag2);
goto delrevpone;
}

if (after == NULL && before == NULL) {
  /* The user is trying to delete all revisions. While an
   RCS file without revisions makes sense to RCS (e.g. the
   state after "rcs-c"), CVS has never been able to cope with
   it. So at least for now we just make this an error.

   We don't include rcs->path in the message since "cvs admin"
   already printed "RCS file:" and the name. */
  error (1, 0, "attempt to delete all revisions");
}

/* The conditionals at this point get really hairy. Here is the
 general idea:

 IF before != NULL and after == NULL
 THEN don't check out any revisions, just delete them
 IF before == NULL and after != NULL
 THEN only check out after's revision, and use it for the new deltatext
 ELSE
   check out both revisions and diff -n them. This could use
   to suppress diagnostic messages and to direct output. */

if (after != NULL) {
  char *diffbuf;
  size_t bufsize, len;

  afterfile = cvstempname();
  status = RCScheckout(rcs, NULL, after, NULL, NULL, afterfile,
                       (RCSCHECKOUTPROC)NULL);
  if (status > 0)
    goto delrevpone;

  if (before == NULL) {
    /* We are deleting revisions from the head of the tree,
       so must create a new head. */
    diffbuf = NULL;
    bufsize = 0;
    get_file(afterfile, afterfile, "r", &diffbuf, &bufsize, &len);

    noexec = noexec;
    noexec = 0;
    if (unlink_file(afterfile) < 0) {
      error (0, errno, "cannot remove %s", afterfile);
      noexec = save_noexec;
      free(afterfile);
      afterfile = NULL;
      free(rcs->head);
      rcs->head = xstrdup(after);
    }
  } else {
    beforefile = cvstempname();
    status = RCScheckout(rcs, NULL, before, NULL, NULL, beforefile,
                       (RCSCHECKOUTPROC)NULL);
    if (status > 0)
      goto delrevpone;

    outfile = cvstempname();
    status = diffexec(beforefile, afterfile, "-n", outfile);
  }

  if (status == 2) {
    /* Not sure we need this message; will diff_exec already
       have printed an error? */
    error (0, 0, "%s: could not diff", rcs->path);
    status = 1;
    goto delrevpone;
  }

  diffbuf = NULL;
  bufsize = 0;
  get_file(outfile, outfile, "r", &diffbuf, &bufsize, &len);
}

/* Save the new change text in after's delta node. */
nodep = findnode(rcs->versions, after);
revp = (RCSVers *)nodep->data;
assert (revp->text == NULL);
revp->text = (Deltatext *) xmalloc(sizeof (Deltatext));
memset ((Deltatext *) revp->text, 0, sizeof (Deltatext));
revp->text->version = xstrdup (revp->version);
revp->text->text = diffbuf;
revp->text->len = len;

/* If DIFFBUF is NULL, it means that OUTFILE is empty and that there are no differences between the two revisions. In that case, we want to force RCS not to write out any
'new change text for this delta,' so we don't want that. */
if (revp->text->text == NULL)
  revp->text->text = xstrdup (""");

/* Walk through the revisions (again) to mark each one as outdated. (FIXME: would it be safe to use the 'dead' field for
this? Doubtful.) */
for (next = rev1;
  next != NULL && (after == NULL || !STREQ (next, after));
  next = revp->next)
  { 
    nodep = findnode (rcs->versions, next);
    revp = rcsVers->nodep->data;
    revp->outdated = 1;
  }

/* Update delta links. If BEFORE == NULL, we're changing the
head of the tree and don't need to update any 'next' links. */
if (before != NULL)
  { /* If REV1 is the first node on its branch, then BEFORE is its
next node (on the trunk) and we have to update its branch list. Otherwise, BEFORE is on the same branch as AFTER, and
we can just change BEFORE's 'next' field to point to AFTER. (This should be safe: since findnode manages its lists via
the 'hashnext' and 'hashprev' fields, rather than 'next' and
'prev', mucking with 'next' and 'prev' should not corrupt the
delta tree's internal structure. Much, -twp.) */
    next = before->next;
    next = next->next;
    beforep = next->prev;
    beforep->next = NULL;
    after = next->next;
    after->prev = NULL;
    beforep->text = after->text;
    free (after->text);
  }
else if (STREQ (rev1, branchpoint))
  { 
    nodep = findnode (rcs->versions, before);
    revp = rcsVers->nodep->data;
    nodep = revp->branches->list->next;
    while (nodep != revp->branches->list &&
      !STREQ (nodep->key, rev1))
      nodep = nodep->next;
    assert (nodep != revp->branches->list);
    if (after == NULL)
      delnode (nodep);
    else
      { 
        free (nodep->key);
        nodep->key = xstrdup (after);
      }
  } else
  { 
    nodep = findnode (rcs->versions, before);
    beforep = rcsVers->nodep->data;
    free (beforep->next);
    beforep->next = xstrdup (after);
  }

status = 0;
delrev_done:
if (rev1 != NULL)
  free (rev1);
if (rev2 != NULL)
  free (rev2);
if (branchpoint != NULL)
  free (branchpoint);
if (before != NULL)
  free (before);
if (after != NULL)
  free (after);
save_noexec = noexec;
noexec = 0;
if (beforefile != NULL)
  {
  }
if (unlink (beforefile) < 0) error (errno, "cannot remove %s", beforefile);
free (beforefile);
}
if (afterfile != NULL) {
if (unlink (afterfile) < 0) error (errno, "cannot remove %s", afterfile);
free (afterfile);
}
if (outfile != NULL) {
if (unlink (outfile) < 0) error (errno, "cannot remove %s", outfile);
free (outfile);
}
noexec = save noexec;
return status;

int RCS_exist_tag (rcs, tag)
RCSNode *rcs;
char *tag;
{
assert (rcs != NULL);
if (findnode (rcs->symbols, tag)) return 1;
return 0;
}

int RCS_exist_rev (rcs, rev)
RCSNode *rcs;
char *rev;
{
assert (rcs != NULL);
if (rcs->flags & PARTIAL)
RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer **) NULL);
if (findnode(rcs->versions, rev) != 0) return 1;
if (walklist (rcs->symbols, findtag, rev) != 0) return 1;
return 0;
}

struct line {
/* Text of this line. Part of the same malloc’d block as the struct line itself (we probably should use the “struct hack” (char text[1]) and save ourselves sizeof (char *) bytes). Does not include \n, instead hasnewline indicates the presence or absence of \n. */
char text;
/* Length of this line, not counting \n if hasnewline is true. */
size_t len;
/* Version in which it was introduced. */
RCSVers *vers;
/* Nonzero if this line ends with \n. This will always be true except possibly for the last line. */
int hasnewline;
/* Number of pointers to this struct line. */
int refcount;
};

struct linevector {
/* How many lines in use for this linevector? */
};
unsigned int nlines;
/* How many lines allocated for this linevector? */
unsigned int linesAllocated;
/* Pointer to array containing a pointer to each line. */
struct linevector *vec;

static void linevector_initPROTO (struct linevector *);  
/* Initialize *VEC to be a linevector with no lines. */
static void

linevector_init (vec)
  struct linevector *vec;
{
  vec->linesAllocated = 0;
  vec->nlines = 0;
  vec->vector = NULL;
}

static int linevector_addPROTO (struct linevector *vec,  
const char *text, size_t len, RCSVers *vers,  
unsigned int pos);  
/* Given some text TEXT, add each of its lines to VEC before line POS  
(where line 0 is the first line). The last line in TEXT may or may  
not be \n terminated. Set the version for each of the new lines to VERS. This  
function returns non-zero for success. It returns zero if the line  
number is out of range.  
Each of the lines in TEXT are copied to space which is managed with  
the linevector (and freed by linevector_free). So the caller doesn't  
need to keep TEXT around after the call to this function. */

linevector_add (vec, text, len, vers, pos)
  struct linevector *vec;
  const char *text;
  unsigned int len, RCSVers *vers;
  unsigned int pos;
{
  const char *textend;
  unsigned int i;
  unsigned int nnew;
  const char *p;
  const char *nextline_text;
  size_t nextline_len;
  int nextline_newline;
  struct linevector *vec;

  if (len == 0)
    return 1;

textend = text + len;
/* Count the number of lines we will need to add. */
nnew = 1;
for (p = text; p < textend; ++p)
  if (*(p++) == \n     && p + 1 < textend)
    ++nnew;

/* Expand VEC->VECTOR if needed. */
if (vec->nlines + nnew > vec->linesAllocated)
{
  if (vec->linesAllocated == 0)
    vec->linesAllocated = 15;
  while (vec->nlines + nnew > vec->linesAllocated)
    vec->vector = xrealloc (vec->vector,  
      vec->linesAllocated + sizeof (*vec->vector));
}

/* Make room for the new lines in VEC->VECTOR. */
for (i = vec->nlines + nnew - 1; i > pos + nnew; --i)
  vec->vector[i] = vec->vector[i - nnew];
if (pos > vec->nlines)
  return 0;
/* Actually add the lines, to VEC->VECTOR. */
i = pos;
nextline_text = text;
nextline_newline = 0;
for (p = text; p < textend; ++p)
  if (*(p++) == \n
    if (p + 1 == textend)
      /* If there are no characters beyond the last newline, we  
      don't consider it another line. */
break;
nextline_len = p - nextline_text;
q = (struct line *) xmalloc(sizeof(struct line) + nextline_len);
q->vers = vers;
q->text = (char *)q + sizeof(struct line);
q->len = nextline_len;
q->has_newline = nextline_newline;
q->refcount = 1;
memcpy(q->text, nextline_text, nextline_len);
vec->vector[i + 1] = q;
vec->nlines += nnew;
return 1;
}

static void linevector_delete PROTO((struct linevector *, unsigned int, unsigned int));
7140
/* Remove N_LINES lines from VEC at position POS (where line 0 is the first line). */
static void linevector_delete(vec, pos, nlines)
struct linevector *vec;
unsigned int pos;
unsigned int nlines;
{
unsigned int i;
unsigned int last;
last = vec->nlines - nlines;
for (i = pos; i < pos + nlines; ++i)
{
if (--vec->vector[i]->refcount == 0)
free(vec->vector[i]);
}
for (i = pos; i < last; ++i)
vec->vector[i] = vec->vector[i + nlines];
vec->nlines -= nlines;
}

static void linevector_copy PROTO((struct linevector *, struct linevector *));
/* Copy FROM to TO, copying the vectors but not the lines pointed to. */
static void linevector_copy(to, from)
struct linevector *to;
struct linevector *from;
7170
{
unsigned int ln;
for (ln = 0; ln < to->nlines; ++ln)
{
if (--to->vector[ln]->refcount == 0)
free(to->vector[ln]);
}
if (from->nlines > to->nlines_allocated)
{
if (to->nlines_allocated == 0)
to->nlines_allocated = 10;
while (from->nlines > to->nlines_allocated)
to->nlines_allocated *= 2;
to->vector = (struct line **)
malloc((to->nlines_allocated + sizeof(*to->vector));
}
memcpy(to->vector, from->vector,
from->nlines * sizeof(*to->vector));
to->nlines = from->nlines;
for (ln = 0; ln < to->nlines; ++ln)
++to->vector[ln]->refcount;
}

static void linevector_free PROTO((struct linevector *));
/* Free storage associated with linevector. */
static void linevector_free(vec);
struct linevector *vec;

unsigned int ln;

if (vec->vector != NULL)
{
    for (ln = 0; ln < vec->nlines; ++ln)
    {
        free (vec->vector[ln]);
    }
    free (vec->vector);
}

static char *month_prototype PROTO ((char *));

/* Given a textual string giving the month (1-12), terminated with any
character not recognised by atoi, return the 3 character name to
print it with. I do not think it is a good idea to change thes e
strings based on the locale; they are standard abbreviations (for
example in rfc822 mail messages) which should be widely understood.

Returns a pointer into static readonly storage. */
static char *
month_prototype (month)
    char *month;
{
    static const char *const months[] =
    int mnum;

    mnum = atoi (month);
    if (mnum < 1 || mnum > 12)
        return "???",
    return (char *)months[mnum - 1];
}

static int
apply_rcs_changes PROTO ((struct linevector *, const char *, size_t,
            const char *, RCSVers *, RCSVers *));

/* Apply changes to the line vector LINES. DIFFBUF is a buffer of
length DIFFLEN holding the change text from an RCS file (the output
of diff -n). NAME is used in error messages. The VERS field of
any line added is set to ADDVERS. The VERS field of any line
deleted is set to DELVERS, unless DELVERS is NULL, in which case
the VERS field of deleted lines is unchanged. The function returns
non-zero if the change text is applied successfully. It returns
zero if the change text does not appear to apply to LINES (e.g., a
line number is invalid). If the change text is improperly
formatted (e.g., it is not the output of diff -n), the function
calls error with a status of 1, causing the program to exit. */
static int
apply_rcs_changes (lines, diffbuf, difflen, name, addvers, delvers)
    struct linevector *lines;
    size_t difflen;
    const char *name;
    RCSVers *addvers;
    RCSVers *delvers;
{
    struct char *p;
    const char *q;
    int op;

    /* The RCS format throws us for a loop in that the deltafrags (if
we define a deltafrag as an add or a delete) need to be applied
in reverse order. So we stick them into a linked list. */
    struct deltafrag {
        enum [ADD, DELETE] type;
        unsigned long pos;
    };
    unsigned long nlines;

    struct char *new_lines;
    size_t len;
    struct deltafrag *next;
    }

    struct deltafrag *dfhead;
    struct deltafrag *df;

dfhead = NULL;
for (p = diffbuf; p != NULL && p < diffbuf + difflen;)
{
    op = *p++;
    if (op != 'a' && op != 'd')
        /* Can't just skip over the deltafrag, because the value
of op determines the syntax. */
        error (1, 0, "unrecognized operation 'X' in 'X'", op, name);
    df = (struct deltafrag *)calloc (sizeof (struct deltafrag));
    df->next = dfhead;
    dfhead = df;
}
if (p == q)
    error (1, 0, "number expected in %s", name);
if (+p++ != 'a')
    error (1, 0, "space expected in %s", name);

if (df pos >= strtoul (p, (char **)&q, 10);
    p = q;
    if (+p++ != 'a')
        error (1, 0, "space expected in %s", name);

if (df nlines = strtoul (p, (char **)&q, 10);
    p = q;
    if (+p++ != 'a')
        error (1, 0, "space expected in %s", name);

if (op == 'a')
{
    unsigned int i;
    df type = ADD;
    i = df nlines;
    /* The text we want is the number of lines specified, or
    until the end of the value, whichever comes first (it
    will be the former except in the case where we are
    adding a line which does not end in newline). */
    for (q = p; i >= 0; ++q)
    if (+q == "\n")
        --i;
    else if (q == diffbuf + difflen)
    {
        if (i != 1)
            error (1, 0, "premature end of change in %s", name);
    
else
    {
        /* Correct for the fact that line numbers in RCS files
        start with 1. */
        --df pos;
        assert (op == 'd');
        df pos -- type = DELETE;
    }

for (df = dfhead; df != NULL;)
{
    unsigned int ln;
    switch (df pos)
    {
        case ADD:
            if (! linevector add (lines, df new lines, df len, addvers, df pos))
                return 0;
            break;
        case DELETE:
            if (df pos > lines nlines || df pos + df nlines > lines nlines)
                return 0;
            if (delvers != NULL)
                for (ln = df pos; ln < df pos + df nlines; ++ln)
                    linevector delete (lines, df pos, df nlines);
            break;
        }
    df = df next;
    free (dfhead);
    dfhead = df;
}
return 1;

/* Apply an RCS change text to a buffer. The function name starts
with rcs rather than RCS because this does not take an RCSNode
argument. NAME is used in error messages. TEXTBUF is the text
buffer to change, and TEXTLEN is the size. DIFFBUF and DIFFLEN are
the change buffer and size. The new buffer is returned in RETBUF
and RETLEN. The new buffer is allocated by malloc.

Return 1 for success. On failure, call error and return 0. */
int
rcs_change_text (name, textbuf, textlen, diffbuf, difflen, retbuf, retlen)
const char *name;
    char *textbuf;
size_t textlen;
const char *diffbuf;
size_t difflen;
char **retbuf;
size_t *retlen;
{
    struct linevector lines;

    int ret;
    *retbuf = NULL;
    *retlen = 0;
    linevector_init (&lines);
    if (! linevector_add (&lines, textbuf, textlen, NULL, 0))
        error (1, 0, “cannot initialize line vector”);

    if (! apply_rcs_changes (&lines, diffbuf, difflen, name, NULL, NULL))
        {
            error (0, 0, “invalid change text in %s”, name);
            ret = 0;
        }
    else
    {
        char *p;
        size_t n;
        unsigned int ln;

        n = 0;
        for (ln = 0; ln < lines.nlines; ++ln)
            /
            /* line */
            /
            n += lines.vector[ln] - len + 1;
        p = xmalloc (n);
        *retbuf = p;
        for (ln = 0; ln < lines.nlines; ++ln)
        {
            memcpy (p, lines.vector[ln] - text, lines.vector[ln] - len);
            p += lines.vector[ln] - len;
            if (lines.vector[ln] - has_newline)
                *p++ = ‘\n’;
        }
        *retlen = p - *retbuf;
        assert (*retlen <= n);
        ret = 1;
    }
    linevector_free (&lines);

    return ret;
}

/* Walk the deltas in RCS to get to revision VERSION.

    If OP is RCS_ANNOTATE, then write annotations using cvs_output.

    If OP is RCS_FETCH, then put the contents of VERSION into a newly-malloc’d array
    and put a pointer to it in *TEXT. Each line is \n terminated; the caller is responsible for converting text files if desired.
    The total length is put in *LEN.

    If FP is non-NULL, it should be a file descriptor open to the file
    RCS with file position pointing to the deltas. We close the file
    when we are done.

    If LOG is non-NULL, then *LOG is set to the log message of VERSION,
    and *LOGLEN is set to the length of the log message.

    On error, give a fatal error. */

static void
RCS_deltas (rcs, fp, rcsbuf, version, op, text, len, log, loglen)
RCSNode *rcs;
FILE *fp;

struct rcsbuffer *rcsbuf;
char *version;
enum rcs_deltas op;
char *text;
size_t *len;
char *log;
size_t *loglen;
{
    struct rcsbuffer rcsbuf_local;
char *branchversion;
char *cversion;
char *key;
sizet vallen;
RCSVers *vers;
RCSVers *prevvers;
RCSVers *trunkvers;
char *next;
int ishead, inext, inversion, onbranch;
Node *node;

struct linevectorheadlines;
struct linevectorcurlines;
struct linevectortrunklines;
int foundhead;

if (fp == NULL) {
    rcsbufcacheopen (rcs, rcs->delta, &fp, &rcsbuflocal);
    rcsbuf = &rcsbuflocal;
}

ishead = 1;
vers = NULL;
prevvers = NULL;
trunkvers = NULL;
next = NULL;
onbranch = 0;
foundhead = 0;

linevectorinit (&curlines);
linevectorinit (&headlines);
linevectorinit (&trunklines);

/* We set BRANCHVERSION to the version we are currently looking for. Initially, this is the version on the trunk from which VERSION branches off. If VERSION is not a branch, then BRANCHVERSION is just VERSION. */
cversion = strchr (branchversion, '.');
if (cversion != NULL) {
cversion = strchr (cversion + 1, '.');
if (cversion != NULL) 
    *cversion = '\0';
do {
    if (!rcsbufgetrevnum (rcsbuf, &key))
        error (1, 0, "unexpected EOF reading RCS file %s", rcs->rpath);
    if (next != NULL && !STREQ (next, key)) {
        /* This is not the next version we need. It is a branch version which we want to ignore. */
        inext = 0;
        inversion = 0;
    } else {
        inext = 1;
        /* look up the revision */
        node = findnode (rcs->versions, key);
        if (node == NULL)
            error (1, 0, "mismatch in rcs file %s between deltas and deltatexts", rcs->rpath);
        /* Stash the previous version. */
        prevvers = vers;
        vers = (RCSVers *)node->data;
        next = vers->next;
        /* Compare key and trunkversion now, because key points to storage controlled by rcsbufgetkey. */
        if (STREQ (branchversion, key))
            inversion = 1;
        else
            inversion = 0;
    }

while (1) {
    if (!rcsbufgetkey (rcsbuf, &key, &value))
        error (1, 0, "%s does not appear to be a valid rcs file", rcs->rpath);

    if (log != NULL && inversion && STREQ (key, "log")

}
& STREQ (branchversion, version))
{
  if (STREQ (key, "text"))
  {
    if (STREQ (branchversion, version))
    {
      linevector_copy (&headlines, &curlines);
      foundhead = 1;
      if (onbranch)
      {
        /* We have found this version by tracking up a branch. Restore back to the lines we saved when we left the trunk, and continue tracking down the trunk. */
        onbranch = 0;
        vers = trunkvers;
        next = vers->next;
        linevector_copy (&curlines, &trunklines);
      }
      else
      {
        Node *p;
        /* We need to look up the branch. */
        onbranch = 1;
        if (numdots (branchversion) < 2)
        {
          unsigned int ln;
          /* We are leaving the trunk; save the current lines so that we can restore them when we continue tracking down the trunk. */
          trunkvers = vers;
          linevector_copy (&curlines, &trunklines);
          /* Reset the version information we have accumulated so far. It only applies to the changes from the head to this version. */
          for (ln = 0; ln < curlines.nlines; ++ln)
            curlines.vector[ln]->vers = NULL;
        }
        /* The next version we want is the entry on VERS>branches which matches this branch. For example, suppose VERSION is 1.21.4.3 and BRANCHVERSION was 1.21. Then we look for an entry starting with "1.21.4" and we'll put it (probably 1.21.4.1) in NEXT. We'll advance BRANCHVERSION by two dots (in this example, to 1.21.4.3). */
        if (vers->branches == NULL)
        {
          error (1, 0, "missing expected branches in %s", rcs->path);
          ++cpversion;
          if (cpversion != NULL)
            error (1, 0, "version number confusion in %s", rcs->path);
        }
        for (p = vers->branches->list->next;
p != vers->branches->list;
p = p->next;
if (strcmp(p->key, branchversion,
    cpversion = branchversion) == 0)
    break;
if (p == vers->branches->list)
    error (1, 0, "missing expected branch in %s",
        rcs->path);
next = p->key;

    cpversion = strchr(cpversion + 1, '.');
    if (cpversion != NULL)
        cpversion = "\0";
}
if (op == RCS_FETCH && &foundhead)
    break;
} while (next != NULL);
free (branchversion);
rcsbuf_cache (rcs, rcsbuf);
if (! foundhead)
    error (1, 0, "could not find desired version %s in %s",
        version, rcs->path);

/* Now print out or return the data we have just computed. */
switch (op)
{
    case RCS_ANNOTATE:
    {
        unsigned int ln;
        for (ln = 0; ln < headlines.nlines; ++ln)
        {
            char buf[80];
            /* Period which separates year from month in date. */
            char *ym;
            /* Period which separates month from day in date. */
            char *md;
            RCSVers *prvers;
            prvers = headlines.vector[ln]->vers;
            if (prvers == NULL)
                prvers = vers;
            sprintf (buf, "%-12s (%-8.8s ",
                prvers->version,
                prvers->author);
            cv_output (buf, 0);
            /* Now output the date. */
            ym = strchr (prvers->date, ' ');
            if (ym == NULL)
                {
                /* ??- is an ANSI trigraph. The ANSI way to avoid it is */
                /* but some pre ANSI compilers complain about the unrecog- */
                /* nized escape sequence. Of course string concatenation */
                /* (???- ???) is also an ANSI-ism. Texting */
                /* STDSEL seems to be a can of worms, since */
                /* compilers do all kinds of things with it. */
                cv_output ("-???", 0);
                cv_output ("-??", 0);
                cv_output ("-??", 0);
            }
            else
            {
                md = strchr (ym + 1, ' ');
                if (md == NULL)
                    cv_output ("??", 0);
                else
                    cv_output (md + 1, 2);
                cv_output ("-\", 1);
                cv_output (monthprintname (ym + 1), 0);
                cv_output ("-\", 1);
                /* Only output the last two digits of the year. Our output */
                /* lines are long enough as it is without printing the */
                /* century. */
                cv_output (ym - 2, 2);
                cv_output ("-\", 1);
            }
            cv_output ("-\", 0);
            if (headlines.vector[ln]->len != 0)
                cv_output (headlines.vector[ln]->text,
                    headlines.vector[ln]->len);
        }
        cv_output ("\n", 1).
    }
break;
case RCS_FETCH:
    {
        char *p;
        size_t n;
        unsigned int ln;
        assert (text != NULL);
        assert (len != NULL);

        n = 0;
        for (ln = 0; ln < headlines.nlines; ++ln)
            for (n+=headlines.vector[ln]->len + 1;
                 p = xmalloc (n);
                 +text = p;
                 for (ln = 0; ln < headlines.nlines; ++ln)
                     { memcp (p, headlines.vector[ln]->text,
                                 headlines.vector[ln]->len);
                       p += headlines.vector[ln]->len;
                       if (headlines.vector[ln]->has_newline)
                           *p++ = 
                       +len = p - +text;
                       assert (*len <= n);
                     }
                     break;
    }
#linevector_free (kcurlines);
#linevector_free (kheadlines);
#linevector_free (ktrunklines);

    return;
}
/* Read the information for a single delta from the RCS buffer RCSBUF,
   whose name is RCSFILE.  *KEYP and *VALP are either NULL, or the
   first key/value pair to read, as set by rcsbuf->key.  Return NULL
   if there are no more deltas.  Store the key/value pair which
   terminated the read in *KEYP and *VALP. */
static RCSVers *
getdelta (rcsbuff, rcfile, keyp, valp)
struct rcsbuffer *rcsbuff;
char *rcfile;
char **keyp;
char **valp;
{
    RCSVers *vnode;
    char **key, **value, **keybuf, **valbuf, *cp;
    Node **kv;

    /* Get revision number if it wasn't passed in. This uses
       rcsbuf->key because it doesn't croak when encountering
       unexpected input. As a result, we have to play unholy games
       with 'key' and 'value'. */
    if (*keyp != NULL)
        {
            key = *keyp;
            value = *valp;
        }
    else
        {
            if (! rcsbuf->key (rcsbuff, &key, &value))
                error (1, 0, "In: unexpected ERE", rcfile);
        }

    /* Make sure that it is a revision number and not a cabbage
     or something. */
    for (cp = key; (isdigit (*cp)) || (*cp == '.'); *cp = 
     "\0", *cp++)
        /* do nothing */;

    /* Note that when comparing with RCSDATE, we are not massaging
     VALUE from the string found in the RCS file. This is OK since
     we know exactly what to expect. */
    if (*cp != 
     "\0" || strncmp (RCSDATE, value, (sizeof RCSDATE) - 1) != 0)
        {
            *keyp = key;
            *valp = value;
            break;
        }
    return NULL;
}

vnode = (RCSVers *) xmalloc (sizeof (RCSVers));
memset (vnode, 0, sizeof (RCSVers));
vnode->version = xstrdup (key);

/* Grab the value of the date from value. Note that we are not
massaging VALUE from the string found in the RCS file. */
cp = value + (sizeof RCSDATE) - 1; /* skip the "date" keyword */
while (whitespace (*cp)) /* take space off front of value */
    cp++;
 vnode->date = xstrdup (cp);
/* Get author field. */
if (! rcsbuf_getkey (rcbuf, &key, &value))
    { error (1, 0, "unexpected end of file reading \"a\", rcsfile);
    }
    if (! STREQ (key, "author"))
        error (1, 0, "); unable to parse \"a\'; 'author' not in the expected place", rcsfile);
    vnode->author = rcsbuf_valcopy (rcbuf, value, 0, (size_t *) NULL);

/* Get state field. */
if (! rcsbuf_getkey (rcbuf, &key, &value))
    { error (1, 0, "unexpected end of file reading \"a\", rcsfile);
    }
    if (! STREQ (key, "state"))
        error (1, 0, "); unable to parse \"a\'; 'state' not in the expected place", rcsfile);
    vnode->state = rcsbuf_valcopy (rcbuf, value, 0, (size_t *) NULL);
/* The value is optional, according to rcsfile(5). */
if (value != NULL && STREQ (value, "dead"))
    { vnode->dead = 1;
    }

/* Note that "branches" and "next" are in fact mandatory, according to doc/RCSFILES. */
/* fill in the branch list (if any branches exist) */
if (! rcsbuf_getkey (rcbuf, &key, &value))
    { error (1, 0, "unexpected end of file reading \"a\", rcsfile);
    }
    if (STREQ (key, RCSDESC))
    {
        /*keyp = key;
         *valp = value;
          */
        /* Probably could/should be a fatal error. */
        error (0, 0, "warning: 'branches' keyword missing from \"a\", rcsfile);
        return vnode;
    }
    if (value != (char *) NULL)
        { vnode->branches = getlist ();
    }
/* Note that we are not massaging VALUE from the string found in the RCS file. */
/* do_branches (vnode->branches, value);
 */
/* fill in the next field if there is a next revision */
if (! rcsbuf_getkey (rcbuf, &key, &value))
    { error (1, 0, "unexpected end of file reading \"a\", rcsfile);
    }
    if (STREQ (key, RCSDESC))
    {
        /*keyp = key;
         *valp = value;
          */
        /* Probably could/should be a fatal error. */
        error (0, 0, "warning: 'next' keyword missing from \"a\", rcsfile);
        return vnode;
    }
    if (value != (char *) NULL)
        { vnode->next = rcsbuf_valcopy (rcbuf, value, 0, (size_t *) NULL);
    }
/* read the "remote branches" list (optional) */
if (! rcsbuf_getkey (rcbuf, &key, &value))
    { error (1, 0, "unexpected end of file reading \"a\", rcsfile);
    }
    if (STREQ (key, "remote-branches"))
    {
        vnode->remote_branches = getlist ();
    }
/* Note that we are not massaging VALUE from the string found in the RCS file. */
/* do_remote_branches (vnode->remote_branches, value);
 */
/* XXX - this is where we put the symbolic link stuff???
 * (into newphrases in the deltas).
 /*
while (1)
{
    int len;
    size_t valbuflen;
    key = NULL;
    if (!rcsbuf_getid (rcsbuf, &keybuf))
        error (1, 0, "unexpected end of file reading %s", rcsfile);
    /* rcsbuf_getid did not terminate the key, so copy it to new space. */
    len = rcsbuf->ptr - keybuf;
    key = (char *) xmalloc (sizeof (char) * (len + 1));
    strcpy (key, keybuf, len);
    key[len] = '\0';
    /* The 'desc' keyword has only a single string value, with no trailing semicolon, so it must be handled specially. */
    if (STREQ (key, RCSDESC))
    {
        (void) rcsbuf_getstring (rcsbuf, &valbuf);
        value = rcsbuf_valcopy (rcsbuf, valbuf, 1, &valbuflen);
        break;
    }
    ifndef PRESERVE_PERMISSIONS_SUPPORT
    /* The 'hardlinks' value is a group of words, which must be parsed separately and added as a list to vnode->hardlinks. */
    if (STREQ (key, "hardlinks"))
    {
        vnode->hardlinks = getlist();
        while (1)
        {
            if (!rcsbuf_getword (rcsbuf, &valbuf))
                error (1, 0, "unexpected end of file reading %s", rcsfile);
            if (valbuf == NULL)
                break;
            n = getnode();
            n->key = rcsbuf_valcopy (rcsbuf, valbuf, 1, NULL);
            addnode (vnode->hardlinks, n);
            continue;
        }
    }endif
    /* Get the value. */
    value = NULL;
    while (1)
    {
        if (!rcsbuf_getword (rcsbuf, &valbuf))
            error (1, 0, "unexpected end of file reading %s", rcsfile);
        if (valbuf == NULL)
            break;
        /* Copy valbuf to new space so we can polish it, then append it to value. */
        if (value == NULL)
            { value = rcsbuf_valcopy (rcsbuf, valbuf, 1, &valbuflen); }
        else
            { char *temp_value;
                temp_value = rcsbuf_valcopy (rcsbuf, valbuf, 1, &valbuflen);
                len = strlen (value);
                value = (char *) xrealloc (value, sizeof (char) * (len + valbuflen + 2));
                value[len] = '\0';
                strcpy (value + len + 1, temp_value);
                free (temp_value);
            }
        /* Enable use of repositories created by certain obsolete versions of CVS. This code should remain indefinitely; there is no procedure for converting old repositories, and checking for it is harmless. */
    }
    if (STREQ (key, RCSDEAD))
    {
        vnode->dead = 1;
        if (vnode->state != NULL)
            free (vnode->state);
        vnode->state = xstrdup ("dead");
        continue;
    }
    /* if we have a new revision number, we're done with this delta */
for (cp = key; (isdigit (*cp)) || (*cp == ','); & & *cp != '\0'; cp++)
    /* do nothing */;
/* Note that when comparing with RCSDATE, we are not massaging
VALUE from the string found in the RCS file. This is OK
since we know exactly what to expect. */
if (*cp == '\0') & & strcmp (RCSDATE, value, strlen (RCSDATE)) == 0
    break;
/* At this point, key and value represent a user-defined field
on the delta node. */
if (vnode->otherdelta == NULL)
    vnode->otherdelta = getlist ();
kv = getnode ();
kv->type = RCSFIELD;
kv->key = key;
kv->data = value;
if (addnode (vnode->otherdelta, kv) != 0)
    {
        /* Complaining about duplicate keys in newphrases
        seems questionable, in that we don't know what they mean and
doc/RCSFILES has no prohibition on several newphrases
        with the same key. But we can't store more than one as
        long as we store them in a List */
        /* error (0, 0, "warning: duplicate key 'Xa' in RCS file 'Ya'",
            key, rcfile);
            free node (kv);
        }
    }
/* Return the key which caused us to fail back to the caller. */
*keyp = key;
*valp = value;
return vnode;
}

static void
freevatext (d)
Deltatext *d;
{
    if (d->version != NULL)
        free (d->version);
    if (d->log != NULL)
        free (d->log);
    if (d->text != NULL)
        free (d->text);
    if (d->other != (List *) NULL)
        delist (&d->other);
    free (d);
}

static Deltatext *
RCS_getdeltatext (rcs, fp, rcsbuf)
RCSNode *rcs,
FILE *fp,
struct rcsbuffer *rcsbuf;
    
char *num;
char *key,*value;
Node *p;
Deltatext *d;

/* Get the revision number. */
if (! rcsbuf->strevnum (rcsbuf, &num))
    {
        /* If num == NULL, it means we reached EOF naturally. That's
        fine. */
        if (num == NULL)
            return NULL;
    else
        error (1, 0, "XA: unexpected EOF", rcs->path);

    p = findnode (rcs->versions, num);
    if (p == NULL)
        error (1, 0, "mismatch in rcs file Xa between deltas and deltatexts",
            rcs->path);
    d = (Deltatext *) xmalloc (sizeof (Deltatext));
    d->version = xstrdup (num);

    /* Get the log message. */
    if (! rcsbuf->getkey (rcsbuf, &key, &value))
        error (1, 0, "XA, delta Xa: unexpected EOF", rcs->path, num);
    if (! STREQ (key, "log"))
        error (1, 0, "XA, delta Xa: expected 'log', got '"Xa'",
            rcs->path, num, key);
    d->log = rcsbuf->valucopy (rcsbuf, value, 0, (size_t) NULL);
    /* Get random newphrases. */
d->other = getlist();

while (1)
{
    if (! rebuf_getkey (rcbuf, &key, &value))
        error (1, 0, "%s, delta %s: unexpected EOF", rcs->path, num);
    if (strcmp (key, "text"))
        break;

    p = getnode();
    p->type = RCSFIELD;
    p->key = xstrdup (key);
    p->data = rebufvalcopy (rcbuf, value, 1, (size_t *) NULL);
    if (addnode (d->other, p) < 0)
        { error (0, 0, "warning: %s, delta %s: duplicate field 'text',
                  rcs->path, num, key);}
}

/* Get the change text. We already know that this key is 'text'. */

8120  d->text = rebufvalcopy (rcbuf, value, 0, &d->len);

return d;

} /* RCS output functions, for writing RCS format files from RCSNode structures.

For most of this work, RCS 5.7 uses an 'aprintf' function which aborts
program upon error. Instead, these functions check the output status
of the stream right before closing it, and aborts if an error condition
is found. The RCS solution is probably the better one: it produces
more overhead, but will produce a clearer diagnostic in the case of
catastrophic error. In either case, however, the repository will probably
not get corrupted. */

8130  static int
putsymbol_proc (symnode, fparg)
    Node *symnode;
    void *fparg;
8140  {
    FILE *fp = (FILE *) fparg;
    /* A fiddly optimization: this code used to just call fprintf, but
       in an old repository with hundreds of tags this can get called
       hundreds of thousands of times when doing a cvstag. Since
       tagging is a relatively common operation, and using putc and
       fputs is just as comprehensible, the change is worthwhile. */
    fputc ("\n", fp);
    fputc ("\t", fp);
    fputs (symnode->key, fp);
    fputc ("\t", fp);
    fputs (symnode->data, fp);
    return 0;
}

8150  static int
putlock_proc PROTO ((Node *, void *));
    /* putlock_proc is like putsymbol_proc, but key and data are reversed. */

8160  static int
putlock_proc (symnode, fp)
    Node *symnode;
    void *fp;
8170  {
    return fprintf ((FILE *) fp, "\n\t\t%a", symnode->data, symnode->key);
}

8180  static int
putcfield_proc (node, vfp)
78170  Node *node;
    void *vfp;
8190  {
    FILE *fp = (FILE *) vfp;
    /* Some magic keys used internally by CVS start with 'i'. Skip them. */
    if (node->key[0] == 'i')
        return 0;
    fprintf (fp, "\n\n\t\t%a", node->key);
8200  if (node->data != NULL)
    {
        /* If the field's value contains evil characters,
           it must be stringified. */
        /* PIXME: This does not quite get it right. "\xf8f" is not a legal
           value for a value in a newphrase, according to doc/RCSSFILES,
           because digits are not valid in an "id". We might do OK by
           always writing strings (enclosed in @). Would be nice to
           explicitly mention this one way or another in doc/RCSSFILES.
           */
A case where we are wrong in a much more clear-cut way is that
we let through non-graphics characters such as whitespace and
control characters. */

```c
int n = strlenp (node->data, "\$\ldots\$" );
if (node->data [n] == $)
   fputs (node->data, fp);
else
   {
      putc ('\', fp);
      expand_symbols (node->data, (offs) strlen (node->data), fp);
      putc ('\$', fp);
   }
/* desc, log and text fields should not be terminated with semicolon;
   all other fields should be. */
if (! STREQ (node->key, "desc") &&
    ! STREQ (node->key, "log") &&
    ! STREQ (node->key, "text" ))
   {
      putc ('!', fp);
   }
return 0;
}
#endif
static int
puthardlink (node, vfp)
Node *node;
void *vfp;
{
   FILE *fp = (FILE *) vfp;
   char *basename = strchr (node->key, '/ ');
   if (basename == NULL)
      basename = node->key;
   else
      ++basename;
   putc ('\n', fp);
   putc ('\%', fp);
   (void) expand_symbols (basename, strlen (basename), fp);
   putc ('\$', fp);
   return 0;
}
#endif
/* Output the admin node for RCS into stream FP. */
static void
RCSSymlinks (rcs, fp)
RCSNode *rcs;
FILE *fp;
{
   fprintf (fp, "\n\n" , RCSHEAD, rcs->head ? rcs->head ; "\n" );
   if (rcs->branch)
      fprintf (fp, "\n\n" , RCSHEAD, rcs->branch);
   fputs ("access", fp);
   if (rcs->access)
      {
      char *s = strchr (rcs->access);
      for (p = strtok (s, "\n\t\n" ); p != NULL; p = strtok (NULL, "\n\t\n" ))
         fprintf (fp, "\n\n" , s); 
         free (s);
   }
   fputs ("\n\n", fp);
   fputs (RCSSYMBOLS, fp);
   /* If we haven't had to convert the symbols to a list yet, don't
      force a conversion now: just write out the string. */
   if (rcs->symbols == NULL && rcs->symbols->head != NULL)
      {
         fputs ("\n\n", fp);
         fputs (rcs->symbols->data, fp);
      } else
      walklist (RCSSymbols (rcs), putsymbolproc, (void *) fp);
   fputs ("\n\n", fp);
   fputs ("locks", fp);
```
if (rcs->lockproc)  
    fprintf(fp, "\%s", rcs->lockproc);  
else if (rcs->lock)  
    walklist(rcs->locks, putlockproc, (void *)fp);  
if (rcs->strict)  
    fprintf(fp, "%s", "strict");  
puts(";\n", fp);  
if (rcs->comment)  
    fprintf(fp, "comment\t0");  
expandsigns(rcs->comment, (offs)) strlen(rcs->comment), fp);  
puts(";\n", fp);  
}  
if (rcs->expand & !STREQ(rcs->expand, "**"))  
    fprintf(fp, "@\n\n", RCSEXPRAND, rcs->expand);  
walklist(rcs->other, putrcsfieldproc, (void *)fp);  
putc(";\n", fp);  
}  
static void  
putdelta(vers, fp)  
RCSTree vers;  
FILE *fp;  
{  
    Node *bp, *start;  
    /* Skip if no revision was supplied, or if it is outdated (cvadmin -o) */  
    if (vers == NULL || vers->outdated)  
        return;  
    fprintf(fp, "\%s\n\n", vers->version,  
            vers->version,  
            RCSDATE, vers->date,  
            "author", vers->author,  
            "state", vers->state ? vers->state : "");  
    if (vers->branches != NULL)  
    {  
        start = vers->branches->list;  
        for (bp = start->next; bp != start; bp = bp->next)  
            fprintf(fp, "\%s\n\n", bp->key);  
    }  
    fprintf(fp, "\n\nnew \n\n");  
    if (vers->remote_branches != NULL)  
    {  
        start = vers->remote_branches->list;  
        for (bp = start->next; bp != start; bp = bp->next)  
            fprintf(fp, "\%s\n\n", bp->key);  
    }  
    fprintf(fp, ";\n");  
    walklist(vers->other, putrcsfieldproc, fp);  
#endif  
#define PRESERVE_PERMISSIONS_SUPPORT  
static void  
putrsetree(rcs, rev, fp)  
RCSTree rcs;  
char rev;  
FILE *fp;  
{  
    RCSVers +vers;  
    Node *p, *branch;  
    if (rev == NULL)  
        return;  
    /* Find the delta node for this revision. */  
    p = findnode(rcs->versions, rev);  
    assert(p != NULL);  
    versp = (RCSVers *)p->data;  
    /* Print the delta node and recurse on its 'next' node. This prints  
the trunk. If there are any branches printed on this revision,
print those trunks as well. */

8370  putdelta (versp, fp);
RCS_outputtree (rcs, versp->next, fp);
if (versp->branches != NULL)
  {
    branch = versp->branches->list;
    for (p = branch->next; p != branch; p = p->next)
      RCS_outputtree (rcs, p->key, fp);
  }
}

8380 static void
RCS_outputdesc (rcs, fp)
RCSNode *rcs, FILE *fp;
{
  if (rcs->desc != NULL)
    {
      off_t len = (off_t) strlen (rcs->desc);
      if (len > 0)
        {
          expand_at_signs (rcs->desc, len, fp);
          if (rcs->desc[len-1] != '\n')
            putc ('\n', fp);
        }
    }
  puts ("\n", fp);
}

static void
8400 putdeltatext (fp, d)
FILE *fp;
Deltatext *d;
{
  if (d->version != NULL)
    {
      int loglen = strlen (d->log);
      expand_at_signs (d->log, (off_t) loglen, fp);
      if (d->log[loglen-1] != '\n')
        putc ('\n', fp);
    }
  puts ("\n", fp);
  walklist (d->other, putrcsfield Proc, fp);
  puts ("\n\n", fp);
  if (d->text != NULL)
    expand_at_signs (d->text, (off_t) d->len, fp);
  puts ("\n", fp);
}

8420 */
/* TODO: the whole mechanism for updating deltas is kludgey... more
 sensible would be to supply all the necessary info in a `newdeltatext'
 field for RCSVers nodes. -tap */

/* Copy delta text nodes from PIN to FOUT. IF NEWDTEXT is non-NULL, it
 is a new delta text node, and should be added to the tree at the
 node whose revision number is INSERTPT. (Note that trunk nodes are
 written in decreasing order, and branch nodes are written in
 increasing order.) */

8450 static void
8440 copydeltas (rcs, fin, rcbufin, fout, newdtext, insertpt)
RCSNode *rcs;
FILE *fin;
struct rcbuf *rcbufin;
FILE *fout;
Deltatext *newdtext;
char *insertpt;
{
  int actions;
  RCSVers *radmin;
  Node *np;
  int insertbefore, found;
  char *buftext, fp;
  int nls;
  size_t buflen;
  char buf[RBUFSIZ];
  int got;

  8450 /* Count the number of versions for which we have to do some
 special operation. */
  actions = walklist (rcs->versions, count_deltas, actions, (void *) NULL);

  /* Make a note of whether NEWDTEXT should be inserted
 before or after its INSERTPT. */
  insertbefore = (newdtext != NULL && numdots (newdtext->version) == 1);
while (actions != 0 || newdtext != NULL)
{
    Deltatext = dtxt;
    dtxt = RCS_getdeltatext (rcs, fin, rcsbufin);
    /* We shouldn’t hit EOF here, because that would imply that
     * some action was not taken, or that we could not insert
     * NEWDTEXT. */
    if (deltatext == NULL)
        error (l, 0, "internal error: EOF too early in RCS_copydeltas");

    found = (insertpt == NULL && STREQ (deltatext->version, insertpt));
    if (found && insertbefore)
    {
        putdeltatext (fout, newdtext);
        newdtext = NULL;
        insertpt = NULL;
    }

    np = findnode (rcs->versions, dtxt->version);
    if (found && !insertbefore)
    {
        putdeltatext (fout, newdtext);
        newdtext = NULL;
        insertpt = NULL;
    }

    if (found && insertpt != NULL)
    {
        freespace = FALSE;
        freespace = FALSE;
    }

    if (found && !insertbefore)
    {
        putdeltatext (fout, newdtext);
        newdtext = NULL;
        insertpt = NULL;
    }

    /* Copy the rest of the file directly, without bothering to
     * interpret it. The caller will handle error checking by calling
     * ferror. We just wrote a newline to the file, either in putdeltatext
     * or in the caller. However, we may not have read the corresponding
     * newline from the file, because rcsbuf_getkey returns as soon as
     * it finds the end of the ‘ ‘ string for the desc or text key.
     * Therefore, we may read three newlines when we should really
     * only write two, and we check for that case here. This is not
     * an semantically important issue; we only do it to make our RCS
     * files look traditional. */
    nls = 3;
    rcsbuf_get_buffered (rcsbufin, &bufrest, &buflen);
    if (buflen > 0)
    {
        if (bufrest[0] == '"'
            || strncmp (bufrest, \"\n\n\n\n", buflen < 3 ? buflen : 3) != 0)
        {
            nls = 0;
        }
        else
            if (buflen < 3)
            {
                nls = buflen;
            }
            else
            {
while ((got = fread (buf, 1, sizeof buf, fin)) != 0)
{
    if (nls > 0
        && got >= nls
        && buf[0] == 'n'
        && strncmp (buf, "\n\n\n", nls) == 0)
    {
        fwrite (buf + 1, 1, got - 1, fout);
    }
    else
    {
        fwrite (buf, 1, got, fout);
    }
}
/* A helper procedure for RCS_copydeltas. This is called via walklist
to count the number of RCS revisions for which some special action
is required. */

static int
count_delta_actions (np, ignore)
Node *np;
void *ignore;
{
    RCSVers *dadmin;
    dadmin = (RCSVers *) np->data;
    if (dadmin->outdated)
        return 1;
    if (dadmin->text != NULL
        && (dadmin->text->log != NULL || dadmin->text->text != NULL))
    {
        return 1;
    }
    return 0;
}

/* RCS_internal_lockfile and RCS_internal_unlockfile perform RCS-style
locking on the specified RCSFILE: for a file called 'foo,v'
open for writing a file called ',foo,'. We probably should
clen the ',foo,' file on ^C. We probably should
write the ',foo,' file anyway (unless perhaps if _EXCL is slower or
something).

A concern has been expressed about whether adopting the RCS
strategy would slow us down. I don’t think so, since we need to
write the ',foo,' file anyway (unless perhaps if _EXCL is slower or
something).

These do not perform quite the same function as the RCS -l option
for locking files: they are intended to prevent competing RCS
processes from stomping all over each other’s laundry. Hence,
they are ‘internal’ locking functions.

Note that we don’t clean up the ',foo,' file on ^C. We probably should.
I’m not completely sure whether RCS does or not (I looked at the code
a little, and didn’t find it).

If there is an error, give a fatal error; if we return we always

*/

static FILE *
rcs_internal_lockfile (rcsfile)
char *rcsfile;
{ char *lockfile;
  struct stat rstat;
FILE *fp;

/* Get the lock file name: 'file,' for RCS file 'file,v'. */
lockfile = rcs_lockfilename(rcsfile);

/* Use the existing RCS file mode, or read-only if this is a new
file. (Really, this is a lie -- if this is a new file,
RCScheckin uses the permissions from the working copy. For
actually creating the file, we use 0444 as a safe default mode.) */
if (stat (rcsfile, &rstat) < 0)
{
  if (stat_error (errno))
    rstat.st_mode = S_IRUSR | S_IWGRP | S_IROTH;
  else
    error (1, errno, "cannot stat file", rcsfile);
}

/* Try to open exclusively. POSIX.1 guarantees that O_EXCL|O_CREAT
 guarantees an exclusive open. According to the RCS source, with
NFS v2 we must also throw in O_TRUNC and use an open mask that makes
the file unwritable. For extensive justification, see the comments for
rcsetwopen() in rcsedit.c, in RCS 5.7. This is kind of pointless
in the CVS case; see comment at the start of this file concerning
general foo file strategy.

There is some sentiment that with NFSv3 and such, that one can
rely on O_EXCL these days. This might be true for unix (I
don't really know), but I am still pretty skeptical in the case
of the non-unix systems. */
ifdef HAVE_FCHMOD
fd = open (lockfile, OPEN_BINARY | O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, S_IRUSR | S_IWGRP | S_IROTH);
endif

if (fd < 0)
{
  error (1, errno, "could not open lock file", lockfile);
}

/* Force the file permissions, and return a stream object. */
/* Because we change the modes later, we don't worry about
this in the non-HAVE_FCHMOD case. */
ifdef HAVE_FCHMOD
if (fchmod (fd, rstat.st_mode) < 0)
  error (1, errno, "cannot change mode", lockfile);
endif

fp = fdopen (fd, OPEN_BINARY_WRITE);
if (fp == NULL)
  error (1, errno, "cannot fdopen", lockfile);

free (lockfile);

return fp;

}

static void
rcs_internal_unlockfile (fp, rcsfile)
FILE *fp;
char *rcsfile;
{
  char *lockfile;

  /* Get the lock file name: 'file,' for RCS file file,v. */
  lockfile = rcs_lockfilename (rcsfile);

  /* Abort if we could not write everything successfully to LOCKFILE.
     This is not a great error-handling mechanism, but should prevent
corrupting the repository. */
  if (ferror (fp))
    /* The only case in which using errno here would be meaningful
       is if we happen to have left errno unmolested since the call
       which produced the error (e.g. fprintf). That is pretty
       fragile even if it happens to sometimes be true. The real
       solution is to check each call to fprintf rather than waiting
       until the end like this. */
    error (1, 0, "error writing to lock file", lockfile);
  if (fclose (fp) == EOF)
    error (1, errno, "error closing lock file", lockfile);
  rename_file (lockfile, rcsfile);
  free (lockfile);

}

static char *
rcs_lockfilename (rcsfile)
char *rcsfile;
{
  char *lockfile, *lockp;
  char *rcsbase, *rcap, *rresend;
  int rcrlen;

  if (rcsfile[0] == '

/* Create the lockfile name. */
rcalen = strlen(rcfile);
lockfile = (char *) xmalloc(rcalen + 10);
rcbase = lastcomponent(rcfile);
rcsend = rcfile + rcalen - sizeof(RCS_EXT);
for (lockp = lockfile, rcp = rcfile; rcp < rcbase; ++rcp)
  *lockp++ = *rcp;
*lockp++ = '\0';
while (rcp <= rcsend)
  *lockp++ = rcp++;
*lockp++ = '\0';
*lockp = '\0';

return lockfile;

/* Rewrite an RCS file. The basic idea here is that the caller should
first call RCS_rename_rcsfile, then munge the data structures as
desired (via RCS_delete_rcs, RCS_setting, &c), then call RCS_rewrite. */

void
RCS_rewrite (rcs, newdtext, insertpt)
RCSNode *rcs;
Deltatext *newdtext;
char *insertpt;

{ FILE *fin, *fout;
struct rcsbuffer *rcsbuffin;
if (noexec)
  return;

fout = rcs_internal_lockfile (rcs->path);
RCS_readmin (rcs, fout);
RCS_reddtree (rcs, rcs->head, fout);
RCS_redesc (rcs, fout);

/* Open the original RCS file and seek to the first delta text. */
rebuf_cache_open (rcs, rcs->delta_pos, &fin, &rcsbuffin);

/* Update delta_pos to the current position in the output file.
Do NOT move these statements: they must be done after fin has
been positioned at the old delta_pos, but before any delta
texts have been written to fout. */
rcs->delta_pos = ftell (fout);
if (rcs->delta_pos == -1)
  error (1, errno, "cannot ftell in RCS file %s", rcs->path);
RCS_copydeltas (rcs, fin, &rcsbuffin, fout, newdtext, insertpt);

/* We don't want to call rebuf_cache here, since we're about to
delete the file. */
rebuf_close (&rcsbuffin);
if (errno (fin))
  /* The only case in which using errno here would be meaningful
   is if we happen to have left errno unmolested since the call
   which produced the error (e.g. fread). That is pretty
   fragile even if it happens to sometimes be true. The real
   solution is to make sure that all the code which reads
   from fin checks for errors itself (some does, some doesn't). */
error (1, "warning: closing RCS file \"", rcs->path);
if (fclose (fin) < 0)
  error (1, errno, "warning: closing RCS file \"", rcs->path);
rcs_internal_unlockfile (fout, rcs->path);
}

/* Annotate command. In rcs.c for historical reasons (from back when
what is now RCS_deltas was port of annotate_fileproc). */

/* Options from the command line. */

static int force_tag_match = 1;
static char *tag = NULL;
static char *date = NULL;
static int annotate_fileproc PROTO ((void *callerdat, struct file_info *));

static int
annotate_fileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;

{ FILE *fp = NULL;
struct rcsbuffer *rebufp = NULL;
struct rcsbuffer rebuf;
char *version;
if (finfo->rcs == NULL) return (1);

    if (finfo->rcs->flags & PARTIAL)
    {
        RCS_reparsercsfile (finfo->rcs, &fp, &rcsbuf);
        rcsbufp = &rcsbuf;
    }

    version = RCS Anatversion (finfo->rcs, tag, date, force_tag_match, (int *) NULL);

if (version == NULL) return (0);

    /* Distinguish output for various files if we are processing several files. */
    cvs_outerr ("Annotate for ", 0);
    cvs_outerr (finfo->fullname, 0);
    cvs_outerr ("***************
", 0);
    RCS_deltas (finfo->rcs, fp, rcsbufp, version, RCS_ANNOTATE, (char **) NULL,

    (size_t) NULL, (char **) NULL, (size_t *) NULL);
    free (version);
    return (0);
}

static const char *const annotate_usage[] =
{ /* Usage: %s %s [\-lRf] [\-r rev|-D date] [files...]
   * '\-l\' Local directory only, no recursion.\n   * '\-R\' Process directories recursively.\n   * '\-t\-1' Process directories recursively.\n   * '\-t\-r\' Process directories recursively.\n   * '\-t\-f' Force revision if tag/date not found.\n   * '\-D\' Specify the \--help global option for a list of other help options.\n   */
    "Usage: %s %s [\-lRf] [\-r rev|-D date] [files...]
   \-l\' Local directory only, no recursion.\n   \-R Process directories recursively.\n   \-f Force revision if tag/date not found.\n   \-D Specify the \--help global option for a list of other help options.\n   \n   \n   */
};

int annotate (argc, argv)
{
    int local = 0;
    int c;

    if (argc == -1) usage (annotate_usage);

   (optarg = 0;
    while ((c = getopt (argc, argv, "\-lRf:dr\")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case 'r':
                tag = optarg;
                break;
            case 'D':
                date = MakeDate (optarg);
                break;
            case 'f':
                force_tag_match = 0;
                break;
            case '?':
                default:
                    usage (annotate_usage);
                    break;
        }
    }

    argv = optarg;
    argv++ = optarg;

    #ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();
        ign_setup ();
        if (local)
            sendarg ("\-l");
        if (force_tag_match)
sendarg ("-t");

option_with_arg ("+t", tag);

if (date)
    client_senddate (date);

send_file_names (arg, argv, 0, SEND_EXPAND_WILD);

send_files (arg, argv, local, 0, SEND_NO_CONTENTS);

send_to_server (*annotate@129*, 0);

return get_responses_send_close ();

}  

sendif /* CLIENT_SUPPORT */

if (tag != NULL)
    tag_check_valid (tag, argv, local, 0, "*");

return start_recursion (annotate_fileproc, (FILESDONEPROC) NULL,
                        (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                        argv, local, NULL, 0, 1, (char *)NULL, 1);

}

/*  
For a given file with full pathname PATH and revision number REV,
produce a file label suitable for passing to diff.  The default
file label as used by RCS 5.7 looks like this:
FILENAME <tab> YYYY/MM/DD <sp> HH:MM:SS <tab> REVNUM
The date and time used are the revision's last checkin date and time.
If REV is NULL, use the working copy's utime instead.
*/
char *

make_file_label (path, rev, rcs)

char *path;
char *rev;
RCSNode *rcs;

{  
    char datelabel[MAXDATELEN];

    char *label;

    char *file;

    label = (char *) xmalloc (strlen (file)
        + (rev == NULL ? 0 : strlen (rev))
        + 96);

    if (rev)
    {
        char *date;
        RCS_getmtime (rcs, rev, datebuf, 0);
        date = printable_date (datebuf);
        (void) sprintf (label, "-%s\t%*s\t%*s", file, date, rev);
    
    free (date);
    }

else
    {
        struct stat sb;
        struct tm *wm;

        if (CVS_SSTAT (file, &sb) < 0)
            error (0, 1, "could not get info for '%s'", path);

        else
        {
            wm = gmtime (&sb.st_mtime);
            (void) sprintf (datebuf, "%04d/%02d/%02d %02d:%02d:%02d",
                            wm->tm_year + 1900, wm->tm_mon + 1,  
                            wm->tm_mday, wm->tm_hour,  
                            wm->tm_min, wm->tm_sec);

            (void) sprintf (label, "-%s\t%*s", file, datebuf);
        }

    return label;

}

else
    {
        label = (char *) xmalloc (strlen (file)
            + (rev == NULL ? 0 : strlen (rev))
            + 96);

        if (rev)
        {
            char *date;
            RCS_getmtime (rcs, rev, datebuf, 0);
            date = printable_date (datebuf);
            (void) sprintf (label, "-%s\t%*s\t%*s", file, date, rev);
        
        free (date);
        }

else
    {
        struct stat sb;
        struct tm *wm;

        if (CVS_SSTAT (file, &sb) < 0)
            error (0, 1, "could not get info for '%s'", path);

        else
        {
            wm = gmtime (&sb.st_mtime);
            (void) sprintf (datebuf, "%04d/%02d/%02d %02d:%02d:%02d",
                            wm->tm_year + 1900, wm->tm_mon + 1,  
                            wm->tm_mday, wm->tm_hour,  
                            wm->tm_min, wm->tm_sec);

            (void) sprintf (label, "-%s\t%*s", file, datebuf);
        }

    return label;

}
A.45  rcs.h

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * RCS source control definitions needed by rcs.c and friends */

/* String which indicates a conflict if it occurs at the start of a line. */

#define RCS_MERGE_PAT >>>>>>
#define RCS_EXT "v"
#define RCS_PAT "*,v"
#define RCS_BRANCH "branch"
#define RCS_SYMBOLS "symbols"
#define RCSDATE "date"
#define RCSDESC "desc"
#define RCS_EXPAND "expand"
#define RCS_REMOTE_BRANCH "1.1.3"

/* Used by the version of death support which resulted from old
versions of CVS (e.g. 1.5 if you define DEATH_SUPPORT and not
DEATH_STATE). Only a hacked up RCS (used by those old versions of
CVS) will put this into RCS files. Considered obsolete. */
#define RCS_DEAD "dead"

#define DATEFORM "%02d.%02d.%02d.%02d.%02d.%02d"
#define SDATEFORM "%d.%d.%d.%d.%d.%d"

/* Opaque structure definitions used by RCS specific lookup routines */

#define VALID 0x1 /* flags field contains valid data */
#define INATTIC 0x2 /* RCS file is located in the Attic */
#define PARTIAL 0x4 /* RCS file not completely parsed */

/* All the "char *" fields in RCSNode, Deltatext, and RCSVers are
\0-terminated (except "text" in Deltatext). This means that we
can't deal with fields containing \0', which is a limitation that
RCS does not have. Would be nice to fix this someday. */

struct rcsnode {
    /* Reference count for this structure. Used to deal with the
fact that there might be a pointer from the VersTS or might
not. Callers who increment this field are responsible for
calling freercsnode when they are done with their reference. */
    int refcount;
    /* Flags (INATTIC, PARTIAL, &c), see above. */
    int flags;
    /* File name of the RCS file. This is not necessarily the name
as specified by the user, but it is a name which can be passed to
system calls and a name which is OK to print in error messages
(the various names might differ in case). */
    char *path;
    /* Value for head keyword from RCS header, or NULL if empty. */
    char *head;
    /* Value for branch keyword from RCS header, or NULL if omitted. */
    char *branch;
    /* Raw data on symbolic revisions. The first time that RCS_symbols is
called, we parse these into SYMBOLS, and free SYMBOLS_data. */
    char *symbols_data;
    /* Value for expand keyword from RCS header, or NULL if omitted. */
    char *expand;
    /* List of nodes, the key of which is the symbolic name and the data
of which is the numeric revision that it corresponds to (malloc'd). */
    List *symbols;
    /* List of nodes (type RCSVERS), the key of which the numeric revision
number, and the data of which is an RCSVers for the revision. */
    List *versions;
    /* Value for access keyword from RCS header, or NULL if empty.
FIXME: RCS_delaccess would also seem to use "" for empty. We
should pick one or the other. */
    char *access;
    /* Raw data on locked revisions. The first time that RCS_getlocks is

called, we parse these into > locks, and free > lock data. */
char *locks data;

/* List of nodes, the key of which is the numeric revision and the
data of which is the user that it corresponds to (malloc'd). */
List *locks;

/* Set for the strict keyword from the RCS header. */
int strict locks;

/* Value for the comment keyword from RCS header (comment leader), or
NULL if omitted. */
char *comment;

/* List of nodes, the key of which is the numeric revision and the
data of which is the user that it corresponds to (malloc'd). */
List *locks;

/* Set for the strict keyword from the RCS header. */
int strict locks;

/* Value for the comment keyword from RCS header (comment leader), or
NULL if omitted. */
char *comment;

/* File offset of the first deltatext node, so we can seek there. */
long delta pos;

/* Newphrases from the RCS header. List of nodes, the key of which
is the "id" which introduces the newphrase, and the value of which
is the value from the newphrase. */
List *other;

typedef struct rcsnode RCSNode;

struct deltatext {
char *version;

/* Log message, or NULL if we do not intend to change the log message
(that is, RCS_copydeltas should just use the log message from the
file). */
char *log;

/* Change text, or NULL if we do not intend to change the change text
(that is, RCS_copydeltas should just use the change text from the
file). Note that it is perfectly legal to have log be NULL and
text non-NULL, or vice-versa. */
char *text;
size_t len;

/* Newphrase fields from deltatext nodes. FIXME: duplicates the
other field in the rcsversnode, I think. */
List *other;
}
typedef struct deltatext Deltatext;

struct rcsversnode {

/* Duplicate of the key by which this structure is indexed. */
char *version;

char *date;
char *author;
char *state;
char *next;
int dead;
int outdated;
Deltatext *text;

/* Newphrases from deltatext nodes. */
List *other;
Deltatext *text;

/* Newphrase fields from delta nodes. */
List *other;
Deltatext *text;

/* Hard link information for each revision. */
List *hardlinks;
}
typedef struct rcsversnode RCSVers;

/* CVS reserves all even-numbered branches for its own use. "magic" branches
are contained as virtual revision numbers (within symbolic
tags) off the RCS_MAGIC_BRANCH, which is 0. CVS also reserves the
"1." branch for vendor revisions. So, if you do your own branching, you
should limit your use to odd branch numbers starting at 3. */
define RCS_MAGIC_BRANCH 0

/* The type of a function passed to RCS_checkout. */
typedef void (*RCSCHECKOUTPROC) PROTO ((void *, const char *, size_t));

define __STDC__
struct rebuffer;

#ifndef

/* exported interfaces */
RCSNode *RCS_parsePROTO((const char *file, const char *repos));
RCSNode *RCS_parsecfilePROTO((char *rczfile));
void RCS_fully_parsePROTO(RCSTagNode *);
void RCS_reparacfilePROTO(RCSTagNode *, FILE **, struct rebuffer *);

char *RCS_checkkflagPROTO((const char *arg));
char *RCS_getdatePROTO(RCSTagNode *, char *date, int force_tag_match);
char *RCS_gettagPROTO(RCSTagNode *, rcs, char *syntag, int force_tag_match, int *simple_tag);
int RCS_getrevrevPROTO(RCSTagNode *, char *rev);
int RCS_get2revPROTO(RCSTagNode *, rcs, char *tag);
char *RCS_get2tagPROTO(RCSTagNode *, rcs, char *tag);
char *RCS_getversionPROTO(RCSTagNode *, rcs, char *tag, char *date, int force_tag_match, int *simple_tag);
char *RCS_getremoteversionPROTO(struct FileInfo *, char *tag, char **local_tag);

char *RCS_magicrevPROTO(RCSTagNode *, rcs, char *rev);
int RCS_abranchPROTO(RCSTagNode *, rcs, const char *rev);
int RCS_nodeisbranchPROTO(RCSTagNode *, rcs, const char *tag);
char *RCS_whatbranchPROTO(RCSTagNode *, rcs, const char *tag);
char *RCS_headPROTO(RCSTagNode *, rcs);
int RCS_datecmpPROTO((char *date1, char *date2));
time_t RCS_gettimePROTO(RCSTagNode *, rcs, char *tag);
int RCS_validrevPROTO((char *rev));

List *RCS_getlocksPROTO(RCSTagNode *rcz);
void freescznodePROTO(RCSTagNode **rczn);
char *RCS_getbranchPROTO(RCSTagNode *, rcs, char *tag, int force_tag_match);
char *RCS_getbranchheadPROTO((RCSTagNode *, rcs, char *rev));
int RCS_jloadPROTO((RCSTagNode *, rcs), const char *);  
char *RCS_getexpandPROTO ((RCSTagNode *));  
int RCS_checkoutPROTO ((RCSTagNode *));
RCS_CheckoutPROC; void *);  
int RCS_checkinPROTO (RCSTagNode *, rcs, char *workfile, char *message, char *rev, int flag);
int RCS_cmp_filePROTO (RCSTagNode *, char *, char *);  
int RCS_settagPROTO (RCSTagNode *, const char *, const char *);
int RCS_deltagPROTO (RCSTagNode *, const char *, const char *);
int RCS_setbranchPROTO (RCSTagNode *, const char *);
int RCS_lockPROTO (RCSTagNode *, const char *);
int RCS_unlockPROTO (RCSTagNode *, const char *, int);
int RCS_delete_protoPROTO (RCSTagNode *, char *, int);
void RCS_deletePROTO (RCSTagNode *, char *);
void RCS_delaccessPROTO (RCSTagNode *, char *);

char *RCS_getaccessPROTO (RCSTagNode *);
void RCS_rewritePROTO ((RCSTagNode *, Deltatext *, char *));
int RCS_change_textPROTO ((const char *, char *, char *, size_t, char *, size_t, char **, size_t *));
char *make_file_labelPROTO ((char *, char *, char *, const char *, char *, size_t, size_t, char *, FILE *));

extern int preserve_perms;

/* From import.c */

extern int add_rcs_filePROTO ((char *, char *, char *, char *, char *, char *, int, char **, char *, size_t, char *, FILE *));
A.46  rcscmds.c

/*
  * Copyright (c) 1992, Brian Berliner and Jeff Polk
  * Copyright (c) 1989-1992, Brian Berliner
  * You may distribute under the terms of the GNU General Public License as
  * specified in the README file that comes with the CVS source distribution.
  * The functions in this file provide an interface for performing
  * operations directly on RCS files.
  */

#include "cvs.h"
#include <assert.h>
#include <diffrun.h>
#include "diffrun.h"

/* This file, rcs.h, and rcs.c, together sometimes known as the “RCS
library”, are intended to define our interface to RCS files.

Whether there will also be a version of RCS which uses this
library, or whether the library will be packaged for uses beyond
CVS or RCS (many people would like such a thing) is an open
question. Some considerations:

1. An RCS library for CVS must have the capabilities of the
   existing CVS code which accesses RCS files. In particular, simple
   approaches will often be slow.

2. An RCS library should not use code from the current RCS
   (5.7 and its ancestors). The code has many problems. Too few
   comments, too many layers of abstraction, too many global variables
   (the correct number for a library is zero), too much intricately
   interwoven functionality, and too many clever hacks. Paul Eggert,
   the current RCS maintainer, agrees.

3. More work needs to be done in terms of separating out the RCS
   library from the rest of CVS (for example, cvs_output should be
   replaced by a callback, and the declarations should be centralized
   into rcs.h, and probably other such cleanups).

4. To be useful for RCS and perhaps for other uses, the library
   may need features beyond those needed by CVS.

5. Any changes to the RCS file format *must* be compatible. Many,
   many tools (not just CVS and RCS) can at least import this format.
   RCS and CVS must preserve the current ability to import/export it
   (preferably improved magic branches are currently a roadblock).
   See doc/RCSFILES in the CVS distribution for documentation of this
   file format.

On a related note, see the comments at diff(run), later in this file,
for more on the diff library. */

static void RCS_output_diff_options PROTO ((char *, char *, char *, char *));

/* Stuff to deal with passing arguments the way bldiff/a wants to deal
with them. This is a crudey interface; there is no good reason for it
to resemble a command line rather than something closer to “struct
options” in lsdiff. */

/* First call call_diff_setup to setup any initial arguments. The
argument will be parsed into whitespace separated words and added
to the global call_diff_arg list.
Then, optionally, call call_diff for each additional argument
that you'd like to pass to the diff library.
Finally, call call_diff or call_diffs to produce the diffs. */

static char **call_diff_argv;
static int call_diffargc;
static int call_diffargc_allocated;

static void call_diffadd_arg PROTO ((const char *));
static void call_diffsetup PROTO ((const char *prog));
static int call_diff PROTO ((char *out));
static int call_diffs PROTO ((char *out));

static void call_diffwrite_output PROTO((const char *, size_t));
static void call_diffflush_output PROTO(void);
static void call_diffwrite_stout PROTO((const char *));
static void call_differror PROTO((const char *, const char *, const char *));

/* VARARGS */
static void
call_diffsetup (prog)
    const char *prog;
static void
call_diff_arg (s)
{
    call_diff_arg = 0;
    call_diff_arg[0] = (char *) 0;
}

static void
call_diff_add_arg (s)
{
    /* allocate more arg entries if we've run out */
    if (call_diff_arg >= call_diff_arg_allocated)
        
        call_diff_arg = (char *)
        realloc ((char *) call_diff_arg,
                call_diff_arg_allocated + sizeof (char *));

    if (s)
        call_diff_arg[call_diff_arg++] = xstrdup (s);
    else
        /* Not post-incremented on purpose! */
        call_diff_arg[call_diff_arg] = (char *) 0;
}

/* Callback function for the diff library to write data to the output file. This is used when we are producing output to stdout. */

static void
call_diff_write_output (text, len)
{
    cvs_output (text, len);
}

/* Callback function for the diff library to write to stderr. */

static void
call_diff_write_stderr (text)
{
    cvs_output (text, 0);
}

/* Callback function for the diff library to write to stdout. */

static void
call_diff_error (format, a1, a2)
{
    const char *format;
    const char *a1;
    const char *a2;
    
    printf (format, a1, a2);
    
    flush output;
    
    free (call_diff_arg);
}
/* FIXME: Should we somehow indicate that this error is coming from the diff library? */
error (0, 0, format, a1, a2);
}

/* This set of callback functions is used if we are sending the diff
to stdout. */
static struct diff_cbs allbacks call_diff stdout allbacks =
{
    call_diff_write output,
    call_diff-null output,
    call_diff-write stdout,
    call_diff-error
};

/* This set of callback functions is used if we are sending the diff
to a file. */
static struct diff_cbs allbacks call_diff file allbacks =
{
    (void (+) PROTO((const char *, size_t)) NULL,
    (void (+) PROTO((void)) NULL,
    call_diff_write stdout,
    call_diff-error
};

static int
call_diff out
char * out;

if (out == RUSAGE_TTY)
    return diff2run (call_diff_argc, call_diff_argv, NULL,
and call_diff-allbacks out);
else
    return diff2run (call_diff_argc, call_diff_argv, out,
and call_diff-allbacks out);

static int
call_diff out
char * out;

if (out == RUSAGE_TTY)
    return diff2run (call_diff_argc, call_diff_argv, NULL,
and call_diff-allbacks out);
else
    return diff2run (call_diff_argc, call_diff_argv, out,
and call_diff-allbacks out);

/* Merge revisions REV1 and REV2. */

int
RCS_merge (rcs, path, workfile, options, rev1, rev2)
RCSNode * rcs;
char * path;
char * workfile;
char * options;
char * rev1;
char * rev2;

char * xrev1, * xrev2;
char * tmp1, * tmp2;
char * diffout = NULL;
int retval;

if (options != NULL & options[0] != ’V’)
    assert (options[0] == ’-’ & options[1] == ’v’);

cvs_output (*RCS_file, ’o’, 0);
cvs_output (rcs->path, 0);
cvs_output (*’\n’, 1);

/* Calculate numeric revision numbers from rev1 and rev2 (may be
symbolic). */
xrev1 = RCS_gettag (rcs, rev1, 0, NULL);
xrev2 = RCS_gettag (rcs, rev2, 0, NULL);

cvs_output (‘retrieving revision ’, 0);
cvs_output (xrev1, 0);
cvs_output (‘\n’, 1);
tmp1 = cvs_remirror();
if (RCS_checkout (rcs, NULL, xrev1, rev1, options, tmp1,
(RCSCHECKOUTPROC)(NULL))

    cvsa_output("revisions co failed\n", 0);
    error_exit();
}

    cvsa_output("retrieving revision \", 0);
    cvsa_output(xrev2, 0);
    cvsa_output("\n", 1);
    tmp2 = cvsa_tmp(filename());

    if (RCSCHECKOUT (rcs, NULL, xrev2, rev2, options, tmp2,
    (RCSCHECKOUTPROC)(NULL))
    {
        cvsa_output("revisions co failed\n", 0);
        error_exit();
    }

    /* Merge changes. */
    cvsa_output("Merging differences between \", 0);
    cvsa_output(xrev1, 0);
    cvsa_output(\", 0);
    cvsa_output(xrev2, 0);
    cvsa_output(\", 0);
    cvsa_output(workfile, 0);
    cvsa_output("\n", 1);

    /* Remember that the first word in the 'call diff\nsetup' string is used now
     * only for diagnostic messages – CVS no longer forks to run diff3. */
    call_diff_setup("diff3");
    call_diff_arg("-w");
    call_diff_arg("-c");
    call_diff_arg("-L");
    call_diff_arg(workfile);
    call_diff_arg(xrev1);
    call_diff_arg(xrev2);

    call_diff_arg(tmp1);
    call_diff_arg(tmp2);
    retval = call_diff3(diffout);
    if (retval == 1)
        cvsa_output("revisions warning: conflicts during merge\n", 0);
    else if (retval == 2)
        error_exit();
    if (diffout)
        copy_file(diffout, workfile);

    /* Clean up. */
    int save_noexec = noexec;
    noexec = 0;
    if (unlink_file(tmp1) < 0)
    {
        if (existance_error(erno))
            error(0, errno, "cannot remove temp file \x", tmp1);
    }
    free(tmp1);
    if (unlink_file(tmp2) < 0)
    {
        if (existance_error(erno))
            error(0, errno, "cannot remove temp file \x", tmp2);
    }
    free(tmp2);
    if (diffout)
    {
        if (unlink_file(diffout) < 0)
        {
            if (existance_error(erno))
                error(0, errno, "cannot remove temp file \x", diffout);
        }
        free(diffout);
    }
    free(xrev1);
    free(xrev2);
    noexec = save_noexec;

    return retval;
}

/* Diff revisions and/or files. OPTS controls the format of the diff
    (it contains options such as "-w -c", &c), or "" for the default.
OPTIONS controls keyword expansion, as a string starting with "k",
"r" to use the default. REV1 is the first revision to compare
against; it must be non-NULL. If REV2 is non-NULL, compare REV1
and REV2; if REV2 is NULL compare REV1 with the file in the working
directory whose name is WORKFILE. LABEL1 and LABEL2 are default
file labels, and (if non-NULL) should be added as -L options
to diff. Output goes to stdout.

Return value is 0 for success, -1 for a failure which set errno,
or positive for a failure which printed a message on stderr.

This used to exec rcsdiff, but now calls RCS\_checkout and diff\_exec.

An issue is what timezone is used for the dates which appear in the
diff output. rcsdiff uses the -z flag, which is not presently
processed by CVS diff, but I'm not sure exactly how hard to worry
about this–any such features are undocumented in the context of
CVS, and I'm not sure how important to users. */

```c
int RCS\_exec\_rcsdiff (rcsfile, opts, options, rev1, rev2, label1, label2, workfile)
RCSNode *rcsfle;

```char opts;
char options;
char rev1;
char rev2;
char label1;
char label2;
char *workfile;
{
    char *tmpfile1;
    char *tmpfile2;
    int status, retval;

tmpfile1 = cvsp\_temp\_name ();
tmpfile2 = NULL;

cvs\_output (*\"
........................................................................\"
RCS file: *.0);
cvs\_output (rcsfile->path, 0);

cvs\_output (*"mv\", 1);
/* Historically, 'cvs diff' has expanded the $Name keyword to the
empty string when checking out revisions. This is an accident,
but no one has considered the issue thoroughly enough to determine
what the best behavior is. Passing NULL for the 'nametag' argument
preserves the existing behavior. */
cvs\_output (*"retrieving revision *", 0);
cvs\_output (rev1, 0);
cvs\_output (*"mv", 1);
status = RCS\_checkout (rcsfile, NULL, rev1, NULL, options, tmpfile1,
(RCS\_CHECKOUTPROC)0, NULL);
if (status > 0)
{
    retval = status;
    goto error\_return;
}
else if (status < 0)
{
    error (0, errno,
        "cannot check out revision %s of %s", rev1, rcsfile->path);
    retval = 1;
    goto error\_return;
}
if (rev2 == NULL)
{
    assert (workfile != NULL);
    use\_file2 = workfile;
}
else
{
    tmpfile2 = cvsp\_temp\_name ();
cvs\_output (*"retrieving revision *", 0);
cvs\_output (rev2, 0);
cvs\_output (*"mv", 1);
status = RCS\_checkout (rcsfile, NULL, rev2, NULL, options,
    directory, workfile, tmpfile2, (RCS\_CHECKOUTPROC)0, NULL);
if (status > 0)
{
    retval = status;
    goto error\_return;
}
else if (status < 0)
{
    error (0, errno,
        "cannot check out revision %s of %s", rev2, rcsfile->path);
    return 1;
}
Show differences between two files. This is the start of a diff library.

Some issues:

- Should option parsing be part of the library or the caller? The former allows the library to add options without changing the callers, but it causes various problems. One is that something like –brief really wants special handling in CVS, and probably the caller should retain some flexibility in this area. Another is that as things stand currently, there is no separate namespace for diff options versus “cvs diff” options like -l (that is, if the library adds an option which conflicts with a CVS option, it is trouble).

- This isn’t required for a first-cut diff library, but if there would be a way for the caller to specify the timestamps that appear in the diffs (rather than the library getting them from the files), that would clean up the kludgy utime() calls in patch.c.

Show differences between FILE1 and FILE2. Either one can be DEVNULL to indicate a nonexistent file (same as an empty file currently, I suspect, but that may be an issue in and of itself).

OPTIONS is a list of diff options, or "" if none. At a minimum, CVS expects that -c (update.c, patch.c) and -n (update.c) will be supported. Other options, like -u, -speed-large-files, &c., will be specified if the user specified them.

OUT is a filename to send the diffs to, or RUN_TTY to send them to stdout. Error messages go to stderr. Return value is 0 for success, -1 for a failure which set errno, 1 for success (and some differences were found), or >1 for a failure which printed a message on stderr. */

int
diff_exec (file1, file2, options, out)
char *file1;
char *file2;
char *options;
char *out;
{ char *args;
ifdef PRESERVE_PERMISSIONS_SUPPORT
/* If either file1 or file2 are special files, pretend they are /dev/null. Reason: Suppose a file that represents a block special device in one revision becomes a regular file. CVS must find the 'difference' between these files, but a special file contains no data useful for calculating this metric. The safe thing to do is to treat the special file as an empty file, thus recording the regular file's full contents. Doing so will create extremely large deltas at the point of transition between device files and regular files, but this is probably very rare anyway.

There may be ways around this, but I think they are fraught with danger. *stop */

if (preserve_perms & &
    strcmp (file1, DEVNULL) != 0 & &
    strcmp (file2, DEVNULL) != 0)
{
    struct stat sb1, sb2;

    if (CVS_LSTAT (file1, &sb1) < 0)
        error (1, errno, "cannot get file information for %s", file1);
    if (CVS_LSTAT (file2, &sb2) < 0)
        error (1, errno, "cannot get file information for %s", file2);

    if (IS_REG (sb1.st_mode) & & IS_SDIR (sb1.st_mode))
        file1 = DEVNULL;
    if (IS_REG (sb2.st_mode) & & IS_SDIR (sb2.st_mode))
        file2 = DEVNULL;
}
#endif

args = xmalloc (strlen (options) + 10);
/* The first word in this string is used only for error reporting. */
 sprintf (args, "diff %s", options);
call_diff_setup (args);
call_diff_arg (file1);
call_diff_arg (file2);
free (args);

return call_diff (out);
}

int
diff_execv (file1, file2, label1, label2, options, out)
char *file1;
char *file2;
char *label1;
char *label2;
char *options;
char *out;
{
c char *args;

ifdef PRESERVE_PERMISSIONS_SUPPORT
/* Pretend that special files are /dev/null for purposes of making diffs. See comments in diff_exec. */

if (preserve_perms & &
    strcmp (file1, DEVNULL) != 0 & &
    strcmp (file2, DEVNULL) != 0)
{
    struct stat sb1, sb2;

    if (CVS_LSTAT (file1, &sb1) < 0)
        error (1, errno, "cannot get file information for %s", file1);
    if (CVS_LSTAT (file2, &sb2) < 0)
        error (1, errno, "cannot get file information for %s", file2);

    if (IS_REG (sb1.st_mode) & & IS_SDIR (sb1.st_mode))
        file1 = DEVNULL;
    if (IS_REG (sb2.st_mode) & & IS_SDIR (sb2.st_mode))
        file2 = DEVNULL;
}
#endif

args = xmalloc (strlen (options) + 10);
/* The first word in this string is used only for error reporting. */
/* I guess we are pretty confident that options starts with a space. */

sprintf (args, "diff %s", options);
call_diff_setup (args);
if (label1)
    call_diff_arg (label1);
if (label2)
    call_diff_arg (label2);
call_diff_arg (file1);
call_diff_arg (file2);
free (args);
return call_diff (out);
}

/* Print the options passed to DIFF, in the format used by rcsdiff.
   The rcsdiff code that produces this output is extremely hairy, and
   it is not clear how rcsdiff decides which options to print and
   which not to print. The code below reproduces every rcsdiff run
   that I have seen. */

static void
RCS_output_diff_options (opts, rev1, rev2, workfile)
    char *opts;
    char *rev1;
    char *rev2;
    char *workfile;
{
    char *tmp;
    tmp = (char *) xmalloc (strlen (opts) + strlen (rev1) + 10);
    sprintf (tmp, "diff%s -r%s",
            opts, rev1);
    cvs_output (tmp, 0);
    free (tmp);
    if (rev2)
        {
            cvs_output (" -r", 3);
            cvs_output (rev2, 0);
        }
    else
        {
            assert (workfile != NULL);
            cvs_output (" ", 1);
            cvs_output (workfile, 0);
        }
    cvs_output ("\n", 1);
A.47 recurse.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * 
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * 
 * General recursion handler
 * */

#include "cvs.h"
#include "sarecord.h"
#include "fileattr.h"
#include "edit.h"

ifdef CLIENT
include "savecwd.h"
endif

ifdef CLIENT
include "cvs.h"
endif

static int do__argument__proc PROTO((Node * p, void *closure));

static int do__dir__proc PROTO((Node * p, void *closure));

static int do__list__proc PROTO((Node * p, void *closure));

static void addlist PROTO((List **listp, char *dir, char *file));

struct frame {
    FILEPROC fileproc;
    FILESDONEPROC filesdoneproc;
    DIRENTPROC direntproc;
    DIRLEAVEPROC dirleaveproc;
    Dtype flags;
    int which;
    int aflag;
    int readdirlock;
    int do coerce;
};

static int do__recursion__proc PROTO ((struct recursion_frame *frame));

/* I am half tempted to shove a struct file_info into the struct recursion_frame (but
then we would need to modify or create a recursion_frame for each file), or shove
a struct recursion_frame into the struct file_info (more tempting, although
it isn't completely clear that the struct file_info should contain info about
recursion processor internals). So instead use this struct. */

struct frame_and_file {
    struct recursion_frame *frame;
    struct file_info *info;
};

/* Similarly, we need to pass the entries list to do__dir__proc. */

struct frame_and_entries {
    struct recursion_frame *frame;
    List **entries;
};

ifdef CLIENT
/* This is a callback to send "Argument" commands to the server in the
case we've done a "cvs update" or "cvs commit" in a top-level
directory where there is no CVSADM directory. */

static int do__argument__proc (p, closure)
    Node *p;
    void *closure;
{
    char *dir = p->key;
    send__to_server ("Argument ", 0);
    send__to_server (dir, 0);
    send__to_server ("\012", 1);
    return 0;
}
#endif

/* Start a recursive command.

    Command line arguments (ARGC, ARGV) dictate the directories and
files on which we operate. In the special case of no arguments, we
default to ".". */

int start_recursion (fileproc, filesdoneproc, direntproc, dirleaveproc, callerdat,
argy, argv, local, which, aflag, readlock,
update_preload, dorks)
FILEPROC fileproc;
FILESDONEPROC filesdoneproc;
DIRENTPROC direntproc;
DIRLEAVEPROC dirleaveproc;
void callerdat;
int argc;
char ++argv;
int local;

/* This specifies the kind of recursion. There are several cases:

1. W_LOCAL is set but W_REPO or W_ATTIC is. The current
directory when we are called must be the repository and
recursion proceeds according to what exists in the repository.

2a. W_LOCAL is set but W_REPO and W_ATTIC are not. The
current directory when we are called must be the working
directory. Recursion proceeds according to what exists in the
working directory, never (I think) consulting any part of the
repository which does not correspond to the working directory
("correspond" == Name_Repository).

2b. W_LOCAL is set and so is W_REPO or W_ATTIC. This is the
weird one. The current directory when we are called must be the
working directory. We recurse through working directories,
but we recurse into a directory if it is exists in the working
directory or if it exists in the repository. If the directory
does not exist in the working directory, the direntproc must
either tell us to skip it (R_SKIP_DIRS), or must create it (I
think those are the only two cases). */
int which;

int aflag;
int readlock;
char ++update_preload;
int dorks;

{ int i, err = 0;
  List *file_list = NULL;
  struct recursion_frame frame;
  frame.fileproc = fileproc;
  frame.filesdoneproc = filesdoneproc;
  frame.direntrproc = direntproc;
  frame.dirleaveproc = dirleaveproc;
  frame.callerdat = callerdat;
  frame.flags = local ? (R_SKIP_DIRS | R_PROCESS);
  frame.which = which;
  frame.aflag = aflag;
  frame.readlock = readlock;
  frame.dors = dorks;
  expandwild (argc, argv, &argc, &argv);
  if (update_preload == NULL)
    update_preload = xstrdup (**);
  else
    update_preload = xstrdup (update_preload);

  /* clean up from any previous calls to start/recursion */
  if (repository)
  { free (repository);
    repository = (char *) NULL;
  }
  if (filelist)
    dellist (&filelist); /* FIXME-krp: no longer correct. */
  if (dirlist)
    dellist (&dirlist);

  ifdef SERVER_SUPPORT
  if (server_active)
  { for (i = 0; i < argc; ++i)
      server_getpathname_check (argv[i]);
  }
  endif

  if (argc == 0)
  {
    /* There were no arguments, so we'll probably just recurse. The
     exception to the rule is when we are called from a directory
    * without any CVS administration files. That has always meant to
    * process each of the sub-directories, so we pretend like we were
    * called with the list of sub-dirs of the current dir as args
    */
if ((which & W_LOCAL) && isdir (CVSADM))
{
    dirlist = FindDirectories ((char *) NULL, W_LOCAL, (List *) NULL);
    /* If there are no sub-directories, there is a certain logic in favor of
doing nothing, but in fact probably the user is just confused about what
directory they are in, or whether they cvs add'd a new directory. In the case of at least one
sub-directory, at least when we recurse into them we notice (hopefully) whether they are under CVS control. */
    if (listisempty (dirlist))
    {
        if (update[0] == '\0')
            error (0, 0, "in directory .");
        else
            error (0, 0, "in directory "$update", update$dir);
        error (1, 0, "there is no version here; run '%s checkout' first", program_name);
    }
    #ifdef CLIENT_SUPPORT
    else if (clientactive && serverstarted)
    {
        /* In the the case "cvs update foo bar baz", a call to
send_file_names in update.c will have sent the
appropriate "Argument" commands to the server. In
this case, that won't have happened, so we need to
do it here. While this example uses "update", this
generalizes to other commands. */
        err += walklist (dirlist, do_argument, proc, NULL);
    #endif
    }
    else
        addlist (&dirlist, ".");
    err += do_recursion (&frame);
    goto out;
}

/* There were arguments, so we have to handle them by hand. To do
* that, we set up the filelist and dirlist with the arguments and
* call do_recursion. do_recursion recognizes the fact that the
* lists are non-null when it starts and doesn't update them.
* * explicitly named directories are stored in dirlist.
* * explicitly named files are stored in filelist.
* * other possibility is named entities whicha are not currently in
* the working directory.
*/
for (i = 0; i < argc; i++)
{
    /* if this argument is a directory, then add it to the list of
directories. */
    if (iswrapname (argv[i], WRAP_TOCVS) & & isdir (argv[i]))
        addlist (&dirlist, argv[i]);
    else
    {
        /* otherwise, split argument into directory and component names. */
        char dir;
        char *comp;
        char *file_ttry;
        /* Now break out argv[i] into directory part (DIR) and file part (COMP).
        DIR and COMP will each point to a newly malloc'd string. */
        dir = xstrdup (argv[i]);
        comp = last_component (dir);
        if (comp == dir)
        {
            /* no dir component. What we have is an implied "/" */
            dir = xstrdup("/");
        }
        else
        {
            char *p = comp;

            p[-1] = '\0';
            comp = xstrdup (p);
        }
        /* if this argument exists as a file in the current
        working directory tree, then add it to the files list. */
        if ((which & W_LOCAL))
        {

if (which & WLOCAL) &;& isdir (CVSADM)
#endif
endif

} /* otherwise, look for it in the repository. */
char *tmp_update_dir;
char *repos;
char *reposfile;

tmp_update_dir = xmalloc(strlen(update_dir) + strlen(dir) + 5);
strcpy(tmp_update_dir, update_dir);
if (*tmp_update_dir == "\")
  (void) strcat(tmp_update_dir, "/");
(void) strcat(tmp_update_dir, dir);

/* look for it in the repository. */
repos = Name(repository, dir, tmp_update_dir);
reposfile = xmalloc(strlen(repos) + strlen(comp) + 5);
(void) sprintf(reposfile, "%s/%s", repos, comp);
free(repos);

if (!walkname_has (comp, WRAP_TOCVS) &;& isdir (reposfile))
  addfile (&dirlst, argv[i]);
else
  addfile (&files_by_dir, dir, comp);
  free (tmp_update_dir);
  free (reposfile);
} else
  addfile (&files_by_dir, dir, comp);

else
  error (1, 0, "no such directory \\
  "name", dir);
  free (file_to_try);
  free (dir);
  free (comp);
}

/* At this point we have looped over all named arguments and built
   a couple lists. Now we unroll the lists, setting up and
calling do_recursion. */
err += walklist (files_by_dir, unroll_files, proc, (void *) &frame);
dellist(&files_by_dir);
/* then do recursion on the dirlst. */
if (dirlst != NULL)
  err += do_recursion (&frame);
/* Free the data which expandallocated. */
free_names (&args, argv);
out:
  free (update_dir);
  update_dir = NULL;
  return (err);
}

/* Implement the recursive policies on the local directory. This may be
   called directly, or may be called by start_recursion
   */
static int
do_recursion (frame)
struct recursion_frame *frame;
{
  int err = 0;
  int doneproc = 1;
char *repository;
List *entries = NULL;
int should_readlock;

/* do nothing if told */
if ((frame->flags == R_SKIP_ALL))
    return (0);

should_readlock = noexec ? 0 : frame->readlock;

/* The fact that locks are not active here is what makes us fail to have
the
If someone commits some changes in one cvs command,
then an update by someone else will either get all the
changes, or none of them.

property (see node Concurrency in cvs.texinfo).
The most straightforward fix would just to readlock the whole
tree before starting an update, but that means that if a commit
gets blocked on a big update, it might need to wait a *long* time.

A more adequate fix would be a two-pass design for update,
checkout, etc.. The first pass would go through the repository,
with the whole tree readlocked, noting what versions of each
file we want to get. The second pass would release all locks
(except perhaps short-term locks on one file at a
time—although I think RCS already deals with this) and
actually get the files, specifying the particular versions it wants.

This could be sped up by separating out the data needed for the
first pass into a separate file(s) for example a file
attribute for each file whose value contains the head version
for each branch. The structure should be designed so that
commit can relatively quickly update the information for a
single file or a handful of files (file attributes, as
implemented in Jan 96, are probably acceptable; improvements
would be possible such as branch attributes which are in
separate files for each branch). */

#define defined(x) defined(x) && defined(x)
/* Now would be a good time to check to see if we need to stop
* generating data, to give the buffers a chance to drain to the
* remote client. We should not have locks active at this point. */
if (server_active /* If there are writelocks around, we cannot pause here */
    && (should_readlock || noexec))
    server_pause_check();

/* Fill in repository with the current repository */
if ((frame->which & W_LOCAL))
    { if (isdir (CVSADM))
        repository = NameRepository ((char *) NULL, update_dir);
    else
        repository = NULL;
    }
else
    { repository = xgetwd ();
        if (repository == NULL)
            error (1, errno, "could not get working directory");
    }
repository = repository; /* remember what to free */

fileattr_startdir (repository);
/* The filesdoneproc needs to be called for each directory where files
* processed, or each directory that is processed by a call where no
* directories were passed in. In fact, the only time we don't want to
* call back the filesdoneproc is when we are processing directories that
* were passed in on the command line (or in the special case of "." when
* we were called with no args */
if (dirlist != NULL && filelist == NULL)
    filesdoneproc = 0;
/* If filelist or dirlist is already set, we don't look again. Otherwise,
* find the files and directories */
/* if (filelist == NULL && dirlist == NULL) */
/* both lists were NULL, so start from scratch */
if (frame->fileproc != NULL && frame->flags != R_SKIP_FILES)
{
    /* be sure to look in the attic if we have sticky tags/date */
    if ((which & W_ATTIC) == 0)
        which |= W_ATTIC;

    /* In the (which & W_LOCAL) case, we filled in repository earlier in the function. In the (which & W_LOCAL) case, the FindNames function is going to look through the Entries file. If we do not have a repository, that does not make sense, so we insist upon having a repository at this point. Namerepository will give a reasonable error message. */
    if (repository == NULL)
        repository = NameRepository ((char *) NULL, update_dir);

    /* find the files and fill in entries if appropriate */
    filelist = FindNames (repository, which, frame->flags, &entries);
}

/* find sub-directories if we will recurse */
if (frame->flags != R_SKIP_DIRS)
dirlist = FindDirectories (repository, frame->which, entries);
else
{
    /* something was passed on the command line */
    if (filelist != NULL && frame->fileproc != NULL)
    {
        /* we will process files, so pre-parse entries */
        if (frame->which & W_LOCAL)
            entries = EntriesOpen (frame->flags, NULL);
    }
}

/* process the files (if any) */
if (filelist != NULL && frame->fileproc)
{
struct fileinfo finfo;
struct frame_and_file ffrile;

    /* read lock it if necessary */
    if (should_readlock && repository && ReaderLock (repository) != 0)
        error (1, 0, "read lock failed - giving up");
}

ifdef CLIENT_SUPPORT
/* For the server, we handle notifications in a completely different place (server_notify). For local, we can't do them here--we don't have writelocks in place, and there is no way to get writelocks here. */
#endif
if (client_active)
    notify_check (repository, update_dir);
endif /* CLIENT_SUPPORT */

findstruct.repository = repository;
findstruct.update_dir = update_dir;
findstruct.entries = entries;
/* do_fileproc will fill in findstruct.file. */
ffrile.finfo = &finfo;
ffrile.frame = frame;

/* process the files */
err += walklist (filelist, do_fileproc, &ffrile);

/* unlock it */
if (should_readlock)
    LockCleanup ();

/* clean up */
dellist (&filelist);

/* call-back files done proc (if any) */
if (do_fileproc & frame->flags) frame->filelistproc != NULL)
    err = frame->filelistproc (frame->callerdat, err, repository,
update_dir[0] ? update_dir : ".",
entries);

fileattr_write ();
fileattr_free ();

/* process the directories (if necessary) */
if (dirlist != NULL)
{
566 – Source code

```c
struct frame_and_entries frent;

frent.frame = frame;
frent.entries = entries;
err += walklist (dirlist, do_dirproc, (void *) &frent);
}
#endif
if (frame->dirleaveproc != NULL)
err += frame->dirleaveproc (frame->callerdat, ".", err, ".");
#endif
delists (&dirlist);

if (entries)
{
    EntriesClose (entries);
    entries = NULL;
}

/*@ free the saved copy of the pointer if necessary */
if (repository)
{
    free (repository);
    repository = (char *) NULL;
}
return (err);

/*@ Process each of the files in the list with the callback proc */

static int
do_file_proc (p, closure)
    Node *p;
    void *closure;
{
    struct frame_and_file *frfile = (struct frame_and_file *) closure;
    struct file_info *info = frfile->info;
    int ret;

    frfile->file = p->key;
    info->fullname = xmalloc (strlen (info->file) + 2);
    info->fullname[0] = '\0';
    if (info->update_dir[0] != '\0')
    {
        strcat (info->fullname, info->update_dir);
        strlcat (info->fullname, ".");
    }
    strlcat (info->fullname, frfile->file);

    if (frfile->frame->dosrcs & repository)
        frfile->rcs = RCS_parse (frfile->file, repository);
    else
        frfile->rcs = (RCSNode *) NULL;
    ret = frfile->frame->fileproc (frfile->frame->callerdat, info);
    freercsnode (&info->rcs);
    free (frfile->fullname);

    /* Allow the user to monitor progress with tail -f. Doing this once per file should be no big deal, but we don't want the performance hit of flushing on every line like previous versions of CVS. */
cvs_flushout ();

    return (ret);
}

/*@ Process each of the directories in the list (recursing as we go) */

static int
do_dir_proc (p, closure)
    Node *p;
    void *closure;
{
    struct frame_and_entries *frent = (struct frame_and_entries *) closure;
    struct recursion *frame = frent->frame;
    struct recursion *xframe;
    char *dir = p->key;

    char *newrepos;
    List *dirlist;
    char *repository;
    Dtype dir->return = RPROCESS;
    int stripped=dot = 0;
    int err = 0;
    struct saved_cwd cwd;
    char *saved_update_dir;
```
if ((ncmp (dir, CVSADM) == 0))
{
  /* This seems to most often happen when users (beginning users,
     generally) try "cvs ci" or something similar. On that
     theory, it is possible that we should just silently skip the
     CVSADM directories, but on the other hand, using a wildcard
     like this isn't necessarily a practice to encourage (it operates
     only on files which exist in the working directory, unlike
     regular CVS recursion). */

  /* FIXME-reentrancy: printed_cvs_msg should be in a "command
     struct" or some such, so that it gets cleared for each new
     command (this is possible using the remote protocol and a
     custom-written client). The struct recursion frame is not
     far back enough though, some commands (commit at least)
     will call start_cvs several times. An alternate solution
     would be to take this whole check and move it to a new function
     validate arguments or some such that all the commands call
     and which maps the offending directory from the argc,argv
     vector. */
  static int printed_cvs_msg = 0;

  if (!printed_cvs_msg)
  {
    error (0, 0, "warning: directory %s specified in argument",
       dir);
    error (0, 0, "/
       but CVS uses %s for its own purposes; skipping %s directory",
       CVSADM, dir);
    printed_cvs_msg = 1;
  }
  return 0;
}

saved_update_dir = update_dir;
update_dir = xmalloc (strlen (saved_update_dir) + strlen (dir) + 5);
strcpy (update_dir, saved_update_dir);
/* set up update_dir - skip dots if not at start */
if ((strcmp (dir, ".")) != 0)
{
  if (update_dir[0] != '\0')
  {
    (void) strcat (update_dir, "/");
    (void) strcat (update_dir, dir);
  }
  else
  {
    (void) strcpy (update_dir, dir);
  }
}

  /* Here we need a plausible repository name for the sub-directory. We
     create one by concatenating the new directory name onto the
     previous repository name. The only case where the name should be
     used is in the case where we are creating a new sub-directory for
     update -d and in that case the generated name will be correct. */

  if (repository == NULL)
  {
    newrepos = xstrdup (**);
  }
  else
  {
    newrepos = xmalloc (strlen (repository) + strlen (dir) + 5);
    sprintf (newrepos, "%s/%s", repository, dir);
    newrepos = xstrdup (**);
  }
  else
  {
    if (update_dir[0] == '\0')
      (void) strcpy (update_dir, dir);
    if (repository == NULL)
      newrepos = xstrdup (**);
    else
      newrepos = xstrdup (repository);
  }

  /* Check to see that the CVSADM directory, if it exists, seems to be
     well-formed. It can be missing files if the user hit "C in the
     middle of a previous run. We want to (a) make this a nonfatal
     error, and (b) make sure we print which directory has the
     problem.

Do this before the direntproc, so that (1) the direntproc
doesn't have to guess/deduce whether we will skip the directory
(e.g. semidirentrc and whether to send the directory), and
(2) so that the warm fuzzy doesn't get printed if we skip the
directory. */
  if (frame->which & W_LOCAL)
  {
    char *cvsadmdir;
568 – Source code

720

cvsadmdir = xmalloc (strlen (dir)
+ sizeof (CVSADM REP)
+ sizeof (CVSADM ENT)
+ 80);
strcpy (cvsadmdir, dir);
strcat (cvsadmdir, "/");
strcat (cvsadmdir, CVSADM);
if (isdir (cvsadmdir))
{
strcpy (cvsadmdir, dir);
strcat (cvsadmdir, "/");
strcat (cvsadmdir, CVSADM REP);
if (!isﬁle (cvsadmdir))
{
/∗ Some commands like update may have printed “? foo” but
if we were planning to recurse, and don’t on account of
CVS/Repository, we want to say why. ∗/
error (0, 0, "ignoring %s (%s missing)", update dir,
CVSADM REP);
dir return = R SKIP ALL;
}

730

740

/∗ Likewise for CVS/Entries. ∗/
if (dir return != R SKIP ALL)
{
strcpy (cvsadmdir, dir);
strcat (cvsadmdir, "/");
strcat (cvsadmdir, CVSADM ENT);
if (!isﬁle (cvsadmdir))
{
/∗ Some commands like update may have printed “? foo” but
if we were planning to recurse, and don’t on account of
CVS/Repository, we want to say why. ∗/
error (0, 0, "ignoring %s (%s missing)", update dir,
CVSADM ENT);
dir return = R SKIP ALL;
}
}

750

760

770

}
free (cvsadmdir);
}

/∗ call-back dir entry proc (if any) ∗/
if (dir return == R SKIP ALL)
;
else if (frame−>direntproc != NULL)
dir return = frame−>direntproc (frame−>callerdat, dir, newrepos,
update dir, frent−>entries);
else
{
/∗ Generic behavior. I don’t see a reason to make the caller specify
a direntproc just to get this. ∗/
if ((frame−>which & W LOCAL) && !isdir (dir))
dir return = R SKIP ALL;
}
free (newrepos);

780

790

/∗ only process the dir if the return code was 0 ∗/
if (dir return != R SKIP ALL)
{
/∗ save our current directory and static vars ∗/
if (save cwd (&cwd))
error exit ();
sdirlist = dirlist;
srepository = repository;
dirlist = NULL;
/∗ cd to the sub-directory ∗/
if ( CVS CHDIR (dir) < 0)
error (1, errno, "could not chdir to %s", dir);
/∗ honor the global SKIP DIRS (a.k.a. local) ∗/
if (frame−>ﬂags == R SKIP DIRS)
dir return = R SKIP DIRS;

800

/∗ remember if the ‘.’ will be stripped for subsequent dirs ∗/
if (strcmp (update dir, ".") == 0)
{
update dir[0] = '\0';
stripped dot = 1;
}
/∗ make the recursive call ∗/
xframe = ∗frame;
xframe.ﬂags = dir return;
err += do recursion (&xframe);


/* put the '.' back if necessary */
if (stripped == NULL) {
    (void) strcpy(update_dir, ".");
}

/* call-back dir leave proc (if any) */
if (frame->dirleaveproc != NULL)
    err = frame->dirleaveproc(frame->callerdat, dir, err, update_dir,
                              frame->entries);

/* get back to where we started and restore state vars */
if (restorecwd(&cwd, NULL)) {
    error(exit());
    free(update_dir);
    update_dir = saved_update_dir;
    return(err);
}

/* Add a node to a list allocating the list if necessary. */
static void addlist(listp, key)
List **listp;
char *key;
{
    Node *n;
    if (*listp == NULL) {
        listp = getlist();
        p = getnode();
        p->type = FILES;
        p->key = xstrdup(key);
        if (addnode(*listp, p) != 0)
            freenode(p);
    }
}

/* Add a file to a list of files. */
static void addfile(listp, dir, file)
List **listp;
char *dir;
char *file;
{
    Node *n;
    /* add this dir. */
    addlist(listp, dir);
    n = findnode(listp, dir);
    if (n == NULL) {
        error(1, 0, "can't find recently added dir node ".n in start_recursion.",
              dir);
    }
    n->type = DIRS;
    addlist((List **) &n->data, file);
    return;
}

static int unroll_files proc (p, closure)
Node *p;
void *eclosure;
{
    Node *n;
    struct recursion_frame *frame = (struct recursion_frame *) closure;
    int err = 0;
    List *save_dirlist;
    char *save_update_dir = NULL;
    struct saved_cwd cwd;
    /* if this dir was also an explicitly named argument, then skip it. We'll catch it later when we do dirs. */
    n = findnode(dirlist, p->key);
    if (n != NULL) {
        return(0);
    }
    /* otherwise, call do_recursion for this list of files. */
    filelist = (List *) p->data;
    p->data = NULL;
    save_dirlist = dirlist;
    dirlist = NULL;
    if (strcmp(p->key, ".") != 0) {
{  
  if (save_cwd (&cwd))  
    error_exit ();  
  if (CVS_CHDIR (p->key) < 0)  
    error (i, errno, "could not chdir to %s", p->key);  
  save_update_dir = update_dir;  
  update_dir = xmalloc (strlen (save_update_dir)  
    + strlen (p->key)  
    + 5);  
  strcpy (update_dir, save_update_dir);  
  if (update_dir != NULL)  
    (void) strcat (update_dir, "/");  
  (void) strcat (update_dir, p->key);  
  }  
  err += do_recursion (frame);  
  if (save_update_dir != NULL)  
  {  
    free (update_dir);  
    update_dir = save_update_dir;  
    if (restore_cwd (&cwd, NULL))  
      error_exit ();  
    free (&cwd);  
  }  
  dirlist = save_dirlist;  
  filelist = NULL;  
  return (err);  
}
A.48  release.c

/∗ Release: "cancel" a checkout in the history log. ∗
- Enter a line in the history log indicating the "release". - If asked to,
- delete the local working directory. ∗/

#include "cvs.h"
#include "getline.h"

static void release_delete_PROTO((char *dir));

static const char *const release_usage[] =
{
    "Usage: %s %s [-d] directories...
    \t-d	Delete the given directory.
    \tSpecify the --help global option for a list of other help options)\n    NULL,
};

#define SERVER_SUPPORT
#endif

static int release_server_PROTO((int argc, char **argv));

/* This is the server side of cvs release. */
static int release_server_PROTO((argc, argv))
    int argc;
    char **argv;
{
    int i;
    /* Note that we skip argv[0]. */
    for (i = 1; i < argc; ++i)
        history_write ('F', argv[i], "", argv[i], "");
    return 0;
}
#endif /* SERVER_SUPPORT */

/∗ There are various things to improve about this implementation:
1. Using run_popen to run "cvs update" could be replaced by a
    fairly simple start_recursion/classify_file loop-a-win for
    portability, performance, and cleanliness. In particular, there is
    no particularly good way to find the right "cvs".
2. The fact that "cvs update" contacts the server slows things down;
    it undermines the case for using "cvs release" rather than "rm .df".
    However, for correctly printing "? foo" and correctly handling
CVSROOTADM_IGNORE, we currently need to contact the server. (One
idea for how to fix this is to stash a copy of CVSROOTADM_IGNORE in
the working directories; see comment at base∗ in entries.c for a
few thoughts on that).
3. Would be nice to take processing things on the client side one step
    further, and making it like edit/unedit in terms of working well if
    disconnected from the network, and then sending a delayed
    notification.
4. Having separate network turnarounds for the "Notify" request
    which we do as part of unedit, and for the "release" itself, is slow
    and unnecessary. */

int release_PROTO((argc, argv))
    int argc;
    char **argv;
{
    FILE *fp;
    int i, c;
    char *repository;
    char *line = NULL;
    size_t allocated = 0;
    char update_command;
    char *thisarg;
    int arg_start_idx;
    int err = 0;
    short delete_flag = 0;

#define SERVER_SUPPORT
if (server_active)
    return release_server_PROTO((argc, argv));
#endif

/* Everything from here on is client or local. */
if (argc == -1)
    usage (release_usage);
    optind = 0;
```c
while ((c = getopt(argc, argv, "+Qdq")) != -1)
{
    switch (c)
    {
    case 'Q':
    case 'q':
        error (1, 0, "-q or -Q must be specified before \"%s\"", command_name);
        break;
    case 'd':
        delete_flag++;
        break;
    case '?':
        default:
            usage (release_usage);
            break;
    }
    argc -= optind;
    argv += optind;
}

/* We're going to run "cvs-n -q update" and check its output; if
 * the output is sufficiently unalarming, then we release with no
 * questions asked. Else we prompt, then maybe release.
 */
update_cmd = xmalloc (strlen (program_path) + strlen (CVSroot_original) + 20);
sprintf (update_cmd, "%s -n -q -d %s update", program_path, CVSroot_original);

#ifdef CLIENT_SUPPORT
    /* Start the server; we'll close it after looping. */
    if (client_active)
    {
        start_server ();
        ignore_setup ();
    }
#endif /* CLIENT_SUPPORT */

arg_start_idx = 0;
for (i = arg_start_idx; i < argc; i++)
{
    thisarg = argv[i];
    if (isdir (thisarg))
    {
        if (CVS_CHDIR (thisarg) < 0)
        {
            if (really_quiet)
                error (0, errno, "can't chdir to: %s", thisarg);
            continue;
        }
        if (isdir (CVSADM))
        {
            if (really_quiet)
                error (0, 0, "no repository directory: %s", thisarg);
            continue;
        }
    }
    else
    {
        if (really_quiet)
            error (0, 0, "no such directory: %s", thisarg);
        continue;
    }

    repository = Name_Repository ((char *) NULL, (char *) NULL);

    if (really_quiet)
    {
        int line_length;

        /* The "release" command piggybacks on "update", which
does the real work of finding out if anything is not
up-to-date with the repository. Then "release" prompts
the user, telling her how many files have been
modified, and asking if she still wants to do the
release. */
        fp = run_popen (update_cmd, "r");
        if (fp == NULL)
            error (1, 0, "cannot run command %s", update_cmd);
        c = 0;
        while ((line_length = getline (kline, &kline_allocated, fp)) >= 0) 
````
if (strchr ("MARC\", line))
  c++;
  (void) printf (line);
} if (line.length < 0 && !feof (fp))
  error (0, errno, "cannot read from subprocess");
/* If the update exited with an error, then we just want to
  complain and go on to the next arg. Especially, we do
  not want to delete the local copy, since it's obviously
  not what the user thinks it is. */
190 if ((pclose (fp)) != 0)
  {
    error (0, 0, "unable to release '%s'", thisarg);
    continue;
  printf ("You have [%d] altered files in this repository.
" , c); printf ("Are you sure you want to release %sdirectory '%s': ",
 delete_flag ? "(and delete) " : "", thisarg);
200 c = !yesno ();
  if (c) /* "No" */
    {
      (void) fprintf (stderr, "** '%s' aborted by user choice.
" , command.name);
      free (repository);
      continue;
    }
210 #ifdef CLIENT_SUPPORT
& & !client.active
  & & (!supported_request ("noop")
| | !supported_request ("Notify")))
#endif
220 { /* We are chdir'd into the directory in question.
  So don't pass args to unedit. */
  int argc = 1;
  char *argv[3];
  argv[0] = "dummy";
  argv[1] = NULL;
  err += unedit (argc, argv);
#endif CLIENT_SUPPORT
230 if (client.active)
  {
    send_to_server ("Argument ", 0);
    send_to_server (thisarg, 0);
    send_to_server ("\012", 1);
    send_to_server ("release\012", 0);
  } else
#endif /* CLIENT_SUPPORT */
240 { /* F = = Free */
    history_write ("F", thisarg, "", thisarg, "");
  free (repository);
  if (delete_flag) release_delete (thisarg);
#endif CLIENT_SUPPORT
250 if (client.active)
  err += get_server_responses ();
#endif /* CLIENT_SUPPORT */
260 if (client.active)
  {
    /* Unfortunately, client.c doesn't offer a way to close
     the connection without waiting for responses. The extra
     network turnaround here is quite unnecessary other than
     that.... */
    send_to_server ("noop\012", 0);
    err += get_server_responses (0);
  }
#endif /* CLIENT_SUPPORT */
270 free (update.cmd);
if (line != NULL)
  free (line);
return err;
/* We want to ”rm -r” the working directory, but let us be a little paranoid. */

static void release_delete (dir)
    char *dir;
{
    struct stat st;
    ino_t ino;
    (void) CVS_STAT (".", &st);
    ino = st.st_ino;
    (void) CVS_CHDIR ("..");
    (void) CVS_STAT (dir, &st);
    if (ino != st.st_ino)
        /* This test does not work on cygwin32, because under cygwin32
         the st_ino field is not the same when you refer to a file
         by a different name. This is a cygwin32 bug, but then I
don’t see what the point of this test is anyhow. */
        ifndef CYGWIN32
            error (0, 0,
            "Parent dir on a different disk, delete of %s aborted", dir);
        return;
        endif
    }  
* XXX - shouldn't this just delete the CVS-controlled files and, perhaps,
* the files that would normally be ignored and leave everything else?
*/
    if (unlink_file (dir) < 0)
        error (0, errno, "deletion of directory %s failed", dir);
}
A.49 remove.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Remove a File
 *
 * Removes entries from the present version. The entries will be removed from
 * the RCS repository upon the next "commit".
 * "remove" accepts no options, only file names that are to be removed. The
 * file must not exist in the current directory for "remove" to work
 * correctly.
 */

#include "cvs.h"

ifdef CLIENT_SUPPORT
static int remove_fileproc Proto((void *callerdata, struct file_info *finfo));
endif
static int remove_fileproc Proto((void *callerdata, struct file_info *finfo));
static Dtype remove_dirproc Proto((void *callerdata, char *dir, char *repo, char *update_dir, List *entries));

static int force;

static int local;
static int removed_files;
static int existing_files;
static const char *const remove_usage[] =
{ "Usage: %s %s [-flR] [files...]
  \t-f\tDelete the file before removing it.
  \t-l\tProcess this directory only(not recursive).
  \t-R\tProcess directories recursively.
  *(Specify the --help global option for a list of other help options)\n", NULL
};

int cvsremove(argc, argv)
int argc;
char **argv;
{ int c, err;

if (argc == -1)
  usage (remove_usage);

optind = 0;
while ((c = getopt (argc, argv, "+flR")) != -1)
{ switch (c)
  { case '+':
      force = 1;
      break;
    case 'l':
      local = 1;
      break;
    case 'R':
      local = 0;
      break;
    case '?':
      default:
      usage (remove_usage);
      break;
  }
  argc -= optind;
  argv += optind;
}
wrap_setup();
#endif CLIENT_SUPPORT

ifdef CLIENT_SUPPORT
if (client_active)
  /* Call expand_wild so that the local removal of files will
   * work. It's ok to do it always because we have to send the
   * file names expanded anyway. */
  expand_wild (argc, argv, &argc, &argv);

if (force)
  { if (!noexec)
{  
    start_recursion (remove_force_fileproc, (FILESDONEPROC) NULL, 
    (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, 
    (void *) NULL, argc, argv, local, W_LOCAL, 
    0, 2, (char *) NULL, 0); 
}  
/* else FIXME should probably act as if the file doesn’t exist 
   in doing the following checks. */

start_server ();
ignore_setup ();
if (local) 
    send_arg ("-l");
/* FIXME: Can’t we set send flavours here? Needs investigation. */
if (send_names (arg, argv, 0);) 
    send_server ("remove\012", 0); 
    return get_responses_and_close ();
}  
#endif

/* start the recursion processor */
err = start_recursion (remove_fileproc, (FILESDONEPROC) NULL, 
remove_dirproc, (DIRLEAVEPROC) NULL, NULL, 
argc, argv, local, W_LOCAL, 0, 1, (char *) NULL, 1); 
if (removed_files) 
    error (0, 0, “use ‘%s commit’ to remove %s permanently”, program_name, 
    (removed_files == 1) ? “this file” : “these files”);
if (existing_files) 
    error (0, 0, 
    (existing_files == 1) ? 
    “%d file exists; remove it first” : 
    “%d files exist; remove them first”), 
    existing_files);
return (err);
}
#endif CLIENT_SUPPORT

/* This is called via start_recursion if we are running as the client 
   and the -f option was used. We just physically remove the file. */
/* ARGUSED */
static int
remove_force_fileproc (callee_dat, finfo)
    void *callee_dat;
    struct file_info *finfo;
{
    if (CVS_\!UNLINK (finfo->file) < 0 & & ! existence_error (errno))
        error (0, errno, “unable to remove %s”, finfo->fullname);
    return 0;
}  
#endif

/* remove the file, only if it has already been physically removed 
   ARGUSED */
static int
remove_fileproc (callee_dat, finfo)
    void *callee_dat;
    struct file_info *finfo;
{
    if (force)
        if (!noexec)
            if (CVS_\!UNLINK (finfo->file) < 0 & & ! existence_error (errno))
                error (0, errno, “unable to remove %s”, finfo->fullname);
    /* else FIXME should probably act as if the file doesn’t exist 
       in doing the following checks. */
    vers = VersionedTS (finfo, NULL, NULL, NULL, 0, 0); 
    if (vers->ts_user != NULL) 
        
}
existing_files++;
if (quiet)
  error (0, 0, "file '%s' still in working directory",
      finfo->fullname);
} else if (vers->vn_user == NULL)
  { 
    if (quiet)
      error (0, 0, "nothing known about '%s'", finfo->fullname);
  }
else if (vers-> vn_user[0] == '0' && vers-> vn_user[1] == '\0')
  { 
    char *fname;
    /* It's a file that has been added, but not committed yet. So,
     * remove the .t file for it and scratch it from the
     * entries file. */
    Scratch_file (finfo->entries, finfo->file);
    fname = xmalloc (strlen (finfo->file) + sizeof (CVSADM) + sizeof (CVSEXT_LOG) + 10);
    (void) sprintf (fname, "%s/%s.tag", CVSADM, finfo->file, CVSEXT_LOG);
    (void) unlink_file (fname);
    if (quiet)
      error (0, 0, "removed '%s'", finfo->fullname);
  
#endif SERVER_SUPPORT
  if (server->active)
    server_checked_in (finfo->file, finfo->update_dir, finfo->repository);
  
#endif
  free (fname);
} else if (vers-> vn_user[0] == '-')
  { 
    if (quiet)
      error (0, 0, "file '%s' already scheduled for removal",
      finfo->fullname);
  }
else if (vers-> tag != NULL && !isdigit (*vers-> tag))
  { 
    /* Commit will just give an error, and so there seems to be
     * little reason to allow the remove. I mean, conflicts that
     * arise out of parallel development are one thing, but conflicts
     * that arise from sticky tags are quite another.
     
     * I would have thought that non-branch sticky tags should be the
     * same but at least now, removing a file with a non-branch sticky
     * tag means to delete the tag from the file. I'm not sure that
     * is a good behavior, but until it is changed, we need to allow
     * it. */
    error (0, 0, ": cannot remove file " , fname->fullname, vers-> tag);
  }
else
  { 
    char *fname;
    /* Re-register it with a negative version number. */
    fname = xmalloc (strlen (vers-> vn_user) + 5);
    (void) strcpy (fname, "-" );
    (void) strcat (fname, vers-> vn_user);
    Register (finfo->entries, finfo->file, fname, vers-> tags, vers-> options,
         vers-> tag, vers-> date, vers-> tconflict, CVSroot, directory, finfo->repository);
    if (quiet)
      error (0, 0, "scheduling " , fname->fullname);
    removed_files++; 
    
#endif SERVER_SUPPORT
  if (server->active)
    server_checked_in (finfo->file, finfo->update_dir, finfo->repository);
  
#endif
  free (fname);
  
freevers ( &vers);
  return (0);
}
```c
char *update_dir;
List *entries;
{
  if (!quiet)
    error (0, 0, "Removing %s", update_dir);
  return (R_PROCESS);
}
```
A.50 repos.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include <assert.h>
#include "cvs.h"
#include "getline.h"

/** Determine the name of the RCS repository for directory DIR in the
 * current working directory, or for the current working directory
 * itself if DIR is NULL.  Returns the name in a newly-malloc'd
 * string.  On error, gives a fatal error and does not return.
 * UPDATE_DIR is the path from where cvs was invoked (for use in error
 * messages), and should contain DIR as its last component.
 * UPDATE_DIR can be NULL to signify the directory in which cvs was
 * invoked. */

char *
Name_Repository (dir, update_dir)
char *dir;
char *update_dir;
{
    FILE *fpin;
    char *update_dir;
    char repos = NULL;
    size_t repo_allocated = 0;
    char *tmp;
    char *tmp2;
    char *cp;
    if (update_dir && *update_dir)
        update_dir = update_dir;
    else
        update_dir = ".*;"

if (dir != NULL)
{
    tmp = xmalloc (strlen (dir) + sizeof (CVSADM_REP) + 10);
    (void) sprintf (tmp, "%s/%s", dir, CVSADM_REP);
}
else
    tmp = xstrdup (CVSADM_REP);

if (dir != NULL) {
    tmp2 = xmalloc (strlen (dir) + sizeof (CVSADM_REP_REMOTE) + 10);
    (void) sprintf (tmp2, "%s/%s", dir, CVSADM_REP_REMOTE);
} else {
    tmp2 = xstrdup (CVSADM_REP_REMOTE);
}

/* We try to read remote repository first, if we are handling remotes */
if (handling_remotes) {
    fpin = CVS_FOPEN (tmp2, "r");
    if (fpin != NULL) {
        getline (repos, &repos_allocated, fpin);
        fclose (fpin);
    }
}

if (repos_allocated == 0) {
    /* The assumption here is that the repository is always contained in the
     * first line of the "Repository" file.
     */

fpin = CVS_FOPEN (tmp, "r");

    if (fpin == NULL)
    {
        int save_errno = errno;
        char *cvsadm;
        if (dir != NULL)
        {
            cvsadm = xmalloc (strlen (dir) + sizeof (CVSADM) + 10);
            (void) sprintf (cvsadm, "%s/%s", dir, CVSADM);
        }
        else
            cvsadm = xstrdup (CVSADM);

        if (isdir (cvsadm))
        {
            error (0, 0, "in directory %s", update_dir);
            error (1, 0, "there is no version here; do 'cvs checkout' first").
        }
    }
}
program name);
}
free (cvsadm);

if (errno)
{
    /* FIXME: This is a very poorly worded error message. It
    occurs at least in the case where the user manually
    creates a directory named CVS, so the error message
    should be more along the lines of "CVS directory found
    without administrative files; use CVS to create the CVS
    directory, or rename it to something else if the
    intention is to store something besides CVS
    administrative files". */
    error (0, 0, "in directory %s", xupdate_dir);
    error (1, 0, "*PANIC* administration files missing");
}
error (1, save_errno, "cannot open %s", tmp);
}

if (getline (&repos, &reposallocated, fpin) < 0)
{
    /* FIXME: should be checking for end of file separately. */
    error (0, 0, "in directory %s", xupdate_dir);
    error (1, errno, "cannot read %s", CVSADM_REP);
}
if (fchdir (fpin) < 0)
error (0, errno, "cannot close %s", tmp);
free (tmp);
}

if (!cp || strrchr (repos, '\n')) != NULL)
    *cp = '\0';
    /* strip the newline */

/*
 * If this is a relative repository pathname, turn it into an absolute
 * one by tacking on the CVSROOT environment variable. If the CVSROOT
 * environment variable is not set, die now.
 * *
 * If [strmcp (repos, ".") == 0 || strmcp (repos, ".*/", 3) == 0)
 *     error (0, 0, "in directory %s", xupdate_dir);
 *     error (0, 0, ".", "/relative repositories are not supported.");
 *     error (1, 0, "illegal source repository");
 */
if (!isabsolute(repos))
{
    char *newrepos;

    if (CVSroot_original == NULL)
{
        error (0, 0, "in directory %s", xupdate_dir);
        error (0, 0, "must set the CVSROOT environment variable\n");
        error (0, 0, "or specify the -d option to %s", program_name);
        error (1, 0, "illegal repository setting");
    }
    newrepos = malloc (strlen (CVSroot_original) + strlen (repos) + 10);
    (void) sprintf (newrepos, "%s/%s", CVSroot_original, repos);
    free (repos);
    repos = newrepos;
}
Sanitize_Repository (repos);

return repos;
}

/* Return a pointer to the repository name relative to CVSROOT from a
 * possibly fully qualified repository
 */
char *Short_Repository (repository)
char *repository;
{
    if (repository == NULL)
        return (NULL);

    /* If repository matches CVSroot at the beginning, strip off CVSroot */
    /* And skip leading '/' in rep, in case CVSroot ended with '/'. */
    if (strncmp (CVSroot_original, repository, strlen (CVSroot_original)) == 0)
    {
        char *rep = repository + strlen (CVSroot_original);
        return (*rep == '/' ? rep+1 : rep);
    }
else
    return (repository);
Sanitize the repository name (in place) by removing trailing slashes and a trailing "." if present. It should be safe for callers to use strcat and friends to create repository names. 
Without this check, names like "/path/to/repos/.foo" and "//path/to/repos/.foo" would be created. For example, one significant case is the CVSROOT-detection code in commit.c. It decides whether or not it needs to rebuild the administrative file 
database by doing a string compare. If we've done a 'cvs co . ' to 
get the CVSROOT files, "/path/to/repos/.CVSROOT" and 
"//path/to/repos/.CVSROOT" are the arguments that are compared!

This function ends up being called from the same places as strip_path, though what it does is much more conservative. Many comments about this operation (which was scattered around in several places in the source code) ran thus:

"repository ends with "/."; omit it. This sort of thing used to be taken care of by strip_path. Now we try to be more selective. I suspect that it would be even better to push it back further someday, so that the trailing "/." doesn't get into repository in the first place, but we haven't taken things that far yet." –Jim Kingdom (recurse.c, 07-Sep-97)

Ahh, all too true. The major consideration is RELATIVE_REPOS. If the "/." doesn't end up in the repository while RELATIVE_REPOS is defined, there will be nothing in the CVS/Repository file. I haven't verified that the remote protocol will handle that correctly yet, so I've not made that change. /*

```
void SanitizeRepositoryName (repository)
    char *repository;
{
    size_t len;
    assert (repository != NULL);

    strip_trailing_slashes (repository);

    len = strlen (repository);

    if (len >= 2 && repository[len - 1] == '.' && ISDIRSEP (repository[len - 2]))
    {
        repository[len - 2] = '\0';
    }
}
```
A.51  root.c

/*
 * Copyright (c) 1992, Mark D. Baushke
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Name of Root
 * Determine the path to the CVSROOT and set "Root" accordingly.
 * If this looks like a modified clone of NameRepository() in repos.c, it is...
 */

#include "cvs.h"
#include "getline.h"

/*
 * Printable names for things in the CVS root
 * method enum variable.
 * Watch out if the enum is changed in cvs.h!
 */

#define method_names[] = {
    "local", "server (rsh)", "pserver", "kserver", "gserver", "ext"
};

ifndef DEBUG
char *NameRoot (dir, update dir)
char *dir;
char *update dir;
{
    FILE *fpin;
    char *ret, *update dir;
    char root = NULL;
    size_t root_allocated = 0;
    char *tmp;
    char cvsadm;
    char *cp;

    if (update dir && *update dir)
        update dir = update dir;
    else
        update dir = ".";

    if ((dir != NULL)
        && xmalloc (strlen (dir) + sizeof (CVSADM) + 10);
    (void) sprintf (cvsadm, "%s/%s", dir, CVSADM);
    tmp = xmalloc (strlen (dir) + sizeof (CVSADM_ROOT) + 10);
    (void) sprintf (tmp, "%s/%s", dir, CVSADM_ROOT);)
    }
else
    {  
        cvsadm = xstrdup (CVSADM);
        tmp = xstrdup (CVSADM_ROOT);
    }

    /*
    * Do not bother looking for a readable file if there is no cvsadm
    * directory present.
    */
    /*
    * It is possible that not all repositories will have a CVS/Root
    * file. This is ok, but the user will need to specify -d
    */
    if (isdir (cvsadm)) || (!isreadable (tmp))
    {  
        ret = NULL;
        goto out;
    }

    /*
    * The assumption here is that the CVS Root is always contained in the
    * first line of the "Root" file.
    */
    fpin = open file (tmp, "r");
    if (getline (&root, &root_allocated, fpin) < 0)
    {  
        /* FIXME: should be checking for end of file separately; errno
        * is not set in that case. */
        error (0, 0, "in directory ", update dir);
        error (0, errno, "cannot read ", CVSADM_ROOT);
        error (0, 0, "please correct this problem");
        ret = NULL;
        goto out;
    }
    (void) fclose (fpin);
    if ((cp = strrchr (root, '\n')) != NULL)
    {  
        
out:  
        
    }
/* cp = "\0"; /* strip the newline */

/*
 * root now contains a candidate for CVSSpec. It must be an
 * absolute pathname or specify a remote server.
 */

if (client

*defined CLIENT_SUPPORT

(!isabsolute(root))

{
    error(0, 0, "in directory%s:",
        xupdate(dir);
    error(0, 0, "ignoring %s because it does not contain an absolute pathname.",
        CVSADM_ROOT);
    ret = NULL;
    goto out;
}

#define CLIENT_SUPPORT

if ((strchr(root, ':') == NULL) && !isdir(root))

{
    error(0, 0, "in directory%s:",
        xupdate(dir);
    error(0, 0, "ignoring %s because it specifies a non-existent repository%s",
        CVSADM_ROOT, root);
    ret = NULL;
    goto out;
}

*/ allocate space to return and fill it in */

strip(trailing slashes (root));

ret = xstrdup(root);

out: free(cvsadm);

free(tmp);

if (root != NULL)

    free(root);

return (ret);

);

/*
 * Write the CVS/Root file so that the environment variable CVSROOT
 * and/or the -d option to cvs will be validated or not necessary for
 * future work.
 */

void Create_Root (dir, rootdir)

char *dir;

char *rootdir;

{ FILE *fout;
    char *tmp;
    if (noexec)
        return;

    if (isdir(root))

    { /* record the current cvsspec root */
        if (rootdir != NULL)

            { if (dir != NULL)
                    { tmp = xmalloc(strlen(dir) + sizeof(CVSADM_ROOT) + 10);
                        (void) sprintf(tmp, "%s%s", dir, CVSADM_ROOT);
                    } else
                        tmp = xstrdup(CVSADM_ROOT);
                fout = open_file(tmp, "w");
                if (fprintf(fout, "%s", rootdir) < 0)
                    error(1, errno, "write to %s failed", tmp);
                if (fclose(fout) == EOF)
                    error(1, errno, "cannot close %s", tmp);
                free(tmp);
            } else
                tmp = xstrdup(CVSADM_ROOT);
                fout = open_file(tmp, "w");
                if (fprintf(fout, "%s", rootdir) < 0)
                    error(1, errno, "write to %s failed", tmp);
                if (fclose(fout) == EOF)
                    error(1, errno, "cannot close %s", tmp);
                free(tmp);
            } /* record the current cvsspec root */
        }
static unsigned int root_allow_count;
static char *root_allow_vector;
static unsigned int root_allow_size;

void
root_allow_add (arg)
char *arg;
{
char *p;
if (root_allow_size <= root_allow_count)
{
if (root_allow_size == 0)
{
root_allow_size = 1;
root_allow_vector =
(char **) malloc (root_allow_size + sizeof (char *));
}
else
{
root_allow_size += 2;
root_allow_vector =
(char **) realloc (root_allow_vector,
root_allow_size + sizeof (char *));
}
if (root_allow_vector == NULL)
{
no_memory:
/* Strictly speaking, we're not supposed to output anything
now. But we're about to exit(), give it a try. */
210
printf ("* Fatal server error, aborting.").n"
errorENOMEM Virtual memory exhausted.
*/

if (root_allow_vector != NULL)
{
no_memory:
/* I'm doing this manually rather than via error_exit ()
because I'm not sure whether we want to call server_cleanup.
Needs more investigation. */
elseif (i < root_allow_count)
strncpy (p, arg);
root_allow_vector[root_allow_count+i] = p;
}
void
root_allow_free ()
{
if (root_allow_vector != NULL)
free (root_allow_vector);
root_allow_count = 0;
}

int
root_allow_get (arg)
char *arg;
{
unsigned int i;
if (root_allow_count == 0)
{
/* Probably someone upgraded from CVS before 1.9.10 to 1.9.10
 or later without reading the documentation about
 -allow-root. Printing an error here doesn't disclose any
 particularly useful information to an attacker because a
 CVS server configured in this way won't let *anyone* in. */

/* Note that we are called from a context where we can spit
 back "error" rather than waiting for the next request which
 expects responses. */
260
printf ("*
Server configuration missing --allow-root in inetd.conf.
error_exit ();
}

for (i = 0; i < root_allow_count; ++i)
if (strcmp (root_allow_vector[i], arg) == 0)
return 1;
return 0;
#include <stdio.h>
#include <string.h>
#include <err.h>

int parse_cvsroot (CVSroot)
{
    char *CVSroot = NULL; /* the CVSroot that was passed in */
    int client_active; /* nonzero if we are doing remote access */
    CVSmethod CVSroot_method; /* one of the enum values defined in cvs.h */
    char *CVSroot_username; /* the username or NULL if method == local */
    char *CVSroot_hostname; /* the hostname or NULL if method == local */
    char *CVSroot_directory; /* the directory name */

    if (Pserver_Repos == NULL) & & (CVSroot_directory != NULL))
        if (strcmp (Pserver_Repos, CVSroot_directory) != 0)
            error (1, 0, "repository mismatch: \"%s\" vs \"%s\"",
                   Pserver_Repos, CVSroot_directory);

    CVSroot_original = xstrdup (CVSroot);
    CVSroot_copy = xstrdup (CVSroot);

    if ((xstrdup_copy == '.')
        { char *method = ++xstrdup_copy;
            /* Access method specified, as in
             * "cvs -d:pserver:username:hostname:/path",
             * "cvs -d:local:/path", or
             * "cvs -d:kserver:username:hostname:/path".
             * We need to get past that part of CVSroot before parsing the
             * rest of it. */
            if (1 (p = strchr (method, '.')))
                { error (0, 0, "bad CVSroot: \"%s\", CVSroot); return 1;
                }
            p = '0';
            xstrdup_copy = ++p;
            /* Now we have an access method – see if it's valid. */
            if (strcmp (method, "local") == 0)
                CVSroot_method = local_method;
            else if (strcmp (method, "pserver") == 0)
                CVSroot_method = pserver_method;
else if (strcmp (method, "kserver") == 0)
    CVSroot_method = kserver_method;
else if (strcmp (method, "gserver") == 0)
    CVSroot_method = gserver_method;
else if (strcmp (method, "server") == 0)
    CVSroot_method = server_method;
else if (strcmp (method, "ext") == 0)
    CVSroot_method = ext_method;
else
{
    error (0, 0, "unknown method in CVSroot: %a", CVSroot);
    return 1;
}
else
{
    /* If the method isn’t specified, assume
     * SERVER_METHOD/EXT_METHOD if the string contains a colon or
     * LOCAL_METHOD otherwise. */
    CVSroot_method = ((strchr (cvsroot_copy, ':'))

sifdef RSH_NOT_TRANSPARENT
    ? server_method
else
    ? ext_method
sendif
    : local_method);
}
client_active = (CVSroot_method != local_method);

/* Check for username/hostname if we’re not LOCAL_METHOD. */
if (CVSroot_username == NULL;
CVSroot_hostname == NULL;
if (CVSroot_method != local_method)
{
    /* Check to see if there is a username in the string. */
    if ((p = strchr (cvsroot_copy, ‘@’)))
    {
        CVSroot_username = cvsroot_copy;
        sp = ‘@’;
        cvsroot_copy = ++p;
        if ((CVSroot_username == \’0\’)
            CVSroot_username = NULL;
    }
    if ((p = strchr (cvsroot_copy, ‘.’)))
    {
        CVSroot_hostname = cvsroot_copy;
        sp = ‘.’;
        cvsroot_copy = ++p;
        if ((CVSroot_hostname == \’0\’)
            CVSroot_hostname = NULL;
    }
}
CVSroot_directory = cvsroot_copy;
sifdef AUTH_SERVER_SUPPORT
check_method_consistent ();
sendif /* AUTH_SERVER_SUPPORT */
sif ! defined (CLIENT_SUPPORT) && ! defined (DEBUG)
if (CVSroot_method != local_method)
{
    error (0, 0, "Your CVSROOT is set for a remote access method");
    error (0, 0, "but your CVS executable doesn’t support it");
    error (0, 0, "%a", CVSroot);
    return 1;
}
sendif /* Do various sanity checks. */
if (CVSroot_username && ! CVSroot_hostname)
{
    error (0, 0, "missing hostname in CVSROOT: %a", CVSroot);
    return 1;
}
check_hostname = 0;
switch (CVSroot_method)
{
    case local_method:
    if (CVSroot_username || CVSroot_hostname)
    {
        error (0, 0, "can’t specify hostname and username in CVSROOT");
    }
error (0, 0, "when using local access method");
error (0, 0, "(Xa)", CVSroot);
returns 1;
} /* cvs.texinfo has always told people that CVSROOT must be an
absolute pathname. Furthermore, attempts to use a relative
pathname produced various errors (I couldn't get it to work),
so there would seem to be little risk in making this a fatal
error. */
if ([isabsolute ([CVSroot|directory])]
error (1, 0, "CVSROOT %s must be an absolute pathname"
CVSroot|directory);
break;
case kserver_method:
#endif
if ([HAVE|KERBEROS]
error (0, 0, "Your CVSROOT is set for a kerberos access method");
error (0, 0, "but your CVS executable doesn't support it");
error (0, 0, "(Xa)", CVSroot);
returns 1;
#endif
check_hostname = 1;
break;
case server_method:
case ext_method:
case pserver_method:
check_hostname = 1;
break;
} if ([check_hostname])
{ if (! [CVSroot|hostname])
{ error (0, 0, "didn't specify hostname in CVSROOT: %s", CVSroot);
returns 1;
}
if ([CVSroot|directory] == '\0')
{ error (0, 0, "missing directory in CVSROOT: %s", CVSroot);
returns 1;
} /* Hooray! We finally parsed it! */
returns 0;
} /* Set up the global CVSroot* variables as if we're using the local
repository DIR. */

void
set_local_cvsroot (dir)
char *dir;
{ CVSroot_original = xstrdup (dir);
CVSroot_method = local_method;
CVSroot_directory = CVSrootoriginal;
#endif
/* AUTH_SERVER_SUPPORT */
check_root_consistent ();
#endif /* AUTH_SERVER_SUPPORT */
CVSroot_username = NULL;
CVSroot_hostname = NULL;
client_active = 0;
}
#endif
/* DEBUG */
/* This is for testing the parsing function. */
#include <stdio.h>
char *CVSroot;
char *program_name = "testing";
char *command_name = "parse_cvsroot"; /* XXX is this used?? */

void
main (argc, argv)
int argc;
char *argv[];
```c
540  program_name = argv[0];
      
      if (argc != 2)
      {
        fprintf (stderr, "Usage: %s <CVSROOT>\n", program_name);
        exit (2);
      }

      if (parse_cvsroot (argv[1]))
      {
        fprintf (stderr, "%s: Parsing failed.\n", program_name);
        exit (1);
      }

      printf ("CVSroot: %s\n", argv[1]);
      printf ("CVSroot_method: %s\n", method_names[CVSroot_method]);
      printf ("CVSroot_username?: %s\n", CVSroot_username ? CVSroot_username : "NULL");
      printf ("CVSroot_hostname?: %s\n", CVSroot_hostname ? CVSroot_hostname : "NULL");
      printf ("CVSroot_directory?: %s\n", CVSroot_directory);

      exit (0);
      /* NOTREACHED */
    } /* endif */
```
A.52 rtag.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Rtag
 * Add or delete a symbolic name to an RCS file, or a collection of RCS files.
 * Uses the modules database, if necessary.
 */

#include "cvs.h"

static int check_file(procPROTO)(void *callerdat, struct file_info *finfo);
static int check_file_nodes(procPROTO)(void *callerdat, int err,
   char *repos, char *update_dir,
   List *entries);
static int pretag_dir(procPROTO)(char *repository, char *filter);
static void masterlist_dir(procPROTO)(Node *p);
static int pretag_dir(procPROTO)(Node *p, void *closure);
static Dtype rtag_dir(procPROTO)(void *callerdat, char *dir,
   char *repos, char *update_dir,
   List *entries);
static int rtag_dir(procPROTO)(int *pargc, char **argv, char *xwhere,
   char *xwhere, char *rfile, int shorten,
   int local_specified, char *rname, char *msg);
static int rtag_delete(procPROTO)(RCSNode *rcsfile);

struct tag_info
{
  Ctype status;
  char *rev;
  char *tag;
  char *options;
};

struct master_lists
{
  List *ntlist;
};

static List *ntlist;
static List *ntlist;

static char *syntax_tag;
static char *syntax_tag;
static int numtag validated = 0;
static int delete_tag;
  /* adding a tag by default */
static int attic_tag;
  /* remove tag from Attic files */
static int branch_mode;
  /* make an automatic "branch" tag */
static char *date;
static int local;
  /* recursive by default */
static int force_tag_match = 1;
  /* force by default */
static int force_tag_move;
  /* don't move existing tags by default */

static const char *const rtag_usage[] =
{
  "Usage: rtag [-f] [-u] [-d] [-v] [-r] [tag] module... \n",
  "\t-v\tVerbose, print extra info\n",
  "\t-f\tForce a hard revision match if tag/date not found\n",
  "\t-u\tLocal directory only, not recursive\n",
  "\t-d\tDirectories recursively, \n",
  "\t-b\tBranch tag, allowing concurrent development\n",
  "\t-n\tNo execution of 'tag program'\n",
  "\t-R\tProcess directories recursively\n",
  "\t-F\tForce move if it already exists\n",
  "\t-a\tClear tag from removed files that would not otherwise be tagged\n",
  "\t-b\tMake the tag a "branch" tag, allowing concurrent development\n",
  "\t-d\tDelete the given tag\n",
  "\t-D\tExisting date\n",
  "\t-F\tMove tag if it already exists\n",
  "(Specify the --help global option for a list of other help options)\n",
  NULL,
};

int rtag (argv, argc)
int argc;
char **argv;
{
  register int i;
  int c;

...
DBM *ib;
int run_module_prog = 1;
int err = 0;
if (argc == -1)
  usage (rtag_usage);
optind = 0;
while ((c = getopt (argc, argv, "+FnlTQqrdbr:D:")) != -1)
  {
    switch (c)
    {
      case 'a':
        attic_mode = 1;
        break;
      case 'n':
        run_module_prog = 0;
        break;
      case 'Q':
        case 'q':
          if (!server_active)
            error (1, 0,
            "-q or -Q must be specified before \"%s\",
            command_name);
        break;
      case 'l':
        local = 1;
        break;
      case 'R':
        local = 0;
        break;
      case 'd':
        delete_flag = 1;
        break;
      case 'f':
        force_tag_match = 0;
        break;
      case 'b':
        if (date)
          error (1, 0,
          "warning: -b ignored with -d options");
        break;
      case 'r':
        numtag = optarg;
        break;
      case 'D':
        if (date)
          free (date);
        date = Make_Date (optarg);
        break;
      case '?'
        default:
          usage (rtag_usage);
        break;
    }
  }
  argc -= optind;
  argv += optind;
if (argc < 2)
  usage (rtag_usage);
symtag = argv[0];
argc--;
argv++;
if (date && numtag)
  error (1, 0, "-r and -D options are mutually exclusive");
if (delete_flag && branch_mode)
  error (0, 0, "warning: -b ignored with -d options");
RCS_check_tag (symtag);
#endif SERVER_SUPPORT
#ifdef CLIENT_SUPPORT
  if (client_active)
    {
      /* We're the client side. Fire up the remote server. */
      start_server ();
      ign_setup ();
      if (force_tag_match)
        sendarg ("-f");
      if (local)
        sendarg("-l");
      if (delete_flag)
        sendarg("-d");
      if (branch_mode)
sendarg("-b");
if (force_tag_move)
  sendarg("-m");
if (runmodule argc)
  sendarg("-n");
if (attic too)
  sendarg("-f");
if (numtag)
  optionwitharg (*r*, numtag);
if (date)
  client senddate (date);
sendarg (symtag);
{
  int i;
  for (i = 0; i < argc + 1)
    sendarg (argv[i]);
}
return get_responses_and_close ();
}
#endif

db = open_module ();
for (i = 0; i < argc + 1)
{
  /* XXX last arg should be repository, but doesn’t make sense here */
  history write ("T", (delete_flag ? "D" : (numtag ? numtag :
  (date ? date : "T")), symtag, argv[i], ");
err += do_module (db, argv[i], TAG,
  delete_flag ? "Untagging" : "Tagging",
  rtagproc, (char *) NULL, 0, runmodule argc, symtag);
}
close_module (db);
return (err);
}

/*
 * callback proc for doing the real work of tagging
 * /
 * ARGUSED */
static int
rtagproc (argv, argc, xwhere, mwhere, mfile, shorten, local_specified,
  mname, mag)
  
  int *argv;
  char **argv;
  char *xwhere;
  char *mwhere;
  char *mfile;
  int shorten;
  int local_specified;
  char *mname;
  char *mag;
{
  /* Begin section which is identical to patchproc—should this
   * be abstracted out somehow? */
  int err = 0;
  int which;
  char *repository;
  char *where;
  repository = xmalloc (strlen (CVSroot directory) + strlen (argv[0])
    + (mfile == NULL ? 0 : strlen (mfile)) + 10);
  (void) sprintf (repository, "%s", CVSroot directory, argv[0]);
  where = xmalloc (strlen (argv[0]) + (mfile == NULL ? 0 : strlen (mfile))
    + 10);
  (void) strcpy (where, argv[0]);
  /* if mfile isn’t null, we need to set up to do only part of the module */
  if (mfile != NULL)
  {
    char *cp;
    char *path;
    /* if the portion of the module is a path, put the dir part on repos */
  if (cp = strchr (mfile, ’/‘)) != NULL)
  {
    *cp = ’/’;
    (void) strcat (repository, "/");
    (void) strcat (repository, mfile);
    (void) strcat (where, "/");
    mfile = cp + 1;
  }
  /* take care of the rest */
path = xmalloc(strlen(repository) + strlen(mfile) + 5);
(void) sprintf(path, "%s/%s", repository, mfile);
if (!isdir(path))
{  /* directory means repository gets the dir tacked on */
    (void) strcpy(repository, path);
    (void) strcat(where, "/");
    (void) strcat(where, mfile);
}
else
{  int i;
    /* a file means muck argv */
    for (i = 1; i < *pargc; i++)
    {  free(argv[i]);
        argv[i] = xstrdup(mfile);
        (*pargc) = 2;
    }
    free(path);
}
/* cd to the starting repository */
if (CVS_CHDIR(repository) < 0)
{  error(0, errno, "cannot chdir to %s", repository);
    free(repository);
    return(1);
}
else
{  /* End section which is identical to patch_proc. */
    if ((delete_flag || attic_too || (force_tag_match & & numtag))
        which = W_REPOS | W_ATTIC;
    else
        which = W_REPOS;
    if (numtag != NULL & & !numtag_validated)
    {  tagcheckvalid(numtag, *pargc = 1, argv + 1, local, 0, NULL);
        numtag_validated = 1;
    }
    /* check to make sure they are authorized to tag all the
called files in the repository */
    mtlist = getlist();
    err = startrecursion(check_fileproc, check_filesdoneproc,
        (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
        *pargc = 1, argv + 1, local, which, 0, 1,
        where, 1);
    if (err)
    {  error(1, 0, "correct the above errors first!");
    }
    /* start the recursion processor */
    err = startrecursion(rtag_fileproc, rtag_filesdoneproc, rtag_dirproc,
        (DIRLEAVEPROC) NULL, NULL,
        *pargc = 1, argv + 1, local,
        where, 0, 1);
    free(where);
    dellist(&mtlist);
    return(err);
}
/* check file that is to be tagged */
/* All we do here is add it to our list */
static int
check_fileproc(callerdat, finfo)
void *callerdat;
struct fileinfo *finfo;
{  char *xdir;
    Node *p;
    Version *vers;
    if (finfo->update_dir[0] == '0')
        xdir = ".";
    else
        xdir = finfo->update_dir;
    if ((p = findnode (mtlist, xdir)) != NULL)
    {  tlist = ((struct master_lists *) p->data)->tlist;
    }
    else
    {  
      xmalloc(strlen(repository) + strlen(mfile) + 5);
      free(repository);
      return(1);
    }
}
struct master_lists *ml;

tlist = getlist();
p = getnode();
p->key = xstrdup(xdir);
p->type = UPDATE;
ml = (struct master_lists *) smallloc((sizeof(struct master_lists)));
ml->tlist = tlist;
p->data = (char *) ml;
p->delproc = masterlist_delproc;

(void) addnode (mtlist, p);
/* do that */
p = getnode();
p->key = xstrdup (info->file);
p->type = UPDATE;
p->delproc = tag_delproc;
vers = VersionTS (info, NULL, NULL, NULL, 0, 0);
p->data = RCS_getversion (vers->arcfile, numtag, date, force_tag_match, (int *) NULL);

if (p->data != NULL)
{
    int addit = 1;
    char soversion;
    oversion = RCS_getversion (vers->arcfile, sysname, (char *) NULL, 1, (int *) NULL);
    if (oversion == NULL)
    {
        if (delete_tag)
        {
            addit = 0;
        }
        else if (strcmp(oversion, p->data) == 0)
        {
            addit = 0;
        }
        else if (!force_tag_move)
        {
            addit = 0;
        }
    }
    if (oversion != NULL)
    {
        free(oversion);
    }
    if (addit)
    {
        free(p->data);
        p->data = NULL;
    }
}
freevers (vers);
(void) addnode (tlist, p);
return (0);

static int
check_tagdoneproc (callerdat, err, repos, update very, entries)
    void *callerdat;
420 int err;

    char *repos;
    char *update very;
    List *entries;
    {
        int n;
        Node *p;
        p = findnode (mtlist, update very);
        if (p != NULL)
        {
            tlist = ((struct master_lists *) p->data)->tlist;
        }
        else
        {
            tlist = (List *) NULL;
        }
        if (!tlist || (tlist->list->next == tlist->list))
        {
            return (err);
        }
}

if (n = Parse_Info (CVSROOTADM_TAGINFO, repos, pretag_proc, 1)) > 0)
{
    error (0, 0, "Pre-tag check failed");
    err += n;
}
return (err);
static int
pretag_proc(repository, filter)
char *repository;
char *filter;
{
    if (filter[0] == '/')
    {
        char *s, *cp;
        s = xstrdup(filter);
        for (cp=s, *cp; cp++)
        {
            if (isspace(*cp))
            {
                *cp = '\0';
                break;
            }
        }
        if (!isfile(s))
        {
            error (0, errno, "cannot find pre-tag filter '%s'", s);
        free(s);
        return (1);
        }
        free(s);
run_setup(filter);
run_arg(symtag);
runcall(delete_flag ? "del" : force_tag ? "mov" : "add");
runcall(repository);
walklist(tlist, pretaglistproc, NULL);
return (run_exec(RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL));
    }
}

static void
masterlist_delproc(p)
Node *p;
{
struct masterlists *ml;
ml = (struct masterlists *)p->data;
dellist(&ml->tlist);
free(ml);
return;
}

static void
tag_delproc(p)
Node *p;
{
    if (p->data != NULL)
    {
        free(p->data);
p->data = NULL;
    }
return;
}

static int
pretag_listproc(p, closure)
Node *p;
void *closure;
{
    if (p->data != NULL)
    {
        runarg(p->key);
runcall(p->data);
    }
return (0);
}

/ *
* Called to tag a particular file, as appropriate with the options that were
* set above.
* /
/ * ARGUSED */
static int
rtag_fileproc(callerdat, finfo)
void *callerdat;
struct fileinfo *finfo;
{
RCSNode *rcsfile;
char *version, *rev;
int retcode = 0;

/ * Lock the directory if it is not already locked. We might be
able to rely on rtag_dirproc for this. */

/ * It would be nice to provide consistency with respect to
commits; however CVS lacks the infrastructure to do that (see

Concurrence in cvs.tiexinfo and comment in do_{\textbf{recursive}}. We can and will prevent simultaneous tag operations from interfering with each other, by write locking each directory as we enter it, and unlocking it as we leave it. */
lockfile WRITE (INFO->repository);

/* find the parsed RCS data */
if ((rcsfile = INFO->rcs) == NULL)
  return (1);

/* For tagging an RCS file which is a symbolic link, you’d beat be
  running with RCS 5.6, since it knows how to handle symbolic links
  correctly without breaking your link! */
if (deleteflag)
  return (rtag_delete (rcsfile));

/* If we get here, we are adding a tag. But, if -a was specified, we
  need to check to see if a -r or -D option was specified. If neither
  was specified and the file is in the Attic, remove the tag. */
if (((attic == 0) && (numtag == 0))
    && (rcfile->flags & VALID)
    && (rcfile->flags & INATTIC))
  return (rtag_write (rcsfile));

version = RCS_getversion (rcsfile, numtag, date, force_tag_match, (int *) NULL);
if ((version == NULL)
  /* If -a specified, clean up any old tags */
  if (attic == 0)
    (void) rtag_delete (rcsfile);
  if (quiet && force_tag_match)
    {
      error (0, 0, "cannot find tag '\"%s\"' in '\"%s\"",
             numtag ? numtag : "head", rcsfile->path);
      return (1);
    }
  return (0);
}
if (numtag && isdigit (+numtag) && strcmp (numtag, version) != 0)
{
  /* We didn’t find a match for the numeric tag that was specified, but
   * that’s OK - just pass the numeric tag on to rcs, to be tagged as
   * specified. Could get here if one tried to tag "1.1.1" and there
   * was a 1.1.1 branch with some head revision. In this case, we want
   * the tag to reference "1.1.1" and not the revision at the head of
   * the branch. Use a symbolic tag for that. */
  if ((rcsfile = RCS_magicrev (rcfile, version)) == numtag)
    rtag_write (rcsfile, NULL, NULL);
else
  {
    char version;
    /* As an enhancement for the case where a tag is being re-applied to
     * a large body of a module, make one extra call to RCS_getversion to
     * see if the tag is already set in the RCS file. If so, check to
     * see if it needs to be moved. If not, do nothing. This will
     * likely save a lot of time when simply moving the tag to the
     * "current" head revisions of a module – which I have found to be a
     * typical tagging operation. */
    if ((rcsfile = RCS_magicrev (rcfile, version)) == version)
      version = RCS_getversion (rcsfile, symtag, numtag);
  }
  if (version != NULL)
    {
      int isbranch = RCS_nodeisbranch (INFO->rcs, symtag);
      /* if versions the same and neither old nor new are branches don’t
       * have to do anything */
      if (strcmp (version, version) == 0 && (branchmode && isbranch))
        {
          free (version);
        }
free (version);
  return (0);
}

if (!force_tag_move)
{  /* we're NOT going to move the tag */
  (void) printf ("%s already exists on %s
", symtag, oversion);
  (void) printf (" : NOT MOVING tag to %s %s
", symtag, oversion);
  free (version);
  free (oversion);
  return (0);
}
free (oversion);
}
retcode = RCS_settag (rcsfile, symtag, rev);
if (retcode == 0)
  RCS_rewrite (rcsfile, NULL, NULL);
if (retcode != 0)
{  error (1, retcode != -1 ? errno : 0,
      "failed to set tag '%s' to revision '%s' in '%s'",
      symtag, rev, rcsfile->path);
  if (branch_mode)
    free (rev);
    free (version);
    return (1);
}
free (version);
free (branch_mode);
free (version);
return (0);

700 /*
 * If -d is specified, "force_tag_match" is set, so that this call to
 * RCS_getversion() will return a NULL version string if the symbolic
 * tag does not exist in the RCS file.
 *  *
 * If the -r flag was used, numtag is set, and we only delete the
 * symtag from files that have numtag.
 *  *
 * This is done here because it's MUCH faster than just blindly calling
 * "rcs" to remove the tag... trust me.
 */

710 static int rtag_delete (rcsfile)
RCSNode *rcsfile;
{
  char *version;
  int retcode;
  if (numtag)
  {  version = RCS_getversion (rcsfile, numtag, (char *) NULL, 1,
       (int *) NULL);
    if (version == NULL)
      return (0);
      free (version);
      }  version = RCS_getversion (rcsfile, symtag, (char *) NULL, 1,
       (int *) NULL);
    if (version == NULL)
      return (0);
      free (version);
    if ((retcode = RCS_deltag(rcsfile, symtag)) != 0)
    {  if (quiet)
      error (0, retcode == -1 ? errno : 0,
            "failed to remove tag '%s' from '%s'", symtag, rcsfile->path);
        return (1);
    }
  RCS_rewrite (rcsfile, NULL, NULL);
  return (0);
}
/* Clear any lock we may have hold on the current directory. */

720 static int rtag_filesdoneproc (callerdat, err, repos, update_dir, entries)
void callerdat;
int err;
char *repos;
char *update_dir;
List *entries;
{
    LockCleanup();
    return (err);
}

/* Print a warm fuzzy message */
/* ARGSUSED */
static Dtype rtag_dirproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
{
    if (ignore_directory (update_dir))
        { /* print the warm fuzzy message */
            if (quiet)
                error (0, 0, "Ignoring %s", update_dir);
            return R_SKIP_ALL;
        }
    if (quiet)
        error (0, 0, "%s %s", delete_flag ? "Untagging" : "Tagging", update_dir);
    return (R_PROCESS);
}
A.53 run.c

/* run.c — routines for executing subprocesses. */

This file is part of GNU CVS.

GNU CVS is free software; you can redistribute it and/or modify it under the
terms of the GNU General Public License as published by the 
Free Software Foundation; either version 2, or (at your option) any 
later version.

This program is distributed in the hope that it will be useful, 
without ANY WARRANTY; without even the implied warranty of 
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the 
GNU General Public License for more details.

*/

#include "cvs.h"

#ifndef HAVE_UNISTD_H
extern int execvp PROTO((char *file, char **argv));
#endif

static void run_addarg PROTO((const char *s));

extern char **run_argv;
static int run(argc); static int run(argc allocated);

/* VARARGS */
void run_setup (prog)
const char **prog
{
    char **cp;
    int i;
    char *run_prog;

    /* clean out any malloc'ed values from run_argv */
    for (i = 0; i < run(argc); i++)
    {
        if (run_argv[i])
        {
            free (run_argv[i]);
            run_argv[i] = (char *) 0;
        }
    }
    run(argc) = 0;

    run_prog = xstrdup (prog);

    /* put each word into run_argv, allocating it as we go */
    for (cp = strtok (run_prog, " "); cp; cp = strtok ((char *) NULL, " "))
        run_addarg (cp);
    free (run_prog);
}

void run_arg (s) const char *s;
{
    run_addarg (s);
}

static void run_addarg (s) const char *s;
{
    /* allocate more argv entries if we've run out */
    if (run(argc) >= run(argc allocated))
    {
        run(argc allocated) += 50;
        run(argc) = (char **) xrealloc ((char **) run(argc),
            run(argc allocated) + sizeof (char **));
    }
    if (s)
run_argv[run_argv+1] = xstrdup(s);
else
    run_argv[run_argv] = (char *) 0; /* not post-incremented on purpose */
}

int
run_exec(stin, stout, sterr, flags)
const char *stin;
const char *stout;
const char *sterr;
int flags;
{
    int shin, shout, sherr;
    int modinout, modead;
    int status;
    int rc = -1;
    int rerrno = 0;
    int pid, w;

define POSIX_SIGNALS
    sigset_t sigset_mask, sigset_umask;

define BSD_SIGNALS
    int mask;
    sigvec vec;
enddef

 ifdef SERVER
    endif

 ifdef BSD
    endif

 endif

 ifdef RETSIGTYPE (+stat) (), (+stat) ()
 enddef

 ifdef SERVER_SUPPORT
    endif

 cwarn(stderr, server ? "S" : *, 1);

 ifdef
    endif

 cwarn(stderr, "system(*, 0);
 run_print(stderr);
 cwarn(stderr, ""?

 if (noexec && (flags & RUN_REALLY) == 0)
    return (0);

 /* make sure that we are null terminated, since we didn't calloc */
 run_argv[0];

 /* setup default file descriptor numbers */
 shin = 0;
 shout = 1;

 sherr = 2;

 /* set the file modes for stdout and stderr */
 mode_out = mode_err = O_WRONLY | O_CREAT,
          mode_out |= (flags & OUTSTDOUTAPPEND) ? O_APPEND : O_TRUNC;
 mode_err  |= (flags & OUTSTDERRAPPEND) ? O_APPEND : O_TRUNC;

 if (stin && (shin = open(stin, O_RDONLY)) == -1)
    rerrno = errno;
 error(0, errnos, "cannot open %s for reading (prog %s)",
    stdin, run_argv[0]);
    goto out0;

 if (stout && (shout = open(stdout, mode_out, 0666)) == -1)
    rerrno = errno;
 error(0, errnos, "cannot open %s for writing (prog %s)",
    stdout, run_argv[0]);
    goto out1;

 if (sterr && (flags & RUN_COMBINED) == 0)
    if ((sherr = open(stderr, mode_err, 0666)) == -1)
        rerrno = errno;
 error(0, errnos, "cannot open %s for writing (prog %s)",
    stderr, run_argv[0]);
        goto out2;

 /* Make sure we don't flush this twice, once in the subprocess. */
 flush(stdout);
 flush(stderr);

 /* The output files, if any, are now created. Do the fork and dup's.

 We use vfork not so much for the sake of unices without
copy-on-write (such systems are rare these days), but for the sake of systems without an MMU, which therefore can't do copy-on-write (e.g. Amiga). The other solution is spawn (see windows-NT/run.c). */

 ifndef HAVE_VFORK
 pid = vfork ();
 else
 pid = fork ();
 endif
 if (pid == 0)
 { 
 if (shin != 0)
 { 
 (void) dup2 (shin, 0);
 (void) close (shin);
 }
 if (shout != 1)
 { 
 (void) dup2 (shout, 1);
 (void) close (shout);
 }
 
 if (flags & RUN_COMBINED)
 (void) dup2 (1, 2);
 else if (sherr != 2)
 { 
 (void) dup2 (sherr, 2);
 (void) close (sherr);
 }
 
 /* dup'ing is done. try to run it now */
 (void) execvp (run_argv[0], run_argv);
 error (0, errno, "cannot exec %s", run_argv[0]);
 _exit (127);
 }
 else if (pid == -1)
 { 
 rerrno = errno;
 goto out;
 }
 /* the parent. Ignore some signals for now */

 ifndef POSIX_SIGNALS
 if (flags & RUN_SIGIGNORE)
 { 
 act.sa_handler = SIG_IGN;
 (void) sigemptyset (&act.sa_mask);
 act.sa_flags = 0;
 (void) sigaction (SIGINT, &act, &iact);
 (void) sigaction (SIGQUIT, &act, &qact);
 }
 else
 { 
 (void) sigemptyset (&sigset_mask);
 (void) sigaddset (&sigset_mask, SIGINT);
 (void) sigaddset (&sigset_mask, SIGQUIT);
 (void) sigprocmask (SIG_SETMASK, &sigset_mask, &sigset_mask);
 }

 else

 ifndef BSD_SIGNALS
 if (flags & RUN_SIGIGNORE)
 { 
 memset ((char *) &vec, 0, sizeof (vec));
 vec.av_handler = SIG_IGN;
 (void) sigvec (SIGINT, &vec, &ivec);
 (void) sigvec (SIGQUIT, &vec, &qvec);
 }
 else
 mask = sigblock (sigmask (SIGINT) | sigmask (SIGQUIT));

 else
 istat = signal (SIGINT, SIG_IGN);
 qstat = signal (SIGQUIT, SIG_IGN);

 endif

 /* wait for our process to die and munge return status */
 ifndef POSIX_SIGNALS
 while ((w = waitpid (pid, &status, 0)) == -1 && errno == EINTR)
 ;

 else
 while ((w = wait (&status)) != pid)
 { 
 if (w == -1 && errno != EINTR)
 break;
 }
 endif

 if (w == -1)
 { 
 rc = -1;
 }
return errno = errno;

#define VMS /* status is return status */
else if (WIFEXITED (status))
  rc = WEXITSTATUS (status);
else if (WIFSIGNALED (status))
  { if (WTERMSIG (status) == SIGPIPE)
    error (1, 0, "broken pipe");
    rc = 2;
  }
else
  rc = 1;
#endif /* VMS */
#endif /* VMS */

/* restore the signals */
#define POSIX_SIGNALS
if (flags & RUN_SIGIGNORE)
  {
    (void) sigaction (SIGINT, &iact, (struct sigaction *) NULL);
    (void) sigaction (SIGQUIT, &iact, (struct sigaction *) NULL);
  }
else
  (void) sigprocmask (SIG_SETMASK, &sigmask, (sigset_t *) NULL);
#endif /* POSIX_SIGNALS */

#define BSD_SIGNALS
if (flags & RUN_SIGIGNORE)
  {
    (void) sigvec (SIGINT, &ivec, (struct sigvec *) NULL);
    (void) sigvec (SIGQUIT, &ivec, (struct sigvec *) NULL);
  }
else
  (void) sigsetmask (mask);
#endif /* BSD_SIGNALS */

/* cleanup the open file descriptors */
out:
  if (stderr) close (stderr);
  out2:
  if (stdout) close (stdout);
  out1:
  if (stdin) close (stdin);
out0:
  if (ferrno)
    errno = errno;
  return (rc);
}

void
runprint (fp)
FILE *fp;

{ int i;
  void (*outfn) PROTO ((const char *, size_t));

    if (fp == stderr)
      outfn = cvwouter;
    else if (fp == stdout)
      outfn = cvwoutput;
    else
      { error (1, 0, "internal error: bad argument to run_print");
        /* Solely to placate gcc -Wall.
           FIXME: it'd be better to use a function named 'fatal' that
           is known never to return. Then kludges wouldn't be necessary. */
        outfn = NULL;
      }
    for (i = 0; i < run_argc; i++)
      { (*outfn) (argv[i], 1);
        (*outfn) (argc, 0);
        (*outfn) (argv[i], 1);
        if (i != run_argc - 1)
          (*outfn) (" ", 1);
      }
  }
/* Return value is NULL for error, or if noexec was set. If there was an
   error, return NULL and I'm not sure whether errno was set (the Red Hat
FILE *
run_popen (cmd, mode)
const char *cmd;
const char *mode;
{
if (trace)
#define SERVER_SUPPORT
(void) fprintf (stderr, "%s\n", run_popen(%s,%s)\n", cmd, mode);
#undef SERVER_SUPPORT
#endif
if (noexec)
    return (NULL);
}

return (popen (cmd, mode));

int
piped_child (command, tofdp, fromfdp)
char ***command;
int *tofdp;
int *fromfdp;
{
    int pid;
    int to_child_pipe[2];
    int from_child_pipe[2];

    if (pipe (to_child_pipe) < 0)
        error (1, errno, "cannot create pipe");
    if (pipe (from_child_pipe) < 0)
        error (1, errno, "cannot create pipe");

#define USE_SETMODE BINARY
setmode (to_child_pipe[0], O_BINARY);
setmode (to_child_pipe[1], O_BINARY);
setmode (from_child_pipe[0], O_BINARY);
setmode (from_child_pipe[1], O_BINARY);
#undef USE_SETMODE

if (vfork () == 0)
    pid = fork ();
else
    pid = fork ();

if (pid < 0)
    error (1, errno, "cannot fork");
else if (pid == 0)
{
    if (dup2 (to_child_pipe[0], STDIN_FILENO) < 0)
        error (1, errno, "cannot dup2");
    if (close (to_child_pipe[1]) < 0)
        error (1, errno, "cannot close");
    if (close (from_child_pipe[0]) < 0)
        error (1, errno, "cannot close");
    if (dup2 (from_child_pipe[1], STDOUT_FILENO) < 0)
        error (1, errno, "cannot dup2");

    execvp (command[0], command);
    error (1, errno, "cannot exec");
}
else
{
    if (close (to_child_pipe[0]) < 0)
        error (1, errno, "cannot close");
    if (close (from_child_pipe[1]) < 0)
        error (1, errno, "cannot close");

    * tofdp = to_child_pipe[1];
    * fromfdp = from_child_pipe[0];

    return pid;
}

void
close_on_exec (fd)
int fd;
{
    if defined (FD_CLOEXEC) & defined (FD_SETFD)
        if (fcntl (fd, F_SETFD, 1))
            error (1, errno, "can't set close-on-exec flag on %d", fd);
}

/*
 * dir = 0: main proc writes to new proc, which writes to oldfd
 * dir = 1: main proc reads from new proc, which reads from oldfd
 *
 * Returns: a file descriptor. On failure (i.e., the exec fails),
* then filter |stream| through |program()| complains and dies.

```c
int
filter_stream_through_program (oldfd, dir, prog, pidp)
{
    int oldfd, dir;
    char **prog;
    pid_t *pidp;
    {
        int p[2], newfd;
        pid_t newpid;
        if (pipe (p))
            error (1, errno, "cannot create pipe");
        ifdef USE_SETMODE_BINARY
            setmode (p[0], O_BINARY);
            setmode (p[1], O_BINARY);
        endif
        ifdef HAVE_VFORK
            newpid = vfork ();
        else
            newpid = fork ();
        endif
        if (pidp)
            *pidp = newpid;
        switch (newpid)
            case -1:
                error (1, errno, "cannot fork");
            case 0:
                /* child */
                if (dir)
                    /* write to new pipe */
                    close (p[0]);
                    dup2 (oldfd, 0);
                    dup2 (p[1], 1);
                }
                else
                    /* read from new pipe */
                    close (p[1]);
                    dup2 (oldfd, 1);
                }
                /* Should I be blocking some signals here? */
                execvp (prog[0], prog);
                error (1, errno, "couldn't exec %s", prog[0]);
            default:
                /* parent */
                close (oldfd);
                if (dir)
                    /* read from new pipe */
                    close (p[1]);
                    newfd = p[0];
                }
                else
                    /* write to new pipe */
                    close (p[0]);
                    newfd = p[1];
                } newfd = exec (newfd);
            return newfd;
        }
    }
```
Trivially encode strings to protect them from innocent eyes (i.e., inadvertent password compromises, like a network administrator who’s watching packets for legitimate reasons and accidentally sees the password protocol go by).

This is NOT secure encryption.

It would be tempting to encode the password according to username and repository, so that the same password would encode to a different string when used with different usernames and/or repositories. However, then users would not be able to cut and paste passwords around. They’re not supposed to anyway, but we all know they will, and there’s no reason to make it harder for them if we’re not trying to provide real security anyway.

Set this to test as a standalone program.

#define DIAGNOSTIC
#ifndef DIAGNOSTIC
#include "cvs.h"
#else
!DIAGNOSTIC
#endif
// cvs.h won't define this for us
#define AUTH_CLIENT_SUPPORT
#define xmalloc malloc
/* Use "gcc -fwritable-strings." */
#endif
#ifndef AUTH_CLIENT_SUPPORT || defined(AUTH_SERVER_SUPPORT)
/* Map characters to each other randomly and symmetrically, A <--> B. */
/* We divide the ASCII character set into 3 domains: control chars (0 thru 31), printing chars (32 through 126), and "meta"-chars (127 through 255). The control chars map to themselves, the printing chars map among themselves, and the meta chars map among themselves. Why is this thus?
   */
   /* No character in any of these domains maps to a character in another domain, because I’m not sure what characters are legal in passwords, or what tools people are likely to use to cut and paste them. It seems prudent not to introduce control or meta chars, unless the user introduced them first. And having the control chars all map to themselves insures that newline and carriage-return are safely handled. */
*/
static unsigned char shifts[33] = {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
  16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
  114,120, 73, 79, 96,109, 72,108, 70, 64, 76, 67,116, 74, 68,87,
  111, 52, 75,119, 49, 34, 82, 81, 95, 65,112, 80,118,110,122,105,
  41, 57, 83, 46,102, 40, 89, 38,103, 45, 50, 42,116, 74, 68, 87,
  125, 55, 54, 64,122,126, 59, 47, 92, 71,115, 78, 88,107,106, 56,
  60, 36,121,117,104,101,100, 69, 73, 99, 63, 94, 93, 39, 37, 61, 48,
  58,113, 32, 90, 44, 98, 60, 51, 33, 97, 62, 77, 84, 80, 85,223,
  225,216,187,166,229,189,222,188,141,249,148,200,184,136,248,190,
  199,170,181,204,138,232,218,183,255,234,220,247,213,203,226,193,
  174,172,228,252,217,201,131,230,197,211,145,238,161,179,160,212,
  207,221,254,173,202,146,224,151,140,196,205,130,135,133,143,246,
  192,159,244,239,185,168,215,144,139,165,180,157,147,186,214,176,
  227,231,219,169,175,156,206,198,129,164,150,210,154,177,134,127,
  182,128,158,208,162,132,167,209,149,241,153,251,237,236,171,195,
  243,233,253,240,194,250,191,155,142,137,245,239,163,242,178,152,
};

/* SCRAMBLE and DESCRAMBLE work like this:
 * scramble(STR) returns SCRM, a scrambled copy of STR. SCRM[0] is a
 * single letter indicating the scrambling method. As of this
 * writing, the only legal method is 'A'; but check the code for more
 * up-to-date information. The copy will have been allocated with
 * malloc(). */

/* Return a malloc'd, scrambled version of STR. */
char *
scramble (str)
  char *str;

return a malloc'd, scrambled version of STR. */
```c
{ int i;
 char *s;
 /* A' prefix indicates which version of scrambling this is (the first, obviously, since we only do one kind of scrambling so far), and then the \"0\" of course. */
 s = (char *) xmalloc(strlen(str) + 2);
 /* Scramble (TM) version prefix. */
 strcpy(s + 1, str);
 for (i = 1; s[i]; i++)
 s[i] = shift((unsigned char)(s[i]));
 return s;
} /* Decode the string in place. */
char *descramble(str){
 char *s;
 int i;
 /* For now we can only handle one kind of scrambling. In the future there may be other kinds, and this \"if\" will become a \"switch\". */
 if (str[0] != 'A')
 #ifndef DIAGNOSTIC
 #ifdef main
 error(1, 0, "descramble: unknown scrambling method");
 s = scramble(str + 1);
 /* Shift the whole string one char to the left, pushing the unwanted \"A\" off the left end. Safe, because s is null-terminated. */
 for (i = 0; s[i]; i++)
 s[i] = s[i + 1];
 return s;
 #endif
 #endif /* (AUTH_CLIENT_SUPPORT || AUTH_SERVER_SUPPORT) from top of file */
 #define DIAGNOSTIC
 error(1, 0, "unknown scrambling method");
 fprintf(stderr, "descramble: unknown scrambling method
" , str);
 fflush(stderr);
 exit(EXIT_FAILURE);
}
#endif /* (AUTH_CLIENT_SUPPORT || AUTH_SERVER_SUPPORT) from top of file */
#endif /* DIAGNOSTIC */
/* Method \"A\" is symmetrical, so scramble again to decrypt. */
s = scramble(str + 1);
/* Shift the whole string one char to the left, pushing the unwanted \"A\" off the left end. Safe, because s is null-terminated. */
for (i = 0; s[i]; i++)
 s[i] = s[i + 1];
return s;
}
/* (AUTH_CLIENT_SUPPORT || AUTH_SERVER_SUPPORT) from top of file */
#define DIAGNOSTIC
int main (){
 int i;
 char *e, *m, biggie[256];
 char *cleartexts[5];
 cleartexts[0] = "first";
 cleartexts[1] = "the second";
 cleartexts[2] = "this is the third";
 cleartexts[3] = "$!% !\1\3;";
 cleartexts[4] = biggie;
 /* Set up the most important test string: */
 /* Can't have a real ASCII zero in the string, because we want to use printf, so we substitute the character zero. */
 biggie[0] = '0';
 /* The rest of the string gets straight ascending ASCII. */
 for (i = 1; i < 256; i++)
 biggie[i] = i;
 /* Test all the strings. */
 for (i = 0; i < 5; i++)
 { printf("clear%d: %s\n", i, cleartexts[i]);
 e = scramble(cleartexts[i]);
 printf("scram%d: %s\n", i, e);
 m = descramble(e);
 free(e);
 printf("clear%d: %s\n", i, m);
 free(m);
 }
 fflush(stdout);
 return 0;
}

} /* scrambler.c - 605 */
/** DIAGNOSTIC */

/*
* ;; The Emacs Lisp that did the dirty work ;;;
* (progn
* ;; Helper func.
* (define random-elt (list)
* (let* ((len (length list))
*        (rnd (random len))
*        (nth rnd list)))
* ;; A list of all characters under 127, each appearing once.
* (setq non-meta-chars
*       (let ((i 0)
*              (l nil))
*            (while (< i 127)
*                (setq l (cons i l)
*                      i (1+ i)))
*       l))
* ;; A list of all characters 127 and above, each appearing once.
* (setq meta-chars
*       (let ((i 127)
*              (l nil))
*            (while (< i 256)
*                (setq l (cons i l)
*                      i (1+ i)))
*       l))
* ;; A vector that will hold the chars in a random order.
* (setq scrambled-chars (make-vector 256 0))
* ;; These characters should map to themselves.
* (let ((i 0))
*      (while (< i 32)
*            (aset scrambled-chars i i)
*            (setq non-meta-chars (delete i non-meta-chars)
*                                i (1+ i))))
* ;; Assign random (but unique) values, within the non-meta chars.
* (let ((i 32))
*      (while (< i 127)
*            (let ((ch (random-elt non-meta-chars)))
*                (if (= 0 (aref scrambled-chars i))
*                    (progn
*                        (aset scrambled-chars i ch)
*                        (aset scrambled-chars ch i)
*                        (setq non-meta-chars (delete ch non-meta-chars)
*                                             non-meta-chars (delete i non-meta-chars)))))
*            (setq i (1+ i))))
* ;; Assign random (but unique) values, within the non-meta chars.
* (let ((i 127))
*      (while (< i 256)
*            (let ((ch (random-elt meta-chars)))
*                (if (= 0 (aref scrambled-chars i))
*                    (progn
*                        (aset scrambled-chars i ch)
*                        (aset scrambled-chars ch i)
*                        (setq meta-chars (delete ch meta-chars))
*                        (setq meta-chars (delete i meta-chars)))))
*            (setq i (1+ i))))
* ;; Now use the 'scrambled-chars' vector to get your C array.
* ) */
A.55 server.c

/* This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
y any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

10 #include <assert.h>
#include "cvs.h"
#include "watch.h"
#include "edit.h"
#include "fiisattr.h"
#include "getline.h"
#include "buffer.h"

#ifndef SERVER_SUPPORT
#endif

ifdef SERVER_SUPPORT

ifdef HAVE_WINSOCK_H
#include <winsock.h>
#endif

ifdef defined (AUTH_SERVER_SUPPORT) || defined (HAVE_KERBEROS) || defined (HAVE_GSSAPI)
#include <sys/socket.h>
#endif

ifdef HAVE_KERBEROS
# include <netinet/in.h>
# include <krb.h>
# ifndef HAVE_KRB_GET_ERROR_TEXT
# define krb_krb5_strerror(status) krb5_strerror[status]
# endif

/* Information we need if we are going to use Kerberos encryption. */
static C_Block kblock;
static Key_schedule sched;
#endif

ifdef HAVE_GSSAPI
#include <netdb.h>

ifdef HAVE_GSSAPI_H
#include <gssapi.h>
#endif

ifdef HAVE_GSSAPI_GSSAPI_H
#include <gssapi/gssapi.h>
#endif

ifdef HAVE_GSSAPI_GSSAPI_GENERIC_H
#include <gssapi/gssapi_generic.h>
#endif

ifdef HAVE_GSSAPI_NT_HOSTBASED_SERVICE
#define GSSAPI_NT_HOSTBASED_SERVICE (gssapi)_service = name
#endif

60 /* We use Kerberos 5 routines to map the GSSAPI credential to a user
name. */
#include <krb5.h>

/* We need this to wrap data. */
static gssctx_t gcontext;
static int gserver_authenticate_connection PROTO((int));
static int kserver_authenticate_connection PROTO((int));
static int pserver_authenticate_connection PROTO((int));

/* Whether we are already wrapping GSSAPI communication. */
static int cvs_gssapi_wraping;

#ifdef ENCRYPTION
/* Whether to encrypt GSSAPI communication. We use a global variable
like this because we use the same buffer type (gssapi_wrap) to
handle both authentication and encryption, and we don’t want
multiple instances of that buffer in the communication stream. */
int cvs_gssapi_encrypt;
#endif

#endif

/* for select */
#include <sys/types.h>
ifdef HAVE_SYS_BSDTYPES_H
#include <sys/bdtyes.h>
#endif

}
# TIME_WITH_SYS_TIME

```c
#ifndef TIME_WITH_SYS_TIME
#define TIME_WITH_SYS_TIME
#endif
```

The server uses it.

The `command` is used to non-blocking writes.

```c
#define NONBLOCK O
```

The `timeout` buffer is used to read input from the client.

```c
static struct
```

This is the original value of `server_temp_dir`, before any possible changes inserted by `server_main_loop()`. *

```c
static char *orig_server_temp_dir;
```

```c
static void server_write_entries PROTO((void));
```

All server communication goes through buffer structures. Most of the buffers are built on top of a file descriptor. This structure is used as the closure field in a buffer. *

```c
struct fd_buffer
```
int blocking;

static struct buffer *fd_bufferInitialize
PROTO ((int, int, void (*)(struct buffer *)));
static int fd_bufferInput
PROTO ((void *, char *, int, int, int, int *));
static int fd_bufferOutput
PROTO ((void *, const char *, int, int *));
static int fd_bufferFlush
PROTO ((void *, int));
static int fd_bufferBlock
PROTO ((void *, int));

/* Initialize a buffer built on a file descriptor. FD is the file
descriptor. INPUT is nonzero if this is for input, zero if this is
for output. MEMORY is the function to call when a memory error
occurs. */

static struct buffer *
fd_bufferInitialize (fd, input, memory)
int fd;
int input;
void (*memory) PROTO ((struct buffer *));
{
  struct fd_buffer *n;
  n = (struct fd_buffer *) xmalloc (sizeof *n);
  n->fd = fd;
  n->blocking = 1;
  return bufInitialize (input ? fd_bufferInput : NULL,
                         input ? NULL : fd_bufferOutput,
                         input ? NULL : fd_bufferFlush,
                         fd_bufferBlock,
                         (int (*)(void *)) PROTO ((void *)) NULL,
                         memory, n);
}

/* The buffer input function for a buffer built on a file descriptor. */

static int
fd_bufferInput (closure, data, need, size, got)
void *closure;
char *data;
int need;
int size;
int *got;
{
  struct fd_buffer *fd = (struct fd_buffer *) closure;
  int nbytes;
  if (!fd->blocking)
    nbytes = read (fd->fd, data, size);
  else
    {
      /* This case is not efficient. Fortunately, I don't think it
         ever actually happens. */
      nbytes = read (fd->fd, data, need == 0 ? 1 : need);
    }
  if (nbytes > 0)
    {
      *got = nbytes;
      return 0;
    }
  *got = 0;
  if (nbytes == 0)
    {
      /* End of file. This assumes that we are using POSIX or BSD
         style nonblocking I/O. On System V we will get a zero
         return if there is no data, even when not at EOF. */
      return -1;
    }
/* Some error occurred. */
if (blocking_error (errno))
  {
    /* Everything's fine, we just didn't get any data. */
    return 0;
  }
return errno;

/* The buffer output function for a buffer built on a file descriptor. */

static int
fd_bufferOutput (closure, data, have, wrote)
void *closure;
const char *data;
```c
int have;
int *wrote;
{
    struct fd_buffer *fd = (struct fd_buffer *) closure;
    *wrote = 0;
    while (have > 0)
    {
        int nbytes;
        nbytes = write ((fd->fd, data, have);
        if (nbytes <= 0)
            {
                if (fd->blocking
                    && (nbytes == 0 || blocking_error (errno)))
                    {  /* A nonblocking write failed to write any data. Just
                        return. */
                        return 0;
                    }
                    /* Some sort of error occurred. */
                    if (nbytes == 0)
                        return EIO;
                    return errno;
                }
                *wrote += nbytes;
                data += nbytes;
                have -= nbytes;
            }
            return 0;
        }
        /* The buffer flush function for a buffer built on a file descriptor. */
        static int fd_buffer_flush (closure)
        {
            /* Nothing to do. File descriptors are always flushed. */
            return 0;
        }
        /*ARGSUSED*/
        static int fd_buffer_block (closure, block)
        {
            struct fd_buffer *fd = (struct fd_buffer *) closure;
            int flags;
            flags = fcntl ((fd->fd, F_GETFL, 0);
            if (flags < 0)
                return errno;
            if (block)
                flags &= ~O_NONBLOCK;
            else
                flags |= O_NONBLOCK;
            if (fcntl ((fd->fd, F_SETFL, flags) < 0)
                return errno;
            fd->blocking = block;
            return 0;
        }
        /* Make directory DIR, including all intermediate directories if necessary.
        * Returns 0 for success or errno code. */
        static int mkdirpPROTO((char *));
        static int mkdirp (dir)
        {
            char *dir;
            char *p;
            char *eq = malloc (strlen (dir) + 1);
            int retval;
```
if (q == NULL)
    return ENOMEM;

retval = 0;

/*
 * Skip over leading slash if present. We won't bother to try
 * make '/':
 */
p = dir + 1;

while (1)
{
    while ((p != '/') && (p != '\0'))
        ++p;
    if (*p == '/')
    {
        strncpy(q, dir, p - dir);
        q[p - dir] = '\0';
        if (q[p - dir - 1] != '/' && CV8_MKDIR (q, 0777) < 0)
        {
            int saved_errno = errno;
            if (saved_errno != EEXIST &&
                (saved_errno != EACCES && saved_errno != EROFS))
            {
                retval = saved_errno;
                goto done;
            }
        }
        else
        {
            if (CV8_MKDIR (dir, 0777) < 0)
                retval = errno;
            goto done;
        }
    }
}
done:
free (q);
return retval;

/*
 * Print the error response for error code STATUS. The caller is
 * responsible for making sure we get back to the command loop without
 * any further output occurring.
 * Must be called only in contexts where it is OK to send output.
 */

static void
print_error (status)
{
    char *msg;
    buf_output0 (buf8out Net, error *
    msg = strerror (status);
    if (msg)
        buf_output0 (buf8out Net, msg);
    buf_append_char (buf8out Net, '\n');
    buf_flush (buf8out Net, 0);
}

static int pending_error;

/*
 * Malloc'd text for pending error. Each line must start with "E ".
 * The last line should not end with a newline.
 */
static char *pending_error_text;

/* If an error is pending, print it and return 1. If not, return 0.
 * Must be called only in contexts where it is OK to send output.
 */

static int
print_pending_error ()
{
    if (pending_error_text)
    {
        buf_output0 (buf8out Net, pending_error_text);
        buf_append_char (buf8out Net, '\n');
        if (pending_error)
            print_error (pending_error);
        else
            buf_output0 (buf8out Net, *error '\n');
        buf_flush (buf8out Net, 0);
        pending_error = 0;
        free (pending_error_text);
    }
pending_error_text = NULL;

return 1;
}
else if (pending_error)
{
    print_error (pending_error);
    pending_error = 0;
    return 1;
}
else
    return 0;

/* Is an error pending? */
#define error_pending () (pending_error || pending_error_text)

static int alloc_pending PROTO ((size_t size));

/* Allocate SIZE bytes for pending_error_text and return nonzero
if we could do it. */
static int alloc_pending (size)
{
    size_t size;
    {
        if (error_pending ()
            /* Probably alloc_pending callers will have already checked for
               this case. But we might as well handle it if they don't, I
               guess. */
            return 0;
        pending_error_text = malloc (size);
        if (pending_error_text == NULL)
        {
            pending_error = ENOMEM;
            return 0;
        }
    return 1;
}

static void serve_unmodified PROTO ((char *));

static int supported_response PROTO ((char *));

static int supported_response (name)
char *name;
{
    struct response *rs;

    for (rs = responses; rs->name != NULL; ++rs)
        if (strcmp (rs->name, name) == 0)
            return rs->status == rs_supported;

    error (1, 0, "internal error: testing support for unknown response?");
    /* NOTREACHED */
    return 0;
}

static void serve_unmodified responses (arg)
char *arg;
{
    char *p = arg;
    char *q;
    struct response *rs;

do{
    q = strchr (p, ' ');
    if (q != NULL)
        q += 1;
    for (rs = responses; rs->name != NULL; ++rs)
    {
        if (strcmp (rs->name, p) == 0)
            break;
    }
    if (rs->name == NULL)
        /*
         * It is a response we have never heard of (and thus never
         * will want to use). So don't worry about it.
         * /
        
    else
        rs->status = rs_supported;

    p = q;
} while (q != NULL);

for (rs = responses; rs->name != NULL; ++rs)
{
    if (rs->status == rs_essential)
    {
        buf_output0 (bufq, net, "% response ");
        buf_output0 (bufq, net, rs->name);
        buf_output0 (bufq, net, " not supported by client\nerr\" ");
    }
540 /* FIXME: This call to bufflush could conceivably cause deadlock, as noted in server.c. cleanup. */
bufflush (buf, net, 1);

/* I'm doing this manually rather than via error_exit ()
because I'm not sure whether we want to call server_cleanup.
Needs more investigation.... */

#define SYSTEM_CLEANUP
/* Hook for OS-specific behavior, for example socket subsystems on NT and OS2 or dealing with windows and arguments on Mac. */

sendif

   exit (EXIT_FAILURE);
}

else if (rs->status == rs_optimal)
   rs->status = rs_not_supported;
}

static void
serve_root (arg)
char *arg;

   char *env;
char *path;
int save_errno;

   if (error_pending()) return;

   if (!isabsolute (arg))
      {
   if (allow_pending (80 + strlen (arg)))
      sprintf (pending_error_text,
            "E Root %s must be an absolute pathname", arg);
      return;
      }

   /* Sending "Root" twice is illegal. It would also be nice to check for the other case, in which there is no Root request prior to a request which requires one.

The other way to handle a duplicate Root requests would be as a request to clear out all state and start over as if it was a new connection. Doing this would cause interoperability headaches, so it should be a different request, if there is any reason why such a feature is needed. */

   if (CVSroot_directory != NULL) {  
   if (allow_pending (80 + strlen (arg)))
      sprintf (pending_error_text,
            "E Protocol error: Duplicate Root request, for \%s", arg);
      return;
      }

   setlocal_cvroot (arg);

   /* For pserver, this will already have happened, and the call will do nothing. But for rsh, we need to do it now. */

   parse_config (CVSroot_directory);

   path = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM) + sizeof (CVSROOTADM_HISTORY) + 10);
   printf (path, "\%s\%s", CVSroot_directory, CVSROOTADM);
   if (isaccessible (path, R_OK | W_OK)) {  
   save_errno = errno;
   pending_error_text = malloc (80 + strlen (path));
   if (pending_error_text != NULL)
      sprintf (pending_error_text, "E Cannot access \%s", path);
   pending_error = save_errno;
   }
   }

   strcat (path, ":/"):
   strcat (path, CVSROOTADM_HISTORY);
   if (isfile (path) && isaccessible (path, R_OK | W_OK))  
   {
   save_errno = errno;
   pending_error_text = malloc (80 + strlen (path));
   if (pending_error_text != NULL)
      sprintf (pending_error_text, "E \

      Sorry, you don't have read/write access to the history file \%s", path);
   pending_error = save_errno;
   }

   free (path);

#define HAVE_PUTENV
static void
serve_max_dotdot (arg)
char *arg;
{
    int lim = atoi (arg);
    int i;
    char *p;
    if (lim < 0)
        return;

    p = malloc (strlen (server_temp_dir) + 2 * lim + 10);
    if (p == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcpy (p, server_temp_dir);
    for (i = 0; i < lim; i++)
        strcat (p, "d/d")
    if (server_temp_dir != orig_server_temp_dir)
        free (server_temp_dir);
    server_temp_dir = p;
    max_dotdot_limit = lim;
}

static char *dir_name;
static void
dirswitch (dir, repos)
char *dir;
{
    int status;
    FILE *f;
    char *b;
    size_t dstuff; 
    server_write_entries ();
    ...
if (errorpending()) return;

if (dir_name != NULL)
    free(dir_name);

dir_len = strlen(dir);

/* Check for a trailing '/'. This is not ISDIRSEP because \ in the
protocol is an ordinary character, not a directory separator (of
course, it is perhaps unwise to use it in directory names, but that
is another issue). */

if (dir_len > 0
    && dir[dir_len - 1] == '/')
{
    if (allocpending(80 + dir_len))
        sprintf(pendingerrorext,
            "E protocol error: invalid directory syntax in '%s'. \ dir
            ");
    return;
}

dir_name = malloc(strlen(server_tempdir) + dir_len + 40);

if (dir_name == NULL)
{
    pending_error = ENOMEM;
    return;
}

strcpy(dir_name, server_tempdir);
strcat(dir_name, "/");
strcat(dir_name, dir);

status = mkdir(dir_name);
if (status != 0
    && status != EEXIST)
{
    pending_error = status;
    if (allocpending(80 + strlen(dir_name)))
        sprintf(pendingerrorext, "E cannot mkdir '%s', dir_name); 
    return;
}

/* Note that this call to SubdirRegister will be a no-op if the parent
directory does not get exist (for example, if the client sends 
"Directory foo" followed by "Directory .", then the subdirectory does
not get registered, but if the client sends "Directory ." followed
by "Directory foo", then the subdirectory does get registered. 
This seems pretty fishy, but maybe it is the way it needs to work. */

b = strchr(dir_name, '/');
*b = '\0';
SubdirRegister((List *) NULL, dir_name, b + 1);
*b = '/';

if (CVS_CHDIR(dir_name) < 0)
{
    pending_error = errno;
    if (allocpending(80 + strlen(dir_name)))
        sprintf(pendingerrorext, "E cannot change to '%s', dir_name); 
    return;
}

/* This is pretty much like calling CreateAdmin, but CreateAdmin doesn't
 * report errors in the right way for us.
 */

if (CVS_MKDIR(CVSADM, 0777) < 0)
{
    if (errno == EEXIST)
        /* Don't create the files again. */
    return;
    pending_error = errno;
    return;
}

f = CVS_FOPEN(CVSADM_REP, "w");
if (f == NULL)
{
    pending_error = errno;
    return;
}

if (fprintf(f, "%s", repos) < 0)
{
    pending_error = errno;
    fclose(f);
    return;
}

/* Non-remote CVS handles a module representing the entire tree 
(e.g., an entry like "world -a ") by putting \ at the end 
of the Repository File, so we do the same. */

if (strcmp(dir, "") == 0
    && CVSroot_directory != NULL
    && strcmp(CVSroot_directory, repos) == 0)
{
if (fprint (f, "/", c) < 0)
{
    pending_error = errno;
    fclose (f);
    return;
}
}
if (fprint (f, "/a") < 0)
{
    pending_error = errno;
    fclose (f);
    return;
}
if (fclose (f) == EOF)
{
    pending_error = errno;
    return;
} /* We open in append mode because we don't want to clobber an existing Entries file. */
f = CVS_FOPEN (CVSADM_ENT, "/a");

if (f == NULL)
{
    pending_error = errno;
    if (alloc_pending (80 + strlen (CVSADM_ENT)))
        sprintf (pending_error_text, "
E cannot open "/a", CVSADM_ENT);
    return;
}
else if (fclose (f) == EOF)
{
    pending_error = errno;
    if (alloc_pending (80 + strlen (CVSADM_ENT)))
        sprintf (pending_error_text, "
E cannot close "/a", CVSADM_ENT);
    return;
}
}

static void 
serve_repository (arg)
{
    char *arg;

    pending_error_text = malloc (80);
    if (pending_error_text == NULL)
        pending_error = ENOMEM;
    else
        strcpy (pending_error_text, 
E repository request is obsolete; aborted
);
    return;
}

static void 
serve_directory (arg)
{
    char *arg;

    int status;
    char repos;

    status = buf_readline (buf_from_net, &repos, (int *) NULL);
    if (status == 0)
{
    /* I think isabsolute (repos) should always be true, and that
any RELATIVE_REPOS stuff should only be in CVS/Repository
files, not the protocol (for compatibility), but I'm putting
in the in isabsolute check just in case. */
    if (isabsolute (repos))
        &repos = strncmp (CVSroot_directory, repos, 
strlen (CVSroot_directory)) != 0)
{
    if (alloc_pending (strlen (CVSroot_directory) + strlen (repos) + 80))
        sprintf (pending_error_text, 
E protocol error: directory "/a" not within root "/a*", repos, CVSroot_directory);
    return;
}
    else
        direct (arg, repos);
    free (repos);
}
else if (status == -2)
{
    pending_error = ENOMEM;
}
else
{
    pending_error_text = malloc (80 + strlen (arg));
    if (pending_error_text == NULL)
{
pending_error = ENOMEM;
}
else if (status == -1)
{
    fprintf (pending_error_text,
     "E end of file reading mode for %s", arg);
}
else
{
    fprintf (pending_error_text,
     "E error reading mode for %s", arg);
}
pending_error = status;
}

static void
serve_static_directory (arg)
{
    FILE *f;

    if (error_pending (!)) return;
    f = CVS_FOPEN (CVSADM_ENTSTAT, "w+");
    if (f == NULL)
    {
        pending_error = errno;
        if (allow_pending (80 + strlen (CVSADM_ENTSTAT)))
            fprintf (pending_error_text, "E cannot open %s", CVSADM_ENTSTAT);
        return;
    }

    if ((fclose (f)) == EOF)
    {
        pending_error = errno;
        if (allow_pending (80 + strlen (CVSADM_ENTSTAT)))
            fprintf (pending_error_text, "E cannot close %s", CVSADM_ENTSTAT);
        return;
    }

    static void
serve_sticky (arg)
{
    FILE *f;

    if (error_pending (!)) return;
    f = CVS_FOPEN (CVSADM_TAG, "w+");
    if (f == NULL)
    {
        pending_error = errno;
        if (allow_pending (80 + strlen (CVSADM_TAG)))
            fprintf (pending_error_text, "E cannot open %s", CVSADM_TAG);
        return;
    }

    if ((fprintf (f, "%a", arg) < 0)
    {
        pending_error = errno;
        if (allow_pending (80 + strlen (CVSADM_TAG)))
            fprintf (pending_error_text, "E cannot write to %s", CVSADM_TAG);
        return;
    }

    if ((fclose (f)) == EOF)
    {
        pending_error = errno;
        if (allow_pending (80 + strlen (CVSADM_TAG)))
            fprintf (pending_error_text, "E cannot close %s", CVSADM_TAG);
        return;
    }

    /*
     * Read SIZE bytes from buf_from_net, write them to FILE.
     * Currently this isn't really used for receiving parts of a file -
     * the file is still sent over in one chunk.  But if/when we get
     * spiffy in-process gzip support working, perhaps the compressed
     * pieces could be sent over as they're ready, if the network is fast
     * enough.  Or something.
    */

    static void
receive_partial_file (size, file)
    {
        int size;
        int file;
        {
            while (size > 0)
            {
                int status, nread;
```c
char *data;

status = buf_read_data (buf_from_net, size, &data, &nread);
if (status != 0)
{
  if (status == -2)
    pending_error = ENOMEM;
  else
    {
      pending_error_text = malloc (80);
      if (pending_error_text == NULL)
        pending_error =ENOMEM;
      else if (status == -1)
        {
          sprintf (pending_error_text, "E premature end of file from client");
          pending_error = 0;
        }
      else
        {
          sprintf (pending_error_text, "E error reading from client");
          pending_error = status;
        }
  }
  return;
}

size -= nread;

while (nread > 0)
{
  int nwrote;

  nwrote = write (file, data, nread);
  if (nwrote < 0)
  {
    pending_error_text = malloc (40);
    if (pending_error_text != NULL)
      sprintf (pending_error_text, "E unable to write");
    pending_error = errno;
  }
  else
  {
    /* Read and discard the file data. */
    while (size > 0)
    {
      int status, nread;
      char *data;
      status = buf_read_data (buf_from_net, size, &data, &nread);
      if (status != 0)
        return;
      size -= nread;
    }
    return;
  }
  nread -= nwrote;
  data += nwrote;
}

/* Receive SIZE bytes, write to filename FILE. */
static void
receive_file (size, file, gzipped)
{
  int size;
  char *file;
  int gzipped;

  int fd;
  char *arg = file;

  pid_t gzippid = 0;
  int gzipstatus;

  /* Write the file. */
  fd = CVS_OPEN (arg, O_WRONLY | O_CREAT | O_TRUNC, 0600);
  if (fd < 0)
  {
    pending_error_text = malloc (40 + strlen (arg));
    if (pending_error_text != NULL)
      sprintf (pending_error_text, "E cannot open %s", arg);
    pending_error = errno;
    return;
  }

  /* FIXME: This doesn't do anything reasonable with gunzip's stderr, which
   * means that if gunzip writes to stderr, it will cause all manner of
   * protocol violations.
   */
}
if (gzipped)
    fd = filter_through_gunzip (fd, 0, &gzip_pid);
receive_partial_file (size, fd);
if (pending_error_text)
{
    char *p = realloc (pending_error_text, 
                     strlen (pending_error_text) + strlen (arg) + 30);
    if (p)
    {
        pending_error_text = p;
        sprintf (p + strlen (p), "", file "f", arg);
    } /* else original string is supposed to be unchanged */
}
if (close (fd) < 0 && error_pending ()
{
    pending_error_text = malloc (40 + strlen (arg));
    if (pending_error_text)
        sprintf (pending_error_text, "E cannot close %s", arg);
    pending_error = errno;
    if (gzip_pid)
        waitpid (gzip_pid, (int *) 0, 0);
    return;
}
if (gzip_pid)
{
    if (waitpid (gzip_pid, &gzip_status, 0) != gzip_pid)
        error (l, errno, "waiting for gunzip process %ld", 
               (long) gzip_pid);
    else if (gzip_status != 0)
        error (l, 0, "gunzip exited %s", gzip_status);
}
/* Kopt for the next file sent in Modified or Is-modified */
static char *kopt = kopt;

static void serve_modifiedPROTO ((char *));

static void
serve_modified (arg)
{
    char *arg;
    int size, status;
    char *size_text;
    char *mode_text;

    int gzipped = 0;
    /* This used to return immediately if error_pending () was true.
     * However, that fails, because it causes each line of the file to
     * be echoed back to the client as an unrecognized command. The
     * client isn't reading from the socket, so eventually both
     * processes block trying to write to the other. Now, we try to
     * read the file if we can.
    */

    status = buf_read_line (buf_from_net, &mode_text, (int *) NULL);
    if (status != 0)
    {
        if (status == -2)
            pending_error = ENOMEM;
        else
            pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text)
                sprintf (pending_error_text, 
                         "E error reading mode for %s", arg);
            else
                sprintf (pending_error_text, 
                         "E end of file reading mode for %s", arg);
            pending_error = status;
    }
    return;
}
status = buf_read_line (buf_from_net, &size_text, (int *) NULL);
if (status != 0)
if (status == -2)
    pending_error = ENOMEM;
else
    pending_error_text = malloc (80 + strlen (arg));
    if (pending_error_text == NULL)
        pending_error = ENOMEM;
else
    if (status == -1)
        sprintf (pending_error_text,
            "E end of file reading size for %s", arg);
    else
        sprintf (pending_error_text,
            "E error reading size for %s", arg);
    pending_error = errno;

return;
}

if (size_text[0] == 'z')
    gzipped = 1;
    size = atoi (size_text + 1);
else
    size = atoi (size_text);
    free (size_text);

if (error_pending ()
    { /* Now that we know the size, read and discard the file data. */
      while (size > 0)
        { int status, nread;
            char *data;
            status = buf_read_data (buf_from_net, size, &data, &nread);
            if (status != 0)
                return;
            size -= nread;
        }
        return;
    }
    if (size > 0)
    { receive_file (size, arg, gzipped);

    if (errorPending ()) return;
    }

    int status = change_mode (arg, mode_text, 0);
    free (mode_text);
    if (status)
    { pending_error_text = malloc (40 + strlen (arg));
        if (pending_error_text)
            sprintf (pending_error_text,
                "E cannot change mode for %s", arg);
        pending_error = status;
        return;
    }

    /* Make sure that the Entries indicate the right kopt. We probably
could do this even in the non-kopt case and, I think, save a stat()
call in time stamp server. But for conservatism I'm leaving the
non-kopt case alone. */
    if (kopt != NULL)
        serve_js_modified (arg);
}

static void serve_remote_revision PROTO ((char *));
static void serve_remote_revision (arg)
    char *arg;

{ int size, status;
    char *size_text;
    char *mode_text;
    char *rev_text;
    char *remote_file;

    int gzipped = 0;
printf ("serve_remote_revision entered\n");

/*
 * This used to return immediately if error_pending () was true.
 * However, that fails, because it causes each line of the file to
 * be echoed back to the client as an unrecognized command. The
 * client isn't reading from the socket, so eventually both
 * processes block trying to write to the other. Now, we try to
 * read the file if we can.
 */

status = buf_readline (buf_from_net, &rev_text, (int *) NULL);
if (status != 0)
{
    if (status == -2)
        pending_error = ENOMEM;
    else
        { pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text == NULL)
                pending_error = ENOMEM;
        }
    else
        { if (status == -1)
            sprintf (pending_error_text,
                "E end of file reading revision for %s", arg);
        else
            { sprintf (pending_error_text,
                "E error reading revision for %s", arg);
                pending_error = status;
            }
        }
    return;
}

printf ("serve_remote_revision read rev (%s)\n", rev_text);
status = buf_readline (buf_from_net, &mode_text, (int *) NULL);
if (status != 0)
{
    if (status == -2)
        pending_error = ENOMEM;
    else
        { pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text == NULL)
                pending_error = ENOMEM;
        }
    else
        { if (status == -1)
            sprintf (pending_error_text,
                "E end of file reading mode for %s", arg);
        else
            { sprintf (pending_error_text,
                "E error reading mode for %s", arg);
                pending_error = status;
            }
        }
    return;
}

printf ("serve_remote_revision read node (%s)\n", mode_text);
status = buf_readline (buf_from_net, &size_text, (int *) NULL);
if (status != 0)
{
    if (status == -2)
        pending_error = ENOMEM;
    else
        { pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text == NULL)
                pending_error = ENOMEM;
        }
    else
        { if (status == -1)
            sprintf (pending_error_text,
                "E end of file reading size for %s", arg);
        else
            { sprintf (pending_error_text,
                "E error reading size for %s", arg);
                pending_error = errno;
            }
        }
    return;
}
1350 } return;
1351 } if (size_text[0] == 'x')
1352 { gzipped = 1;
1353   size = atoi (size_text + 1);
1354 } else
1355 size = atoi (size_text);
1356 printf ("M serve_remote_revision read size \%s, \%d, size\text", size);
1357 free (size_text);
1358 if (error_pending ())
1359 { /* Now that we know the size, read and discard the file data. */
1359 while (size > 0)
1360 { int status, nread;
1361   char *data;
1362
1363   status = buf_read_data (buf_from_net, size, &data, &nread);
1364   if (status != 0)
1365     return;
1366   size -= nread;
1367 }
1368 return;
1369 }
1370 remote_rev_file = xmalloc (strlen (arg) + strlen (CVSADM_REMOTP_TMP) + strlen (rev_text) + 3);
1371 sprintf (remote_rev_file, "\%s\%s\%s", CVSADM_REMOTP_TMP, rev_text, arg);
1372 if (size >= 0)
1373 { receive_file (size, remote_rev_file, gzipped);
1374   if (error_pending ()) return;
1375 }
1376 int status = change_mode (remote_rev_file, mode_text, 0);
1377 free (mode_text);
1378 if (status)
1379 { pending_error_text = malloc (40 + strlen (remote_rev_file));
1380   if (pending_error_text)
1381     sprintf (pending_error_text, "E cannot change mode for \%s", remote_rev_file);
1382   pending_error = status;
1383   return;
1384 }
1385 printf ("M serve_remote_revision returning\n");
1386 }
1387
static void
serve_enable_unchanged (arg)
char *arg;
1388 { }
1389 struct an_entry {
1390   struct an_entry *next;
1391   char *entry;
1392 };
static struct an_entry *entries;
static void serve_unchanged PROTO ((char *));
1399 static void
serve_unchanged (arg)
char *arg;
1400 { struct an_entry *p;
1401   char *name;
1402   char *cp;
1403   char *timefield;
1404   if (error_pending ()
1405     return;
1406 /* Rewrite entries file to have ‘\n’ in timestamp field. */
1407 for (p = entries; p != NULL; p = p->next)
1408 { name = p->entry + 1;
1409   cp = strchr (name, '\n');
1410   if (cp != NULL
1411     *cp = '\n';
1412     *cp++ = '\n';
1413     *cp++ = '\n';
1414     *cp++ = '\n';
1415     *cp++ = '\n';
1416     *cp++ = '\n';
1417     *cp++ = '\n';
1418     *cp++ = '\n';
1419     *cp++ = '\n';
1420     *cp++ = '\n';
1421     *cp++ = '\n';
1422     *cp++ = '\n';
1423     *cp++ = '\n';
1424     *cp++ = '\n';
1425     *cp++ = '\n';
1426     *cp++ = '\n';
1427     *cp++ = '\n';
1428     *cp++ = '\n';
1429 }
& & strlen (arg) == cp - name
& & strncmp (arg, name, cp - name) == 0)
{
  timefield = strchr (cp + 1, '/ ') + 1;
  if (*timefield != 's')
  {
    cp = timefield + strlen (timefield);
    cp[0] = '\0';
    while (cp > timefield)
    {
      *cp = cp[-1];
      --cp;
    }
  }
  timefield = '* ';
}
break;
}
static void
serve_is_modified (arg)
{
  char *arg;
  struct an_entry *p;
  char *name;
  char *cp;
  char *timefield;
  /* Have we found this file in "entries" yet. */
  int found;
  if (errno_pending ())
    return;
  /* Rewrite entries file to have 'M' in timestamp field. */
  found = 0;
  for (p = entries; p != NULL; p = p->next)
  {
    name = p->entry + 1;
    cp = strchr (name, '/ ');
    if (cp != NULL)
    {
      & & strlen (arg) == cp - name
      & & strncmp (arg, name, cp - name) == 0)
      {
        timefield = strchr (cp + 1, '/ ') + 1;
        if ((timefield[0] == 'M' & & timefield[1] == '/ '))
        {
          cp = timefield + strlen (timefield);
          cp[0] = '\0';
          while (cp > timefield)
          {
            *cp = cp[-1];
            --cp;
          }
        }
      }
    }
    else
    {
      if (alloc_pending (strlen (name) + 80))
        sprintf (pending_error_text, "E protocol error: both Kopt and Entry for %s",
        arg);
    free (kopt);
    kopt = NULL;
    return;
    }
  }
  found = 1;
  break;
}
if (!found)
{
  /* We got Is-modified but no Entry. Add a dummy entry.
   * The "D" timestamp is what makes it a dummy. */
  p = (struct an_entry *) malloc (sizeof (struct an_entry));
  if (p == NULL)
  {
    pending_error = ENOMEM;
    return;
  }
  p->entry = malloc (strlen (arg) + 80);
  if (p->entry == NULL)
  {
    pending_error = ENOMEM;
    free (p);
    return;
  }
  strcpy (p->entry, " /*");
  strcat (p->entry, arg);
  strcat (p->entry, "/D */");
}
if (kopt != NULL)
     {
         strcat (p->entry, kopt);
         free (kopt);
         kopt = NULL;
     }  
strcat (p->entry, "/");
p->next = entries;
entries = p;
}

static void serve_entry PROTO ((char *));

static void serve_entry (arg)
     {
     struct an_entry *p;
     char *cp;
     if (error_pending()) return;
     p = (struct an_entry *) malloc (sizeof (struct an_entry));
     if (p == NULL) {
         pending_error = ENOMEM;
         return;
     } /* Leave space for serve unchanged to write '=' if it wants. */
     cp = malloc (strlen (arg) + 2);
     if (cp == NULL) {
         pending_error = ENOMEM;
         return;
     }
     strcpy (cp, arg);
p->entry = cp;
p->next = entries;
entries = p;
}

static void serve_kopt PROTO ((char *));

static void serve_kopt (arg)
     {
     if (error_pending ())
         return;
     if (kopt != NULL) {
     if (allow_pending (80 + strlen (arg)))
         sprintf (pending_error_text,  
             "E protocol error: duplicate Kopt request: %s", arg);
         return;
     } /* Do some sanity checks. In particular, that it is not too long. */
     if (strlen (arg) > 10)
     if (allow_pending (80 + strlen (arg)))
         sprintf (pending_error_text,  
             "E protocol error: invalid Kopt request: %s", arg);
         return;
     kopt = malloc (strlen (arg) + 1);
     if (kopt == NULL) {
         pending_error = ENOMEM;
         return;
     }
     strcpy (kopt, arg);
     }

static void server_write_entries ()
     {
     FILE *f;
     struct an_entry *p;
     struct an_entry *q;
     if (entries == NULL) return;
     f = NULL;
     if (kopt != NULL) 
         return;
     }
/* Note that we free all the entries regardless of errors. */

if (error_pending (i))
{
  /* We open in append mode because we don't want to clobber an existing Entries file. If we are checking out a module which explicitly lists more than one file in a particular directory, then we will wind up calling server_list/entries for each such file. */
  f = CVS_OPEN (CVSADM_ENT, "a");
  if (f == NULL)
  {
    pending_error = errno;
    if (alloc_pending (80 + strlen (CVSADM_ENT)))
      sprintf (pending_error_text, "E cannot open %s", CVSADM_ENT);
  }
  for (p = entries; p != NULL;)
  {
    if (error_pending (i))
    {
      if (fprint (f, "%\n", p->entry) < 0)

        pending_error = errno;
      if (alloc_pending (80 + strlen (CVSADM_ENT)))
        sprintf (pending_error_text, "E cannot write to %s", CVSADM_ENT);
    }
    free (p->entry);
    p = p->next;
    free (p);
  }
}

struct notify_note {
  /* Directory in which this notification happens. malloc'd. */
  char *dir;

  /* malloc'd. */
  char *filename;

  /* The following three all in one malloc'd block, pointed to by TYPE.
   * Each \0 terminated. */
  char *type;

  /* time + host + dir */
  char *val;

  struct notify_note *next;
};

static struct notify_note *notify_list;

/* Used while building list, to point to the last node that already exists. */
static struct notify_note *last_node;

static void serve_notify PROTO ((char *));

static void
serve_notify (arg)
{
  struct notify_note *new;

  char *data;
  int status;
  if (error_pending (i)) return;

  new = struct notify_note * malloc (sizeof (struct notify_note));
  if (new == NULL)
  {
    pending_error = ENOMEM;
    return;
 }
  if (dir_name == NULL)
    goto error;
  new->dir = malloc (strlen (dir_name) + 1);
  if (new->dir == NULL)
  {
    pending_error = ENOMEM;
    return;
  }

  serve_notify (arg);
strcpy (new->dir, dir_name);
new->filename = malloc (strlen (arg) + 1);
if (new->filename == NULL)
{
    pending_error = ENOMEM;
    return;
}
strcpy (new->filename, arg);
status = buf_read_line (buf_from_net, &data, (int *) NULL);
if (status != 0)
{
    if (status == -2)
        pending_error = ENOMEM;
    else
    {
        pending_error_text = malloc (80 + strlen (arg));
        if (pending_error_text == NULL)
            pending_error =ENOMEM;
        else
        {
            if (status == -1)
                sprintf (pending_error_text, "E end of file reading notification for %s", arg);
            else
                sprintf (pending_error_text, "E error reading notification for %s", arg);
            pending_error = status;
        }
    }
}
else
{
    char *cp;
    new->type = data;
    if (data[1] != '\t')
        goto error;
    data[1] = '\0';
    cp = data + 2;
    new->val = cp;
    cp = strchr (cp, '\t');
    if (cp == NULL)
        goto error;
    *cp++ = '*';
    cp = strchr (cp, '\t');
    if (cp == NULL)
        goto error;
    *cp++ = '*';
    cp = strchr (cp, '\t');
    if (cp == NULL)
        goto error;
    *cp++ = '\0';
    new->next = cp;
    /* If there is another tab, ignore everything after it,
     * for future expansion. */
    cp = strchr (cp, '\t');
    if (cp != NULL)
    {
        *cp = '\0';
    }
    new->next = NULL;
    if (last_node == NULL)
    {
        notify_list = new;
    }
    else
    {
        last_node->next = new;
        last_node = new;
    }
    return;
error:
    pending_error_text = malloc (40);
    if (pending_error_text)
        strcpy (pending_error_text, "E Protocol error; misformed Notify request");
    pending_error = 0;
    return;
}

/* Process all the Notify requests that we have stored up. Returns 0
 * if successful, if not prints error message (via error()) and
 * returns negative value. */
static int
server_notify ()
{
struct notify_note *p;
char *repos;

while (notify_list != NULL) {
    if (CVS_CHDIR (notify_list->dir) < 0) {
        error (0, errno, "cannot change to %s", notify_list->dir);
        return -1;
    }
    repos = NameRepository (NULL, NULL);
    lockdirforwrite (repos);
    fileattribstartdir (repos);
    notify_do (*notify_list->type, notify_list->filename, getcaller(),
               notify_list->val, notify_list->watches, repos);
    buf_output0 (buf_to_net, "Notified ");
    char *dir = notify_list->dir + strlen (server_tempdir) + 1;
    if (dir[strlen(dir)] == '\0')
        buf_append_char (buf_to_net, '\n');
    else
        buf_output0 (buf_to_net, dir);
    buf_append_char (buf_to_net, ' ');
    buf_append_char (buf_to_net, '\n');
    buf_output0 (buf_to_net, repos);
    buf_append_char (buf_to_net, '\n');
    buf_output0 (buf_to_net, notify_list->filename);
    buf_append_char (buf_to_net, '\n');
    p = notify_list->next;
    free (notify_list->filename);
    free (notify_list->dir);
    free (notify_list->type);
    free (notify_list);
    notify_list = p;
}

fileattribwrite ();
fileattribfree ();

LockCleanup ();

return 0;

static int argument_count;
static char **argument_vector;
static int argument_vector_size;

static void
serve_argument (arg)
char *arg;

char *p;

if (error_
pending()) return;

if (argument_vector_size <= argument_count) {
    argument_vector_size += 2;
    argument_vector = (char **) realloc ((char *)argument_vector,
                                        argument_vector_size +
                                        sizeof (char *));
    if (argument_vector == NULL) {
        pending_error = ENOMEM;
        return;
    }
    p = malloc (strlen (arg) + 1);
    if (p == NULL) {
        pending_error = ENOMEM;
        return;
    }
    strcpy (p, arg);
    argument_vector[argument_count++] = p;
}

static void
serve_arguments (arg)
char *arg;

1890  {
1891      char *p;
1892      if (error_pending()) return;
1893      p = argument_vector[argument_count - 1];
1894      if (p == NULL) pending_error = ENOMEM;
1895      return;
1896  }
1897  strcat(p, "\n");
1898  strcat(p, arg);
1899  argument_vector[argument_count - 1] = p;

static void
1900  serve_global_option (arg)
1901      char *arg;
1902  {  
1904          error_return:
1905          if (alloc_pending(strlen(arg) + 80))
1906              sprintf(pending_error_text,
1907                      "E Protocol error: bad global option %s", arg);
1908          return;
1909      }
1910      switch (arg[1])
1911      {
1912          case 'n':
1913              noexec = 1;
1914              break;
1915          case 'q':
1916              quiet = 1;
1917              break;
1918          case 'r':
1919              cvswrite = 0;
1920              break;
1921          case 'Q':
1922              really_quiet = 1;
1923              break;
1924          case 'l':
1925              logoff = 1;
1926              break;
1927          case 't':
1928              trace = 1;
1929              break;
1930          default:
1931              goto error_return;
1932      }
1933  }

static void
1934  serve_set (arg)
1935      char *arg;
1936  {
1937      /* FIXME: This ends errors immediately (I think); they should be put into pending_error. */
1938      variable_set (arg);
1939  }

#ifdef ENCRYPTION
1940  
1941  ifdef HAVE_KERBEROS
1942  
1943  static void
1944  serve_kerberos_encrypt (arg)
1945      char *arg;
1946  {  
1947      /* All future communication with the client will be encrypted. */
1948      buf_to_net = krb_encrypt_buffer_initialize (buf_to_net, 0, sched, kblock,
1949        buf_to_net->memory_error);
1950      buf_from_net = krb_encrypt_buffer_initialize (buf_from_net, 1, sched, kblock,
1951        buf_from_net->memory_error);
1952  }
1953  
1954  endif /* HAVE_KERBEROS */

#ifdef HAVE_GSSAPI
1955  
1956  static void
1957  serve_gssapi_encrypt (arg)
1958      char *arg;
1959  {  
1960      /* All future communication with the client will be encrypted. */
1961      buf_to_net = krb_encrypt_buffer_initialize (buf_to_net, 0, sched, kblock,
1962        buf_to_net->memory_error);
1963      buf_from_net = krb_encrypt_buffer_initialize (buf_from_net, 1, sched, kblock,
1964        buf_from_net->memory_error);
1965  }
1966  
1967  endif /* HAVE_GSSAPI */

#endif

if (cva_gssapi_wrapping)
{
    /* We're already using a gssapi_wrap buffer for stream
       authentication. Flush everything we've output so far, and
       turn on encryption for future data. On the input side, we
       should only have unwrapped as far as the Gssapi-encrypt
       command, so future unwrapping will become encrypted. */
    bufferflush(buf_to_net, 1);
    cva_gssapi_encrypt = 1;
return;
}

/* All future communication with the client will be encrypted. */
cva_gssapi_encrypt = 1;

buf_to_net = cva_gssapi_wrap_bufferinitialize(buf_to_net, 0,
gcontext,
buf_to_net->memory_error);
buf_from_net = cva_gssapi_wrap_bufferinitialize(buf_from_net, 1,
gcontext,
buf_from_net->memory_error);

cva_gssapi_wraping = 1;
}

#define HAVE_GSSAPI */

static void
serve_gssapi_authenticate (arg)
char *arg;
{
    if (cva_gssapi_wraping)
    {
        /* We're already using a gssapi_wrap buffer for encryption.
           That includes authentication, so we don't have to do
           anything further. */
        return;
    }

    buf_to_net = cva_gssapi_wrap_bufferinitialize(buf_to_net, 0,
gcontext,
buf_to_net->memory_error);
buf_from_net = cva_gssapi_wrap_bufferinitialize(buf_from_net, 1,
gcontext,
buf_from_net->memory_error);

    cva_gssapi_wraping = 1;
}

#endif /* HAVE_GSSAPI */

#define SERVER_FLOWCONTROL
/* The maximum we'll queue to the remote client before blocking. */
#endif /* SERVER_FLOWCONTROL */
#define SERVER_FLOOD_WAT
#define SERVER_HIL_WAT(2 * 1024 + 1024)
#endif /* SERVER_HIL_WAT */
#define SERVER_LO_WAT
#define SERVER_LO_WAT((1 * 1024 + 1024)
#endif /* SERVER_LO_WAT */

static int
t_nonblock_fd [fd]

int flags;
{
    int flags;
flags = fcntl(fd, F_SETFL, 0);
if (flags < 0)
return errno;
if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) < 0)
return errno;
return 0;
}

#endif /* SERVER_FLOWCONTROL */
static void serve_questionable PROTO((char *));

static void serve_questionable (arg)
    char *arg;
{
    static int initted;

    if (limited)
    {
        /* Pick up ignores from CVSROOTADMIGNORE, $HOME/.cvsignore on server, 
         and CVSIGNORE on server. */
        ign_setup ();
        initted = 1;
    }

    if (dirname == NULL)
    {
        buf_output (buf_to_net, "E Protocol error: 'Directory' missing");
        return;
    }

    if (!ign_name (arg))
    {
        char *update_dir;

        buf_output (buf_to_net, " M? ", 4);
        update_dir = dirname + strlen (server_temp_dir) + 1;
        if ((update_dir[0] == '.' && update_dir[1] == '\0'))
        {
            buf_output (buf_to_net, update_dir);
            buf_output (buf_to_net, "/", 1);
        }
        buf_output (buf_to_net, arg);
        buf_output (buf_to_net, "/\n", 2);
    }
}

static void serve_case PROTO ((char *));

static void serve_case (arg)
    char *arg;
{
    ign_case = 1;
}

static struct buffer *protocol;
/* This is the output which we are saving up to send to the server, in the 
child process. We will push it through, via the 'protocol' buffer, when 
we have a complete line. */
static struct buffer *saved_output;
/* Likewise, but stuff which will go to stderr. */
static struct buffer *saved_outerr;

static void protocol_memory_error (buf)
    struct buffer *buf;
{
    error (1, ENOMEM, "Virtual memory exhausted");
}

static void outbuf_memory_error (buf)
    struct buffer *buf;
{
    static const char msg[] = "E Fatal server error\nerror ENOMEM Virtual memory exhausted.\n";
    if (command_pid > 0)
        kill (command_pid, SIGTERM);

    /* We have arranged things so that printing this now either will 
    be legal, or the "E fatal error" line will get glommed onto the 
    end of an existing "E" or "M" response. */

    /* If this gives an error, not much we could do. syslog() it? */
    write (STDOUT_FILENO, msg, sizeof (msg) - 1);
    error_exit ();
}

static void
input memory error (buf)

    struct buffer *buf;
    {
        output memory error (buf);
    }

/* If command is legal, return 1.
 * Else if command is illegal and croak() illegal is set, then die.
 * Else just return 0 to indicate that command is illegal.
 */
static int
check_command legal() (cmd_name)
    char *cmd_name;
    {
        /* Right now, only server notices illegal commands - namely,
         * write attempts by a read-only user. Therefore, if CVS_Username
         * is not set, this just returns 1, because CVS_Username unset
         * means server is not active.
         */
    }

#define AUTH_SERVER_SUPPORT
if (CVS_Username == NULL)
    return 1;

if (lookup_command_attribute (cmd_name) & CVS_CMD_MODIFIES_REPOSITORY)
    {
        /* This command has the potential to modify the repository, so
         * we check if the user have permission to do that.
         *
         * (Only relevant for remote users -- local users can do
         * whatever normal Unix file permissions allow them to do.)
         *
         * The decision method:
         *
         * 1. If $CVSROOT/CSADMROOT_READERS exists and user is listed
         *    in it, then read-only access for user.
         *
         * 2. Or if $CVSROOT/CSADMROOT_WRITERS exists and user NOT
         *    listed in it, then also read-only access for user.
         *
         * 3. Else read-write access for user.
         */
        char *linebuf = NULL;
        int num_red = 0;
        size_t linebuf_len = 0;
        char *fname;
        size_t flen;
        FILE *fp;
        int found_it = 0;

        /* else */
        flen = strlen (CVSroot_directory)
            + strlen (CVSROOTADM)
            + strlen (CVSROOTADM_READER)
            + 1;

        fname = xmalloc (flen);
        (void) sprintf (fname, "%s/\%s/\%s", CVSroot_directory,
            CVSROOTADM, CVSROOTADM_READER);

        fp = fopen (fname, "r");
        if (fp == NULL)
            {
            if (existenece error (errno))
                {
                /* Need to deny access, so that attackers can't fool
                 * us with some sort of denial of service attack. */
                error (0, errno, "cannot open \%s", fname);
                free (fname);
                return 0;
                }
            }
        else /* successfully opened readers file */
            {
            while ((num_red = getline (&linebuf, &linebuf_len, fp)) >= 0)
                {
                /* Hmmm, is it worth importing my own readline
                 * library into CVS? It takes care of chopping
                 * leading and trailing whitespace, "#" comments, and
                 * newline automatically when so requested. Would
                 * save some code here... -kff */
                /* Chop newline by hand, for strcmp()'s sake. */
                if (linebuf[num_red - 1] == '\n'
                    linebuf[num_red - 1] = '\0';

                if (strcmp (linebuf, CVS_Username) == 0)
```c
/* If not listed specifically as a reader, then this user has write access by default unless writers are also specified in a file. */
if (num_red < 0 && !feof (fp))
    error (0, errno, "cannot read %s", fname);

/* Now check the writers file. */
flen = strlen (CVSroot_directory) + strlen (CVSROOTADM_Writers) + 3;
fname = xmalloc (flen);
( void ) sprintf (fname, "%s/%s/%s", CVSroot_directory, CVSROOTADM, CVSROOTADM_Writers);
fp = fopen (fname, "r");
if (fp == NULL)
{
    if (linebuf)
        free (linebuf);
    if (existence_error (errno))
    {
        /* Writers file does not exist, so everyone is a writer, by default. */
        free (fname);
        return 1;
    }
    else
    {
        /* Need to deny access, so that attackers can’t fool us with some sort of denial of service attack. */
        error (0, errno, "cannot read %s", fname);
        free (fname);
        return 0;
    }
}
found_it = 0;
while ((num_red = getline (&linebuf, &linebuf_len, fp)) >= 0)
{
    /* Chop newline by hand, for strcmp()'s sake. */
    if (linebuf[num_red - 1] == '\n')
        linebuf[num_red - 1] = '\0';
    if (strcmp (linebuf, CVS_Username) == 0)
    {
        found_it = 1;
        break;
    }
}
if (num_red < 0 && !feof (fp))
    error (0, errno, "cannot read %s", fname);

if (found_it)
{
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", fname);
    if (linebuf)
        free (linebuf);
    free (fname);
    return 1;
}
else /* writers file exists, but this user not listed in it */
{
    handleIllegal:
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", fname);
    if (linebuf)
        free (linebuf);
    free (fname);
    return 0;
}
```
2340 / * Execute COMMAND in a subprocess with the appropriate funky things done. */

static struct filewrapper { filenet fds; } command; stdout, drain;

static int main_commandfd;

#define SERVERFLOWCONTROL

static int flowcontrol_pipe[2];

sendit /* SERVERFLOWCONTROL */

static void
do_vw_command (cmd_name, command)
char *cmd_name;
int (*command) PROTO((int argc, char *argv));
{
    static int

    / * The following file descriptors are set to -1 if that file is not
     * currently open.
     */

    / * Data on these pipes is a series of \n'-terminated lines. */

    int stdout_pipe[2];
    int stderr_pipe[2];

    / * Data on this pipe is a series of counted (see buf_send_counted)
     * packets. Each packet must be processed atomically (i.e. not
     * interleaved with data from stdout_pipe or stderr_pipe).
     */

    int protocol_pipe[2];

    int dev_null_fd = -1;

    int err;

    command_fd = -1;
    stdout_pipe[0] = -1;
    stdout_pipe[1] = -1;
    stderr_pipe[0] = -1;
    stderr_pipe[1] = -1;
    protocol_pipe[0] = -1;
    protocol_pipe[1] = -1;

    server_write_entries();

    if (print_pending_error ()
        goto free_args_and_return;

    /* Global ‘command_name’ is probably “server” right now – only
     * serve_cport() sets it to anything else. So we will use local
     * parameter ‘cmd_name’ to determine if this command is legal for
     * this user. */

    if (check_command_legal_p (cmd_name))
        {
            buf_output0 (buf_vecet, "% ");
            buf_output0 (buf_vecet, programs_name);
            buf_output0 (buf_vecet, " (server aborted): \\
                \n         ");
            buf_output0 (buf_vecet, cmd_name);
            buf_output0 (buf_vecet, "\n requires write access to the repository\n     ");
            goto free_args_and_return;
        }

    (void) server_notify ();

    /* We use a child process which actually does the operation. This
     * is so we can intercept its standard output. Even if all of CVS
     * were written to go to some special routine instead of writing
     * to stdout or stderr, we would still need to do the same thing
     * for the RCS commands.
     */

    if (pipe (stdout_pipe) < 0)
        {
            print_error (errno);
            goto error_exit;
        } else if (pipe (stderr_pipe) < 0)
        {
            print_error (errno);
            goto error_exit;
        }

    if (pipe (protocol_pipe) < 0)
        {
            print_error (errno);
            goto error_exit;
        }

    sendit /* SERVERFLOWCONTROL */
    if (pipe (flowcontrol_pipe) < 0)
{
    print_error (errno);
    goto error_exit;
}

set_nonblock_fd (flowcontrol Pipes[0]);
set_nonblock_fd (flowcontrol Pipes[1]);

sendfd /* SERVER FLOWCONTROL */

devnull_fd = CVS_OPEN (DEVNULL, O_WRONLY);
if (devnull_fd < 0)
{
    print_error (errno);
    goto error_exit;
}

/*@ We shouldn't have any partial lines from cvs_output and
   cvs_output.err, but we handle them here in case there is a bug. */
/*@ FIXME: appending a newline, rather than using "MT" as we
do in the child process, is probably not really a very good
way to "handle" them. */
if (! buf_empty_p (saved_output))
{
    buf_append_char (saved_output, 'n');
    buf_copy_lines (buf_to_net, saved_output, 'n');
}
if (! buf_empty_p (saved_output))
{
    buf_append_char (saved_output, 'n');
    buf_copy_lines (buf_to_net, saved_output, 'n');
}

/*@ Flush out any pending data. */
buf_flush (buf_to_net, 1);

/*@ Don't use fork; we're not going to exec(). */
command_pid = fork ();
if (command_pid < 0)
{
    print_error (errno);
    goto error_exit;
}
if (command_pid == 0)
{
    int exitstatus;

    /* Since we're in the child, and the parent is going to take
       care of packaging up our error messages, we can clear this
flag. */
    error_use_protocol = 0;

    protocol = dll_buffer_initialize (protocol_pipes[0], 0,
protocol_memory_error);

    /* At this point we should no longer be using buf_from_net and
       buf_to_net. Instead, everything should go through
protocol. */
    buf_to_net = NULL;
    buf_from_net = NULL;

    /* These were originally set up to use outbuf/memory_error.
Since we're now in the child, we should use the simpler
protocol_memory_error function. */
    saved_output->memory_error = protocol_memory_error;
    saved_output->memory_error = protocol_memory_error;

    if (dup2 (devnull_fd, STDIN_FILENO) < 0)
    {
        error (1, errno, "can't set up pipes");
    }
    if (dup2 (stdin_pipes[0], STDOUT_FILENO) < 0)
    {
        error (1, errno, "can't set up pipes");
    }
    if (dup2 (stderr_pipes[0], STDERR_FILENO) < 0)
    {
        error (1, errno, "can't set up pipes");
    }

    close (stdout_pipes[0]);
    close (stderr_pipes[0]);
    close (protocol_pipes[0]);

    stiodef SERVER FLOWCONTROL
    close (flowcontrol Pipes[1]);

sendfd /* SERVER FLOWCONTROL */

/*@ Set this in .bashrc if you want to give yourself time to attach
* to the subprocess with a debugger. */

if (getenv ("CVS_SERVER_SLEEP"))
{
    int secs = atoi (getenv ("CVS_SERVER_SLEEP"));
    sleep (secs);
}

exitstatus = (command) (argument_count, argument_vector);
/* Output any partial lines. If the client doesn't support
"MT", we just throw out the partial line, like old versions
of CVS did, since the protocol can't support this. */ 
if (supported_response("MT") || buf[empty]) (savedoutput)) 
{ 
buf_output0 (protocol, "MT text ");
buf_terminate (protocol, savedoutput); 
buf_output (protocol, "\n", 1); 
buf_terminate (protocol); 
} 
/* For now we just discard partial lines on stderr. I suspect
that CVS can't write such lines unless there is a bug. */ 
/* 
When we exit, that will close the pipes, giving an EOF to
the parent. */
exit (exitstatus); 

/* OK, sit around getting all the input from the child. */ 
{
struct buffer *stdoutbuf;
struct buffer *stderrbuf;
struct buffer *protocol_inbuf;
/* Number of file descriptors to check in select(). */ 
int num_to_check;
int count_needed = 0;
define SERVER FLOWCONTROL
int have_flowcontrolled = 0;
define / * SERVER FLOWCONTROL */
FD_ZERO (command_fds_to_drain_fds);
um_to_check = stdout_pipe[0];
FD_SET (stdout_pipe[0], &command_fds_to_drain_fds);
if (stdout_pipe[0] > num_to_check)
um_to_check = stdout_pipe[0];
FD_SET (stdout_pipe[0], &command_fds_to_drain_fds);
if (protocolpipe[0] > num_to_check)
um_to_check = protocolpipe[0];
FD_SET (protocolpipe[0], &command_fds_to_drain_fds);
if (STDOUT_FILENO > num_to_check)
um_to_check = STDOUT_FILENO;
max_command_fds = num_to_check;
/* 
File descriptors are numbered from 0, so num_to_check needs to 
be one larger than the largest descriptor. */
++num_to_check;
if (num_to_check > FD_SETSIZE)
{
buf_output0 (buf_to_net,
"E internal error: FD_SETSIZE not big enough.\n\nerror \n"); 
goto error_exit;
}
stdoutbuf = fdbuffer (stdoutpipe[0], 1,
input_memory_error);
stderrbuf = fdbuffer (stderrpipe[0], 1,
input_memory_error);
protocol_inbuf = fdbuffer (protocolpipe[0], 1,
input_memory_error);
set_nonblock (buf_to_net);
set_nonblock (stdoutbuf);
set_nonblock (stderrbuf);
set_nonblock (protocol_inbuf);

if (close (stdoutpipe[1]) < 0)
{ 
print_error (errno);
goto error_exit;
}
stdoutpipe[1] = -1;
if (close (stderrpipe[1]) < 0)
{ 
print_error (errno);
goto error_exit;
}
stderrpipe[1] = -1;
if (close (protocolpipe[1]) < 0)
{ 
print_error (errno);
goto error_exit;
}
protocol[pipe] = −1;

```c
ifdef SERVER_FLOWCONTROL
  if (close(flowcontrolled Pipes[0]) < 0)
    {  
      printf (errno);
      goto error_exit;
    }
  flowcontrolled Pipes[0] = −1;
endif */ SERVER_FLOWCONTROL */
```

```c
if (close (devnullfd) < 0)
  {  
    printf (errno);
    goto error_exit;
  }
devnullfd = −1;
while (stdout_pipes[0] >= 0  
  |  stderr_pipes[0] >= 0  
  |  protocol_pipes[0] >= 0)
```

```c
{  
  fd_set readfds;
  fd_set writefds;
  int numfds;

  ifdef SERVER_FLOWCONTROL
    int bufmemsize;
    
    /*  
     *  See if we are swamping the remote client and filling our VM.
     *  Tell child to hold off if we do.
     */
    
    /* bufmemsize = buf_count_mem (buf_to_net);
     *  if (have_flowcontrolled && (bufmemsize > SERVER_HI_WATER))
     *    {  
     *      if (write(flowcontrolled Pipes[1], "G", 1) == 1)  
     *        have_flowcontrolled = 1;
     *    }
     */
    else if (have_flowcontrolled && (bufmemsize < SERVER_LO_WATER))
    {  
      if (write(flowcontrolled Pipes[1], "G", 1) == 1)
        have_flowcontrolled = 0;
    }
endif /* SERVER_FLOWCONTROL */
```

```c
  FD_ZERO (&readfds);
  FD_ZERO (&writefds);
  if (! buf_empty_up (buf_to_net))
    FD_SET (STDOUT_FILENO, &writefds);
  if (stdout_pipes[0] >= 0)
    {  
      FD_SET (stdout_pipes[0], &readfds);
    }
  if (stderr_pipes[0] >= 0)
    {  
      FD_SET (stderr_pipes[0], &readfds);
    }
  if (protocol_pipes[0] >= 0)
    {  
      FD_SET (protocol_pipes[0], &readfds);
    }
```

```c
    /*  
     *  This process of selecting on the three pipes means that
     *  we might not get output in the same order in which it
     *  was written, thus producing the well-known
     *  "out-of-order" bug.  If the child process uses
     *  coneoutput and coneater, it will send everything on
     *  the protocol pipe and avoid this problem, so the
     *  solution is to use coneoutput and coneater in the
     *  child process. */
     */
     do  
     {  
      /*  
       *  This used to select on exceptions too, but as far
       *  as I know there was never any reason to do that and
       *  SCO doesn't let you select on exceptions on pipes. */
      numfds = select(num_to_check, &readfds, &writefds,  
                      &timeout, NULL);
      if (numfds < 0  
          && errno != EINTR)
        {  
          printf (errno);
          goto error_exit;
        }
    while (numfds < 0);
```

```c
    if (FD_ISSET (STDOUT_FILENO, &writefds))
    {  
      /*  
       *  What should we do with errors? syslog() them? */
       buffer output (buf_to_net);
    }
```
if (stdout_pipe[0] >= 0 &
    (FD_ISSET (stdout_pipe[0], &readfds)))
{
    int status;
    status = bufinputdata (stdoutbuf, (int *) NULL);
    buf_copy_lines (buf_to_net, stdoutbuf, 'E');
    if (status == -1)
        stdout_pipe[0] = -1;
    else if (status > 0)
    {
        print_error (status);
        goto error_exit;
    }
    /* What should we do with errors? syslog() them? */
    buf_send_output (buf_to_net);
}

if (stderr_pipe[0] >= 0 &
    (FD_ISSET (stderr_pipe[0], &readfds))
{
    int status;
    status = bufinputdata (stderrbuf, (int *) NULL);
    buf_copy_lines (buf_to_net, stderrbuf, 'E');
    if (status == -1)
        stderr_pipe[0] = -1;
    else if (status > 0)
    {
        print_error (status);
        goto error_exit;
    }
    /* What should we do with errors? syslog() them? */
    buf_send_output (buf_to_net);
}

if (protocol_pipe[0] >= 0 &
    (FD_ISSET (protocol_pipe[0], &readfds))
{
    int status;
    int count_read;
    int special;
    status = bufinputdata (protocol_inbuf, &count_read);
    if (status == -1)
        protocol_pipe[0] = -1;
    else if (status > 0)
    {
        print_error (status);
        goto error_exit;
    }
    /* We only call buf_copy_counted if we have read
     * enough bytes to make it worthwhile. This saves us
     * from continually recounting the amount of data we
     * have.
     */
    count_needed = count_read;
    while (count_needed <= 0)
    {
        count_needed = buf_copy_counted (buf_to_net,
            protocol_inbuf,
            &special);
        /* What should we do with errors? syslog() them? */
        buf_send_output (buf_to_net);
        /* If SPECIAL got set to -1, it means that the child
         * wants us to flush the pipe. We don't want to block
         * on the network, but we flush what we can. If the
         * client supports the 'F' command, we send it. */
        if (special == -1)
        {
            if (supported_response ('P'))
            {
                buf_append_char (buf_to_net, 'P');
                buf_append_char (buf_to_net, 'a');
            }
            cvs_flusherr ();
        }
    }
2790
2791 } } 
2792 */
2793 */ OK, we've gotten EOF on all the pipes. If there is
2794 */ anything left on stdoutbuf or stderrbuf (this could only
2795 */ happen if there was no trailing newline), send it over.
2796 */
2797 if (!buf_empty (stdoutbuf))
2798 {  
2799   buf_append_char (stdoutbuf, '\n');
2800   buf_copy_lines (bufset.stdoutbuf, stdoutbuf, 'E');
2801 }
2802 if (!buf_empty (stderrbuf))
2803 {  
2804   buf_append_char (stderrbuf, '\n');
2805   buf_copy_lines (bufset.stderrbuf, stderrbuf, 'E');
2806 }
2807 if (!buf_empty (protocol_input))
2808   buf_output0 (buf_to_net,  
2809   "E Protocol error: uncounted data discarded\n");
2810 
2811 errors = 0;
2812 while (command_pid > 0)
2813 {  
2814   int status;
2815   pid_t waited_pid;
2816   waited_pid = waitpid (command_pid, &status, 0);
2817 
2818   if (waited_pid < 0)  
2819     {  
2820     */
2821     * Intentionally ignoring EINTR. Other errors
2822     * can't happen."
2823     */
2824     continue;
2825     }
2826   
2827   if (WIFEXITED (status))
2828     errors += WEXITSTATUS (status);
2829   else
2830     {  
2831     int sig = WTERMSIG (status);
2832     char buf[50];
2833     
2834     /* This is really evil, because signals might be numbered
2835     * differently on the two systems. We should be using
2836     * signal names (either of the "Terminated" or the "SIGTERM"
2837     * variety). But cvs doesn't currently use libiberty... we
2838     * could roll our own.... FIXME.
2839     */
2840
2841     buf_output0 (buf_to_net, "E Terminated with fatal signal ");
2842     sprintf (buf, "%d
", sig);
2843     buf_output0 (buf_to_net, buf);
2844     
2845     /* Test for a core dump. Is this portable? */
2846     if (status & 0x80)  
2847     {  
2848       buf_output0 (buf_to_dir, "E Core dumped; preserving ");
2849       buf_output0 (buf_to_dir, orig_working_dir);
2850       if (waited_pid == command_pid)
2851         command_pid = -1;
2852     
2853     /* OK, we've waited for the child. By now all CVS locks are free
2854     * and it's OK to block on the network.
2855     */
2856     set_block (buf_to_net);
2857     buf_flush (buf_to_net, 1);
2858     
2859     if (errors)
2860      /* We will have printed an error message already. */
2861     buf_output0 (buf_to_net, "error \n");
2862     else
2863       buf_output0 (buf_to_net, "\n\n");
2864     goto free_args_and_return;
2865   }
2866 
2867   error_exit:
2868   if (command_pid > 0)
2869     kill (command_pid, SIGTERM);
while (command.pid > 0)
{
    pid_t waited_pid;
    waited_pid = waitpid (command.pid, (int *) 0, 0);
    if (waited_pid < 0 & & errno == EINTR)
        continue;
    if (waited_pid == command.pid)
        command.pid = -1;
}
close (devnull_fd);
close (protocol_pipe[0]);
close (protocol_pipe[1]);
close (stdin_pipe[0]);
close (stdin_pipe[1]);
close (stdout_pipe[0]);
close (stdout_pipe[1]);
free_args_and_return:
    /* Now free the arguments. */
{
    /* argument_vector[0] is a dummy argument, we don't mess with it. */
    char *cp;
    for (cp = argument_vector + 1;
        cp < argument_vector + argument_count;
        ++cp)
        free (*cp);
    argument_count = 1;
}
/* Flush out any data not yet sent. */
set_block (buf_fwdnet);
buf_flush (buf_l_to_net, 1);
return;

#ifndef SERVER_FLOWCONTROL
    /* Called by the child at convenient points in the server's execution for
    * the server child to block. ie: when it has no locks active.
    */
    void
server_pause_check()
{
    int paused = 0;
    char buf[1];

    while (read (flowcontrol_pipe[0], buf, 1) == -1)
    {
        if (*buf == 'P') /* Stop */
            paused = 1;
        else if (*buf == 'G') /* Go */
            paused = 0;
        else
            return; /* ??? */
    }
while (paused) {
    int numfds, numtocheck;
    fd_set fds;

    FD_ZERO (&fds);
    FD_SET (flowcontrol_pipe[0], &fds);
    numtocheck = flowcontrol_pipe[0] + 1;
    do {
        numfds = select (numtocheck, &fds, (fd_set *)0, (fd_set *)0, (struct timeval *)NULL);
        if (numfds < 0)
            & & errno != EINTR)
            printerror (errno);
        return;
    } while (numfds < 0);

    if (FD_ISSET (flowcontrol_pipe[0], &fds))
    {
        int got;

        while ((got = read (flowcontrol_pipe[0], buf, 1)) == 1)
        {
            if (*buf == 'P') /* Stop */
                paused = 1;
            else if (*buf == 'G') /* Go */
                paused = 0;
            else
                return; /* ??? */
        }
    }
}
*/
server.c – 639
2970    /* This assumes that we are using BSD or POSIX nonblocking I/O. System V
2971     nonblocking I/O returns zero if there is nothing to read. */
2972    if (got == 0)
2973      error (1, 0, “flow control EOF”);
2974    if (got < 0 && ! blockingerror (errno))
2975    {
2976      error (1, errno, “flow control read failed”);
2977    }
2978
2980
2981}
2982
2983/* SERVER_FLOWCONTROL */
2984
2985void server_output_not_carried_file (struct file_info *info, VersTS *vers)
2986{
2987  char* server = NULL;
2988  char* path = NULL;
2989
2990  if (server != NULL) {
2991    server = strchr (server + 1, ‘:’);
2992  }
2993  if (server != NULL) {
2994    server += 1;
2995    path = strchr (server, ‘:’);
2996  }
2997  if (path != NULL) {
2998    *path = ‘\’;
2999    path++;
3000  }
3001  if ((server == NULL) || (path == NULL)) {
3002    error (0, 0, “Remote entry invalid”);
3003  } else {
3004      /* Figure out the path on the other server where the file is located.
3005       * Replace our root with remote root and remove filename component
3006       */
3007      char* remote_path = xmalloc (strlen (vers->srcfile->path) + strlen (path) + 1);
3008      sprintf (remote_path, “%s%*d%s
%s
%s
%s
”,
3009                server, vers->srcfile->path + strlen (CVSroot_directory)),
3010                path, path, vers->srcfile->path + strlen (CVSroot_directory));
3011      *strchr (remote_path, ‘\’) = ‘\’;
3012      if (server_active) {
3013        printf (“Not-carried %s %s
%s
%s
%s
”,
3014              server, vers->srcfile->path + strlen (CVSroot_directory)),
3015              path, vers->tag, server, path, remote_path);
3016      } else {
3017        server_output_not_carried (info->fullname, vers->tag, server, path, remote_path);
3018      }
3019      free (remote_path);
3020    }
3021
3022    void server_output_not_carried (char* file, char* rev, char* server, char* root, char* repository)
3023    {
3024      buf_output0 (protocol, “Not-carried “);
3025      buf_output0 (protocol, file);
3026      buf_output0 (protocol, “\”);
3027      buf_output0 (protocol, rev);
3028      buf_output0 (protocol, “\"");
3029      buf_output0 (protocol, server);
3030      buf_output0 (protocol, “\"");
3031      buf_output0 (protocol, root);
3032      buf_output0 (protocol, “\"");
3033      buf_output0 (protocol, repository);
3034      buf_output0 (protocol, “\"");
3035      buf_send_counted (protocol);
3036    }
3037
3038    /* This variable commented in server.h. */
3039
3040  char *server_dir = NULL;
3041  static void output_dir PROTO((char *, char *));
3042
3043  static void
3044  output_dir (update_dir, repository)
3045  char *update_dir;
3046  char *repository;
3047  {
3048    if (server_dir != NULL)
3049    {
3050      buf_output0 (protocol, server_dir);
3051      buf_output0 (protocol, “*/”);
3052      if (update_dir[1] == ‘\’)
3053        buf_output0 (protocol, “\”);
3054      else
3055        buf_output0 (protocol, update_dir);
3056        buf_output0 (protocol, “*/”);
3057    }
Entries line that we are squirreling away to send to the client when we are ready.

static char *entries_line;

File which has been Scratch'd, we are squirreling away that fact to inform the client when we are ready.

static char *scratched_file;

The scratched file will need to be removed as well as having its entry removed.

static int kill_scratched_file;

int len;

if (options == NULL) options = "";

if (trace)
{
    (void) fprintf(stderr,
        "x-> server_register(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)\n";
    (server_active) ? 'B' : ' ', /* silly */
        name, version, timestamp ? timestamp : "", options,
        tag ? tag : "", date ? date : "",
        conflict ? conflict : "", repository);
}

if (entries_line != NULL)
{
    /*
    * If CVS decides to Register it more than once (which happens
    * on "cvs update foo/foo.c" where foo and foo.c are already
    * checked out), use the last of the entries lines Register'd.
    */
    free (entries_line);
}

if (scratched_file != NULL)
{
    free (scratched_file);
    scratched_file = NULL;
}

len = (strlen(name) + strlen(version) + strlen(options) + 80);

if (tag)
    len += strlen(tag);

if (date)
    len += strlen(date);

if (repository)
    len += strlen(repository) + 1;

entries_line = xmalloc (len);

sprintf(entries_line, "+%s/%s/%s/", name, version);

if (conflict != NULL)
{
    strcat(entries_line, "**");
}

strcat(entries_line, "+");

strcat(entries_line, options);

strcat(entries_line, "+");

if (tag != NULL)
{
    strcat(entries_line, "T");
strcat (entries_line, tag);
}
else if (date != NULL)
{
    strcat (entries_line, "D");
    strcat (entries_line, date);
}
strcat (entries_line, "/");
if (repository != NULL) {
    strcat (entries_line, repository);
}

void server_scratch (fname)
{
    char *fname;
    {
        /* I have reports of Scratch_Entry and Register both happening, in
        * two different cases. Using the last one which happens is almost
        * surely correct; I haven't tracked down why they both happen (or
        * even verified that they are for the same file).
        */
        if (entries_line != NULL)
        {
            free (entries_line);
            entries_line = NULL;
        }
    }
}

void server_scratch_entry_only ()
{
    kill_scratched_file = 0;
}

/* Print a new entries line, from a previous server_register. */
static void new_entries_line ()
{
    if (entries_line)
    {
        buf_output0 (protocol,
            "CVS server internal error: duplicate Scratch_Entry\n");
        buf_output counted (protocol);
        return;
    }
    scratched_file = xstrdup (fname);
    kill_scratched_file = 1;
}

static void serve_ci (arg)
{
    char *arg;
    do cvss command ("commit", commit);
}

static void checked_in_response (file, update_dir, repository)
{
    char *file;
    char *update_dir;
    char *repository;
    {
        if (supported_response ("Mode"))
        {
            struct stat sb;
            char *mode_string;

            if (CVS_STAT (file, &sb) < 0)
            {
                /* Not clear to me why the file would fail to exist, but it
                * was happening somewhere in the test suite. */
                if (existence_error (errno))
                    error (0, errno, "cannot stat %s", file);
            }
        }
    }
}
{  
  buf_output0 (protocol, "Mode ");
  mode_string = mode_to_string (preset_mode);
  buf_output0 (protocol, mode_string);
  buf_output0 (protocol, "u");
  free (mode_string);
}

buf_output0 (protocol, "Checked-in ");
output_dir (updated_dir, repository);
buf_output0 (protocol, file);
buf_output0 (protocol, "u", 1);
new_entries_line ();
}

void
server_checkedin (file, update_dir, repository)
char *file;
char *update_dir;
char *repository;

3260  
  {  
    if (noexec)
      return;
    if (scratched_file != NULL && entries_line == NULL)
    {  
      /*
       * This happens if we are now doing a “cvs remove” after a previous
       * “cvs add” (without a “cvs ci” in between).
       */
      buf_output0 (protocol, "remove-entry ");
      output_dir (updated_dir, repository);
      buf_output0 (protocol, file);
      buf_output0 (protocol, "u", 1);
      free (scratched_file);
      scratched_file = NULL;
    }
    else
    {  
      checkedin_response (file, update_dir, repository);
    }
  }

3280  
  buf_send_counted (protocol);
}

void
server_update_entries (file, update_dir, repository, updated)
char *file;
char *update_dir;
char *repository;
enum server_updated_arg4 updated;

3290  
  {  
    if (noexec)
      return;
    if (updated == SERVER_UPDATED)
      checkedin_response (file, update_dir, repository);
    else
    {  
      if (supported_response ("New-entry"))
        return;
      buf_output0 (protocol, "New-entry ");
      output_dir (updated_dir, repository);
      buf_output0 (protocol, file);
      buf_output0 (protocol, "u", 1);
      new_entries_line ();
    }
  }

3300  
  buf_send_counted (protocol);
}

static void
serve_update (arg)
char *arg;

3310  
  {  
    do_cvs_command ("update", update);
  }

static void
serve_diff (arg)
char *arg;

3320  
  {  
    do_cvs_command ("diff", diff);
  }

static void
serve_log (arg)
char *arg;

3320  
  {  
    do_cvs_command ("log", cvslog);
  }
static void serve_add (arg)
    char *arg;
    do cvscommand ("add", add);
}
static void serve_remove (arg)
    char *arg;
    do cvscommand ("remove", cvsremove);
}
static void serve_status (arg)
    char *arg;
    do cvscommand ("status", cvstatus);
}
static void serve_diff (arg)
    char *arg;
    do cvscommand ("rdiff", patch);
}
static void serve_tag (arg)
    char *arg;
    do cvscommand ("cvstag", cvtag);
}
static void serve_rtag (arg)
    char *arg;
    do cvscommand ("rtag", rtag);
}
static void serve_import (arg)
    char *arg;
    do cvscommand ("import", import);
}
static void serve_admin (arg)
    char *arg;
    do cvscommand ("admin", admin);
}
static void serve_history (arg)
    char *arg;
    do cvscommand ("history", history);
}
static void serve_release (arg)
    char *arg;
    do cvscommand ("release", release);
}
static void serve_watch_on PROTO ((char *));
static void serve_watch_on (arg)
    char *arg;
    do cvscommand ("watch_on", watch_on);
}
static void serve_watch_off PROTO ((char *));
static void serve_watch_off (arg)
    char *arg;
    do cvscommand ("watch_off", watch_off);
}
static void serve_watch_add PROTO ((char *));
static void serve_watch_add (arg)
    char *arg;
    { do_cvs_command ("watch_add", watch_add);
    }

static void serve_watch_remove PROTO ((char *));
static void serve_watch_add (arg)
    char *arg;
    { do_cvs_command ("watch_remove", watch_remove);
    }

3430 static void serve_watchers PROTO ((char *));
static void serve_watchers (arg)
    char *arg;
    { do_cvs_command ("watchers", watchers);
    }

3440 static void serve_editors PROTO ((char *));
static void serve_editors (arg)
    char *arg;
    { do_cvs_command ("editors", editors);
    }

3450 static int noop PROTO ((int, char **));
static int noop (argc, argv)
    int argc;
    char **argv;
    { return 0;
    }

3460 static void serve_noop PROTO ((char *));
static void serve_noop (arg)
    char *arg;
    { do_cvs_command ("noop", noop);
    }

3470 static void serve_init PROTO ((char *));
static void serve_init (arg)
    char *arg;
    { if (!isabsolute (arg))
        { if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text, "E Root %s must be an absolute pathname", arg);
            /* fall through to do_cvs_command which will return the actual error. */
        }
        set_local_cvsroot (arg);
        do_cvs_command ("init", init);
    }

3480 static void serve_annotate PROTO ((char *));
static void serve_annotate (arg)
    char *arg;
    { do_cvs_command ("annotate", annotate);
    }

3490 static void serve_co (arg)
    char *arg;
    { char *tempdir;
      int status;
      if (print_pending_error ())
        return;

3500 static void serve_noop PROTO ((char *));
static void serve_noop (arg)
    char *arg;
    { do_cvs_command ("noop", noop);
    }

3510 static void serve_init (arg)
    char *arg;
    { if (!isabsolute (arg))
        { if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text, "E Root %s must be an absolute pathname", arg);
            /* fall through to do_cvs_command which will return the actual error. */
        }
        set_local_cvsroot (arg);
        do_cvs_command ("init", init);
    }

3520 ** server.c – 645 **
if (lsdir(CVSADM))
{
    /* The client has not sent a "Repository" line. Check out
     * into a pristine directory.
     */
    tempdir = malloc(strlen(server) + 80);
    if (tempdir == NULL)
    {
        buf_output0(buf_to_net, "E Out of memory\n");
        return;
    }
    strcpy(tempdir, server);
    strcat(tempdir, "/checkout-dir");
    status = mkdir(tempdir);
    if (status != 0 && status != EEXIST)
    {
        buf_output0(buf_to_net, "E Cannot create \n");
        buf_output0(buf_to_net, tempdir);
        buf_append_char(buf_to_net, '\n');
        print_error(erno);
        free(tempdir);
        return;
    }
    if (CVS_CHDIR(tempdir) < 0)
    {
        buf_output0(buf_to_net, "E Cannot change to directory \n");
        buf_output0(buf_to_net, tempdir);
        buf_append_char(buf_to_net, '\n');
        print_error(erno);
        free(tempdir);
        return;
    }
    /* Compensate for server\n     *'s setting of command\n     *name.  
     *  [It probably doesn't matter if do_vna_command() gets \n     * "export" 
     * or "checkout", but we ought to be accurate where possible.] 
     */
    do_vna_command((strcmp(command_name, "export") == 0)
    {
        "export" : "checkout",
        checkout);
    }
}

static void
serve_export(arg)
char *arg;
{
    /* Tell checkout() to behave like export not checkout. */
    command_name = "export";
    serve_co(arg);
}

void
serve_copy_file(file, updatedir, repository, newfile)
char *file;
char *updatedir;
char *repository;
char *newfile;
{
    if (supported_response("Copy-file"))
        return;
    buf_output0(protocol, "Copy-file ");
    outputdir(updatedir, repository);
    buf_output0(protocol, file);
    buf_output0(protocol, "\n");
    buf_output0(protocol, "\n");
    /* See server.h for description. */
}

void
serve_mtime(finfo, vers)
struct file_info *finfo;
Vers = vers;
{
    char date[MAXDATELEN];
    int year, month, day, hour, minute, second;
    /* Note that these strings are specified in RFC822 and do not vary 
     * according to locale. */
    static const char *const month_names[] =
    assert(vers->vn_co != NULL);
    

if (supported_response("Mod-time"))
    return;

/* The only hard part about this routine is converting the date formats. In terms of functionality it all boils down to the call to RUS_getmtime. */
if (RUS_getmtime (info->vers, vers, &date, 0) == (time_t) -1)
    /* FIXME? should we be printing some kind of warning? For one thing I'm not 100% sure whether this happens in non-error circumstances. */
    return;

    sscanf (date, "%d %s %d %d:%d:%d -0000", day, month, &year, &hour, &minute, &second);
    sprintf (buf, "Mod-time ", date);
    buf_output0 (protocol, "Mod-time ");
    buf_output0 (protocol, "\n");
}

/* See server.h for description. */

if (defined (USE_PROTOTYPES) ? USE_PROTOTYPES : defined (__STDC__))
/* Need to prototype because mode_t might be smaller than int. */
void server_updated 
{
    struct file_info *finfo, VersTS vers,
    enum server_prevarg updated,
    mode_t mode,
    unsigned char *checksum,
    struct buffer *filebuf

    else
    void server_updated (finfo, vers, updated, mode, checksum, filebuf)
    struct file_info *finfo, VersTS vers,
    enum server_prevarg updated,
    mode_t mode;

    else
    unsigned char *checksum;
    struct buffer *filebuf;

    send

    {
    if (noexec)
        {
            /* Hmm, maybe if we did the same thing for entries_file, we could get rid of the kludges in server_register and server_scratch which refrain from warning if both Scratch_tEntry and Register get called. Maybe. */
            if (scratched_file)
                {
                    free (scratched_file);
                    scratched_file = NULL;
                }
        return;
    }

    if (entries_line != NULL && scratched_file != NULL)
    {
        FILE *f;
        struct buffer_data *list, *last;
        unsigned long size;
        char size[10][16];

        if (filebuf != NULL)
            {
                size = buf_length (filebuf);
                if (mode != (mode_t) -1)
                    error (1, 0, "\n");

            } else
                {
                    struct stat sb;

                    if (CVS_STAT (info->vers, &sb) < 0)
                        {
                            if (existence_error (errno))
                                {
                                    /* If we have a sticky tag for a branch on which the file is dead, and cvs update the directory, it gets a T_CHECKOUT but no file. So in this case just forget the whole thing. */
                                    free (entries_line);
                                    entries_line = NULL;
                                    goto done;
                                }
                    error (1, errno, "reading %s", info->vers);
```c
3690  
size = stat_size;
if (mode == (mode ^ 1) - 1)
{
    /* FIXME: When we check out files the mask of the server (set in .bashrc if rsh is in use) affects
what mode we send, and it shouldn't. */
    mode = stat_mode;
}
}
3700  
if (checksum != NULL)
{
    static int checksum_supported = -1;
    if (checksum_supported == -1)
    {
        checksum_supported = supported_response("Checksum");
    }
    
    if (checksum_supported)
    {
        int i;
        char buf[3];
        buf_output0 (protocol, "Checksum ");
        for (i = 0; i < 16; i++)
        {
            sprintf (buf, "%d", (unsigned int) checksum[i]);
            buf_output0 (protocol, buf);
        }
        buf_append_char (protocol, '\n');
    }
}
3710  
if (updated == SERVER_UPDATED)
{
    Node *node;
    Entnode *entnode;
    
    if (!supported_response("Created")
        && supported_response("Update-existing"))
        buf_output0 (protocol, "Updated ");
    else
    {
        assert (vers != NULL);
        if (vers->tx_user == NULL)
            buf_output0 (protocol, "Created ");
        else
            buf_output0 (protocol, "Update-existing ");
    }
    
3740  
/* Now munge the entries to say that the file is unmodified,
in case we end up processing it again (e.g. modules3-6
in the testsuite). */
node = findnodefn (info->entries, info->file);
entnode = (Entnode *) node->data;
free (entnode->timestamp);
entnode->timestamp = xstrdup ("*");
}
else if (updated == SERVER_MERGED)
    buf_output0 (protocol, "Merged ");
else if (updated == SERVER_PATCHED)
    buf_output0 (protocol, "Patched ");
else if (updated == SERVER_RCS_DIFF)
    buf_output0 (protocol, "Rcs-diff ");
else
    abort (1);
output_dir (info->update_dir, info->repository);
buf_output0 (protocol, info->file);
buf_output (protocol, "\n", 1);
3750  
new_entries_line ();
{
    char *mode_string;
    mode_string = mode_to_string (mode);
    buf_output0 (protocol, mode_string);
    buf_output0 (protocol, "\n");
    free (mode_string);
}
3770  
list = last = NULL;
if (size > 0)
{
    /* Throughout this section we use binary mode to read the
file we are sending. The client handles any line ending
translation if necessary. */
```
if (gsize >= level)
  /*
   * For really tiny files, the gzip process startup
   * time will outweigh the compression savings. This
   * might be computable somehow; using 100 here is just
   * a first approximation.
   */
  if (gsize > 100)
    {
    int status, fd, gzip_status;
    pid_t gzip_pid;
  }

  /* Callers must avoid passing us a buffer if
     file_gzip_level is set. We could handle this case,
     but it's not worth it since this case never arises
     with a current client and server. */
  if (filebuf != NULL)
    error (1, 0, "\nCVS server internal error: unhandled case in server_updated()");

  fd = CVS_OPEN (finfo->file, O_RDONLY | OPEN_BINARY, 0);
  if (fd < 0)
    error (1, errno, "reading %s", finfo->fullname);
  fd = fdenopen (fd, "rb");
  status = buf_read_file_to_eof (fd, &list, &last);
  size = buf->chain_length (list);
  if (status == -2)
    {protocol->memory_error} (protocol);
  else if (status != 0)
    error (1, errno, "reading %s", finfo->fullname);
  if (fclose (fd) == EOF)
    error (1, errno, "reading %s", finfo->fullname);
  if (waitpid (gzip_pid, &gzip_status, 0) == -1)
    error (1, errno, "waiting for gzip process %ld",
           (long) gzip_pid);
  else if (gzip_status != 0)
    error (1, 0, "gzip exited %d", gzip_status);
  /* Prepending length with "z" is flag for using gzip here. */
  buf->output0 (protocol, "z");
}

else if (filebuf == NULL)
  {
  long status;
  fd = CVS_OPEN (finfo->file, "rb");
  if (fd == NULL)
    error (1, errno, "reading %s", finfo->fullname);
  status = buf->read_file (fd, size, &list, &last);
  if (status == -2)
    {protocol->memory_error} (protocol);
  else if (status != 0)
    error (1, errno, "reading %s", finfo->fullname);
  if (fclose (fd) == EOF)
    error (1, errno, "reading %s", finfo->fullname);
  }

sprintf (size->text, "%lu", size); 
buf->output0 (protocol, size->text);

if (filebuf == NULL)
  buf->append_data (protocol, list, last);
else
  {
  buf->append_buffer (protocol, filebuf);
  buf->free (filebuf);
  } /* Note we only send a newline here if the file ended with one. */

/*
 * Avoid using up too much disk space for temporary files.
 * A file which does not exist indicates that the file is up-to-date,
 * which is now the case. If this is SERVER_MERGED, the file is
 * not up-to-date, and we indicate that by leaving the file there.
 * I'm thinking of cases like "cvs update foo/foo.c foo".
 */
if (updated == SERVER_UPDATED || updated == SERVER_PATCHED || updated == SERVER_RCS_DIFF)
  {filebuf == NULL
   /* But if we are joining, we'll need the file when we call
      join_file. */
   &filebuf == NULL
   else if (scratched_file == NULL & entries_line == NULL)
     {
if (strcmp(scratchfile, finfo->file) != 0)
  error (1, 0,
       "CVS server internal error: 'sa' vs. 'sa' scratched",
       scratchfile, finfo->file);
free(scratchfile);
scratchfile = NULL;
if (!scratchfile) {
  buf_output0 (protocol, "%removed ");
} else {
  buf_output0 (protocol, "%remove-entry ");
  output_dir (finfo->update_dir, finfo->repository);
  buf_output0 (protocol, finfo->file);
  buf_output0 (protocol, "sa", 1);
}
else if (scratchfile == NULL && entries_line == NULL) {
  /* This can happen with death support if we were processing
     a dead file in a checkout.
     */
  return;
}
else {                      
  error (1, 0,                      
       "CVS server internal error: Register "staff", Scratch_Entry, \n"
       protocol);                     
  done;
}

/* Return whether we should send patches in RCS format. */

int server_use_rcs_diff ()
{
  return supported_response ("Rcs-diff");
}

void server_set_entstat (update_dir, repository)
char *update_dir;
char *repository;
{
  static int set_static_supported = -1;
  if (set_static_supported == -1)
    set_static_supported = supported_response ("Set-static-directory");
  if (set_static_supported) return;
  buf_output0 (protocol, "Set-static-directory ");
  output_dir (update_dir, repository);
  buf_output0 (protocol, "sa");
}

void server_clear_entstat (update_dir, repository)
char *update_dir;
char *repository;
{
  static int clear_static_supported = -1;
  if (clear_static_supported == -1)
    clear_static_supported = supported_response ("Clear-static-directory");
  if (clear_static_supported) return;
  if (noexec) return;
  buf_output0 (protocol, "Clear-static-directory ");
  output_dir (update_dir, repository);
  buf_output0 (protocol, "sa");
  buf_send_counted (protocol);
}

void server_set_sticky (update_dir, repository, tag, date, nonbranch)
char *update_dir;
char *repository;
char *tag;
char *date;
int nonbranch;
{
  static int set_sticky_supported = -1;
  assert (update_dir != NULL);
  if (set_sticky_supported == -1)
    set_sticky_supported = supported_response ("Set-sticky");
  if (set_sticky_supported) return;
  if (noexec)
return;

if (tag == NULL && date == NULL)
{
    buf_output0 (protocol, "Clear-sticky ");
    output_dir (update_dir, repository);
    buf_output0 (protocol, "xml");
}
else
{
    buf_output0 (protocol, "Set-sticky ");
    output_dir (update_dir, repository);
    buf_output0 (protocol, "xml");
    if (tag != NULL)
    {
        if (nonbranch)
            buf_output0 (protocol, "N");
        else
            buf_output0 (protocol, "P");
    } else
    {
        buf_output0 (protocol, "P");
        buf_output0 (protocol, date);
    }
    buf_output0 (protocol, "xml");
    buf_send_counted (protocol);
}

struct template_proc_data
{
    char *update_dir;
    char *repository;
};
/* Here as a static until we get around to fixing ParseInfo to pass along
   a void * for it. */

static struct template_proc_data *tpd;

static int
template_proc (repository, template)
char *repository;
char *template;
{
    FILE *fp;
    char buf[1024];
    size_t n;
    struct stat sb;
    struct template_proc_data *data = tpd;

    if (supported_response ("Template"))
    /* Might want to warn the user that the rcsinfo feature won't work. */
    return 0;
    buf_output0 (protocol, "Template ");
    output_dir (data->update_dir, data->repository);
    buf_output0 (protocol, "xml");
    fp = CVS_FOPEN (template, "rb");
    if (fp == NULL)
    {
        error (0, errno, "Couldn't open rcsinfo template file %s", template);
        return 1;
    }
    if (fstat (fileno (fp), &sb) < 0)
    {
        error (0, errno, "cannot stat rcsinfo template file %s", template);
        return 1;
    }
    sprintf (buf, "%d\n", (long) sb.at_size);
    buf_output0 (protocol, buf);
    while (fseek (fp))
    {<tex>if (ferror (fp))
    {<tex>error (0, errno, "cannot read rcsinfo template file %s", template);
        (void) fclose (fp);
        return 1;
    }
    n = fread (buf, 1, sizeof buf, fp);
    buf_output (protocol, buf, n);
    if (ferror (fp))
    {<tex>error (0, errno, "cannot read rcsinfo template file %s", template);
        (void) fclose (fp);
        return 1;
    }
    }
    if (fclose (fp) < 0)
    {<tex>error (0, errno, "cannot close rcsinfo template file %s", template);
        return 0;
    }
}

void
server_template (update_dir, repository)
static void serve_gzip_contents (arg)
    char *arg;
    { 
        int level;
        level = atoi (arg);
        if (level == 0)
            level = 6;
        file_gzip_level = level;
    }

static void serve_gzip_stream (arg)
    char *arg;
    { 
        int level;
        level = atoi (arg);
        if (level == 0)
            level = 6;
        /* All further communication with the client will be compressed. */

        buf_to_net = compress_buffer_initialize (buf_to_net, 0, level,
            buf_to_net->memory_error);
        buf_from_net = compress_buffer_initialize (buf_from_net, 1, level,
            buf_from_net->memory_error);
    }

    /* Tell the client about RCS options set in CVSROOT/cowwrappers. */
    static void serve_wrapper_send_options (arg)
        char *arg;
        { 
            /* Actually, this is kind of a drawkab-sse: the client wants
                 * verbatim lines from a cowwrappers file, but the server has
                 * already parsed the cowwrappers file into the wrap_list struct.
                 * Therefore, the server loops over wrap_list, unparsing each
                 * entry before sending it.
            */
            char *wrapper_line = NULL;

        wrap_setup ();
        for (wrap_unparse_rcs_options (&wrapper_line, 1);
            wrapper_line;
                wrap_unparse_rcs_options (&wrapper_line, 0))
        { 
            buf_output0 (buf_to_net, "wrapper-rsoption ");
            buf_output0 (buf_to_net, wrapper_line);
            buf_output0 (buf_to_net, "012");
            free (wrapper_line);
        }

        buf_output0 (buf_to_net, "ok");

        /* The client is waiting for us, so we better send the data now. */
        /* buf_flush (buf_to_net, 1); */
    }

static void serve_ignore (arg)
    char *arg;
    { 
    /* Just ignore this command. This is used to support the
         * update-patches command, which is not a real command, but a signal
         * to the client that update will accept the -u argument.
    */
    }

static int expand_proc (pargc, argv, where, mwhere, mfile, shorten,
    local_specified, omodule, msg)
    int *pargc;
    char **argv;
    char *where;
    char *mwhere;
    char *mfile;
    int shorten;
int local_specified;
char *module;
char *msg;

int i;
char *dir = argv[0];

/* If mwhere has been specified, the thing we're expanding is a module - just return its name so the client will ask for the right thing later. If it is an alias or a real directory, mwhere will not be set, so send out the appropriate expansion. */

if (mwhere != NULL)
{
    buf_output0 (buf_to_net, "Module-expansion *");
    if (server_dir != NULL)
    {
        buf_output0 (buf_to_net, server_dir);
        buf_output0 (buf_to_net, "/");
    }
}
else
{
    /* We may not need to do this anymore - check the definition
     of aliases before removing */
    if (+pargc == 0)
    {
        buf_output0 (buf_to_net, "Module-expansion *");
        if (server_dir != NULL)
        {
            buf_output0 (buf_to_net, server_dir);
            buf_output0 (buf_to_net, "/");
        }
    }
    else
    {
        for (i = 1; i < +pargc; ++i)
        {
            buf_output0 (buf_to_net, "Module-expansion *");
            if (server_dir != NULL)
            {
                buf_output0 (buf_to_net, server_dir);
                buf_output0 (buf_to_net, "/");
            }
        }
    }
}
return 0;

static void
serve_expand_modules (arg)
{
    int i;
    int err;
    DRM *db;
    err = 0;

    server_expanding = 1;
    db = open_module ();
    for (i = 1; i < argument_count; i++)
    {
        CHECKOUT, "Getting", expandproc,
        NULL, 0, 0, 0,
        (char *) NULL);
        close_module (db);
    }
    server_expanding = 0;

    *argument_vector[0] is a dummy argument, we don't mess with it. */
    char *cp;
    for (cp = argument_vector + 1;
        cp < argument_vector + argument_count;
        ++cp)
        free (*cp);
argument, count = 1;
}
if (err)
  /* We will have printed an error message already. */
  buf_output0 (buf_to_net, "error \n");
else
  buf_output0 (buf_to_net, "\n\n");
/* The client is waiting for the module expansions, so we must
send the output now. */
buf_fflush (buf_to_net, 1);
}

void
server_prog (dir, name, which)
  char *dir;
  char *name;
  enum progs which;
{
  if (!supported_response ("Set-checkin-prog"))
    
warning: this client does not support -i or -u flags in the modules file. \n");
return;
}
switch (which)
{
  case PROG_CHECKIN:
    buf_output0 (buf_to_net, "Set-checkin-prog \n");
    break;
  case PROG_UPDATE:
    buf_output0 (buf_to_net, "Set-update-prog \n");
    break;
}
buf_output0 (buf_to_net, dir);
buf_append_char (buf_to_net, ";\n");
buf_output0 (buf_to_net, "\n");
}

static void
server_checkin_prog (arg)
  char *arg;
{
  FILE *f;
  f = CVS_FOPEN (CVSADM_CIPROG, "w");
  if (f == NULL)
    
pending_error = errno;
    if (alloc_pending (80 + strlen (CVSADM_CIPROG)))
      sprintf (pending_error_text, "E cannot open \n", CVSADM_CIPROG);
return;
}
if (fprintf (f, "%s", arg) < 0)
  
pending_error = errno;
    if (alloc_pending (80 + strlen (CVSADM_CIPROG)))
      sprintf (pending_error_text, "E cannot write to \n", CVSADM_CIPROG);
return;
}
if (fclose (f) == EOF)
  
pending_error = errno;
    if (alloc_pending (80 + strlen (CVSADM_CIPROG)))
      sprintf (pending_error_text, "E cannot close \n", CVSADM_CIPROG);
return;
}

static void
server_update_prog (arg)
  char *arg;
{
  FILE *f;
  /* Before we do anything we need to make sure we are not in readonly
  mode. */
  if ![check_command_legalp ("commit")]
    
/* I might be willing to make this a warning, except we lack the
  machinery to do so. */
    if (alloc_pending (80))
      sprintf (pending_error_text, "$\n");
    if_flag = modules not allowed in readonly mode$);
return;
}
if (f == NULL)


```c
{ pending_error = errno;
  if (allow_pending (80 + strlen (CVSADM_UPROG)))
    sprintf (pending_error_text, "E cannot open %s", CVSADM_UPROG);
  return;
}
if (fprintf (f, "%s\n", arg) < 0)
{ pending_error = errno;
  if (allow_pending (80 + strlen (CVSADM_UPROG)))
    sprintf (pending_error_text, "E cannot write to %s", CVSADM_UPROG);
  return;
}
if (fclose (f) == EOF)
{ pending_error = errno;
  if (allow_pending (80 + strlen (CVSADM_UPROG)))
    sprintf (pending_error_text, "E cannot close %s", CVSADM_UPROG);
  return;
}
}

static void serve_valid_requests PROTO((char *arg));
#endif /* SERVER_SUPPORT */
if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)
  /* * Parts of this table are shared with the client code,
   * but the client doesn't need to know about the handler
   * functions.
   */

struct request requests[] =
{ }
#endif /* SERVER_SUPPORT */
define REQ_LINE(n, f, s) {n, f, s}
else
define REQ_LINE(n, f, s) {n, s}
#endif

REQ_LINE("root", serve_root, rq_essential),
REQ_LINE("valid responses", serve_valid_responses, rq_essential),
REQ_LINE("valid requests", serve_valid_requests, rq_essential),
REQ_LINE("repository", serve_repository, rq_optional),
REQ_LINE("directory", serve_directory, rq_essential),
REQ_LINE("max-dotdot", serve_max_dotdot, rq_optional),
REQ_LINE("static directory", serve_static_directory, rq_optional),
REQ_LINE("sticky", serve_sticky, rq_optional),
REQ_LINE("check-in", serve_checkin, rq optional),
REQ_LINE("update-prog", serve_update_prog, rq optional),
REQ_LINE("entry", serve_entry, rq_essential),
REQ_LINE("log", serve_log, rq_optional),
REQ_LINE("log", serve_log, rq_optional),
REQ_LINE("modified", serve_modified, rq_essential),
REQ_LINE("remote-revision", serve_remote_revision, rq_optional),
REQ_LINE("unmodified", serve_unmodified, rq optional),
REQ_LINE("skip-stream", serve_skip_stream, rq optional),
REQ_LINE("server-send-receive-options",)
  serve_send_receive_options, rq_optional),
REQ_LINE("set", serve_set, rq_optional),
endif

ENCRIPTION
#ifdef HAVE_KERBEROS
REQ_LINE("kerberos-encrypt", serve_kerberos_encrypt, rq optional),
#endif
#ifdef HAVE_GSSAPI
REQ_LINE("gssapi-encrypt", serve_gssapi_encrypt, rq optional),
#endif
#endif
#endif
#endif
```
```c
REQ_LINE("log", serve_log, rq Optional),
REQ_LINE("add", serve_add, rq Optional),
REQ_LINE("remove", serve_remove, rq Optional),
REQ_LINE("update-patches", serve_ignore, rq Optional),
REQ_LINE("skip-file-contents", serve_annotate, rq Optional),
REQ_LINE("status", serve_status, rq Optional),
REQ_LINE("status", serve_status, rq Optional),
REQ_LINE("offline", serve_offline, rq Optional),
REQ_LINE("online", serve_online, rq Optional),
REQ_LINE("init", serve_init, rq Optional),
REQ_LINE("stop", serve_stop, rq Optional),
REQ_LINE("start", serve_start, rq Optional),
REQ_LINE("status", serve_status, rq Optional),
REQ_LINE("annotate", serve_annotate, rq Optional),
REQ_LINE("noop", servenoop, rq Optional),
REQ_LINE(NULL, NULL, rq Optional)
}
#endif
#endif SERVER_SUPPORT or CLIENT_SUPPORT */
#endif SERVER_SUPPORT

4440 static void
serve_valid_requests (arg)
char *arg;
{
 struct request *rq;
 if (print_pending_error ()
 return;
 bu_output0 (buf_to_net, "Valid-requests");
 for (rq = requests; rq->name != NULL, rq++)
 { if (rq->func != NULL)
 { bu_output0 (buf_to_net, "*");
 bu_output0 (buf_to_net, rq->name); } }
 bu_output0 (buf_to_net, "*\n\n");
/* The client is waiting for the list of valid requests, so we must send the output now. */
4450 bu_flush (buf_to_net, 1);
}
#endif
#endif
/** Delete temporary files. SIG is the signal making this happen, or */
/* 0 if not called as a result of a signal. */
/* static int commandโหลid = lead;
static void wait_lsig (sig)
4470 int sig;
{ int status;
pid_t s = wait (&status);
if (s == command lrpid)
 command_pids[lead] = lead++;
}
#endif
void
4480 serve_cleanup (sig)
int sig;
{ /* Do "rm -rf" on the temp directory. */
int status;
int save_hoseec;
if (buf_to_net != NULL)
{ /* FIXME: If this is not the final call from server, this */
could deadlock, because the client might be blocked writing */
to us. This should not be a problem in practice, because */
we do not generate much output when the client is not */
waiting for it. */
set_block (buf_to_net);
buf_flush (buf_to_net, 1);
/* The calls to buf_showdown are currently only meaningful */
/* when we are using compression. First we shut down */
4490
```

BUF_FROM_NET. That will pick up the checksum generated when the client shuts down its buffer. Then, after we have generated any final output, we shut down BUF_TO_NET: */

status = buf_ut shutdown (buf FROM_net);
if (status != 0)
{ error (0, status, "shutting down buffer from client");
  buf_ut flush (buf FROM_net, 1);
}

if (!delete) { void) buf_ut shutdown (buf FROM_net);
return;
}

/* What a bogus kludge. This disgusting code makes all kinds of assumptions about SunOS, and is only for a bug in that system. */

if (command.pid > 0) {
  /* To avoid crashes on SunOS due to bugs in SunOS tmpfs triggered by the use of rename() in RCS, wait for the subprocess to die. Unfortunately, this means draining output while waiting for it to unblock the signal we sent it. Yuck! */
  int status;
  pid_t r;
  signal (SIGCHLD, wait_sig);
  if (sig) {
    /* Perhaps SIGTERM would be more correct. But the child process will delay the SIGINT delivery until its own children have exited. */
    kill (command.pid, SIGINT);
    /* The caller may also have sent a signal to command.pid, so always try waiting. First, though, check and see if it's still there... */

    do waitpid:
    r = waitpid (command.pid, &status, WNOHANG);
    if (r == 0)
      ;
    else if (r == command.pid)
      command.pid LNG dead++;
    else if (r == -1)
      switch (errno)
        { case ECHILD:  
          command.pid LNG dead++;
          break;
        case EINTR:  
          goto do waitpid;
        }
    else /* waitpid should always return one of the above values */
      abort ();
    while (command.pid LNG dead)
    {
      struct timeval timeout;
      struct fd_set readfds;
      char buf[100];
      int i;
      /* Use a non-zero timeout to avoid eating up CPU cycles. */
      timeout.tv_usecs = 2;
      timeout.tv_sec = 0;
      readfds = command.FD LNG to_read;
      switch (select (max(command_fd + 1, &readfds.fds,
                    (fd_set *)0, (fd_set *)0, &timeout))
        { case -1:
          if (errno != EINTR)
            abort ();
          break;
        case 0:
          /* timeout */
          break;
        case 1:
          for (i = 0; i <= max_command_fd; i++)
            { if (!FD_ISSET (i, &readfds.fds))
              continue;
              /* this fd is non-blocking */
              while (read (i, buf, sizeof (buf)) >= 1)
                ;
            }
          break;
default:
  abort ();
}
}
#endif

CVS \CHDIR (Tmpdir);
/** Temporarily clear noexec, so that we can clean up our temp directory
 * regardless of it (this could more cleanly be handled by moving
 * the noexec check to all the unlink_filedir callers from
 * unmark_filedir itself). */

unlink_filedir = noexec;
noexec = 0;
/** FIXME: Would be nice to not ignore errors. But what should we do?
 * We could try to do this before we shut down the network connection,
 * and try to notify the client (but the client might not be waiting
 * for responses). We could try something like syslog() or our own
 * log file. */
unlink_filedir (orig_server_temp_dir);
noexec = save_noexec;

if (buf_to_net != NULL)
  (void) buf_shutdown (buf_to_net);

int server_active = 0;
int server_expanding = 0;

int server (argc, argv)
{ int arg;
  char **argv;

  if (argc == -1)
    {
      static const char *const msg[] =
        { "Usage: %s
",
          " Normally invoked by a cvs client on a remote machine.
"
        NULL}
    ;
    usage (msg);
  } /* Ignore argc and argv. They might be from .csrc. */

  buf_to_net = fd_buffer_initialize (STDOUT_FILENO, 0,
outbuf_memory_error);
  buf_from_net = stdin_buffer_initialize (stdin, 1,outbuf_memory_error);

  saved_output = buf_nonio_initialize (outbuf_memory_error);
  saved_output = buf_nonio_initialize (outbuf_memory_error);

  /* Since we're in the server parent process, error should use the
   * protocol to report error messages. */
  error_use_protocol = 1;

  /* OK, now figure out where we stash our temporary files. */

  char *p;

  /* The code which wants to chdir into server_temp_dir is not set
   * up to deal with it being a relative path. So give an error
   * for that case. */
  if (isabsolute (Tmpdir))
    {
      pending_error_text = malloc (80 + strlen (Tmpdir));
      if (pending_error_text == NULL)
        { pending_error = ENOMEM;
        }
    }
  else
    { sprintf (pending_error_text,
        "E Value of %s for TMPDIR is not absolute", Tmpdir);
    } /* FIXME: we would like this error to be persistent, that
   * is, not cleared by print_pending_error. The current client
   * will exit as soon as it gets an error, but the protocol spec
   * does not require a client to do so. */

  else
    { int status;
      noexec check to all the unmark_filedir callers from
      unmark_filedir itself.
    }

    else
    { Strictly speaking, we're not supposed to output anything
/* now. But we're about to exit(), give it a try. */

error EXIT_FAILURE;

} /* I'm doing this manually rather than via error_exit() because
I'm not sure whether we want to call server_cleanup. 
Needs more investigation... */

#define SYSTEM_CLEANUP

/* Hook for OS-specific behavior, for example socket subsystems on NT and OS2 or dealing with windows
and arguments on Mac. */

#endif

#endif

exit(EXIT_FAILURE);

strcpy(server_temp_dir, Tmpdir);

/* Remove a trailing slash from TMPDIR if present. */
p = server_temp_dir + strlen(server_temp_dir) - 1;
if (p[-1] == '/')
  p[-1] = '0';

/* I wanted to use cvs-serv/PID, but then you have to worry about
* the permissions on the cvs-serv directory being right. So
* use cvs-serv/PID.
*/

strcat(server_temp_dir, "\/*cvs-serv*/

#define CHMOD_BROKEN

else

  if (chmod (server_temp_dir, S_IRWXU) < 0)
    
      /* E cannot change permissions on temporary directory*/
      pending_error = save_errno;
    
#endif

#define SIGHUP

(void) SIGCHID (SIGHUP, server_cleanup);

#endif

#define SIGINT

(void) SIGCHID (SIGINT, server_cleanup);

#endif

#define SIGQUIT

(void) SIGCHID (SIGQUIT, server_cleanup);

#endif

#define SIGPIPE

(void) SIGCHID (SIGPIPE, server_cleanup);

#endif

#define SIGTERM

(void) SIGCHID (SIGTERM, server_cleanup);

#endif

/* Now initialize our argument vector (for arguments from the client). */

/* Small for testing. */

argument_vector_size = 1;

if (argument_vector == NULL)
  
    /* Strictly speaking, we're not supposed to output anything

}
* now. But we’re about to exit(); give it a try.

/* */
printf("E Fatal server error, aborting.
error ENOMEM Virtual memory exhausted.");

/* I’m doing this manually rather than via error
exit () because I’m not sure whether we want to call server
CLEANUP. Needs more investigation... */

#define SYSTEM_CLEANUP
/* Hook for OS-specific behavior, for example socket subsystems on
NT and OS2 or dealing with windows and arguments on Mac. */
SYSTEM_CLEANUP ();
#endif

exit (EXIT_FAILURE);
}

argument_count = 1;
/* This gets printed if the client supports an option which the
server doesn’t, causing the server to print a usage message.

FIXME: probably should be using program_name here.
FIXME: just a nit, I suppose, but the usage message the server
prints isn’t literally true—it suggests “cvs server” followed
by options which are for a particular command. Might be nice to
say something like “client apparently supports an option not supported
by this server” or something like that instead of usage message. */
argument_vector[0] = "cvs server";

while (1)
{
char *cmd, *orig_cmd;
struct request *rq;
int status;

status = buf_readline (buf_from_net, &cmd, (int *) NULL);
if (status == -2)
{
  buf_output0 (buf_to_net, "E Fatal server error, aborting.
error ENOMEM Virtual memory exhausted.");
  break;
}
if (status != 0)
{
  break;
}
orig_cmd = cmd;
for (rq = requests; rq->name != NULL; ++rq)
{
  if (strncmp (cmd, rq->name, strlen (rq->name))) != 0)
    {
    int len = strlen (rq->name);
    if (cmd[len] == ‘\0’)
      cmd += len;
    else if (cmd[len] == ‘.’)
      cmd += len + 1;
    else
      { /* The first len characters match, but it’s a different
        command. e.g. the command is “cooperate” but we matched
        “c0”. */
        continue;
      }
  }
  else
    { (*rq->func) (cmd); break;
  }
if (rq->name == NULL)
  {
    if (print_pending_error ())
      { buf_output0 (buf_to_net, "error unrecognized request ‘\’");
        buf_output0 (buf_to_net, cmd);
        buf_append_char (buf_to_net, ‘\’);}
    buf_append_char (buf_to_net, ‘\n’);
  }
  free (orig_cmd);
}
server_cleanup (0);
return 0;
}

if defined (HAVE_KERBEROS) || defined (AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI)
static void switch_to_user PROTO((const char *));

static void
switch_to_user (username)
const char *username;
{
  struct passwd *pw,
pw = getpwnam (username);

if (pw == NULL)
{
  printf ("* Fatal error, aborting.\n\nERROR 0: No such user "%s", username.\
   */ I'm doing this manually rather than via error_exit ()
   because I'm not sure whether we want to call server_cleanup.
   Needs more investigation.... *");
  exit ();
}

extern char *auth_PROTO(const char *, const char *);

ifdef SYSTEM_CLEANUP

  /* Hook for OS-specific behavior, for example socket subsystems
NT and OS2 or dealing with windows and arguments on Mac. */
  SYSTEM_CLEANUP ();
endif

exit (EXIT_FAILURE);
}

/*FIXME? We don't check for errors from initgroups, setuid, &c.
I think this mainly would come up if someone is trying to run
the server as a non-root user. I think we should be checking for
errors and aborting (as with the error above from getpwnam) if
there is an error (presumably EPERM). That means that pserver
should continue to work right if all of the "system usernames"
in CVSSROOT/passwd match the user which the server is being run
as (in inetd.conf), but fail otherwise. */

if HAVE_INITGROUPS

  initgroups (pw->pw_name, pw->pw_gid);
endif /* HAVE_INITGROUPS */

ifdef SETXID_SUPPORT

  /* honor the setgid bit off set*/
  if (setgid) != getgid())
    setgid (getgid ());
  else
    setgid (pw->pw_gid);
endif

setuid (pw->pw_uid);

/* We don't want our umask to change file modes. The modes should
be set by the modes used in the repository, and by the umask of
the client. */

u mask (0);

if HAVE_PUTENV

  /* Set LOGNAME and USER in the environment, in case they are
already set to something else. */
  {
    char *env;

    env = xmalloc (sizeof "LOGNAME" + strlen (username));
    (void) sprintf (env, "LOGNAME=%s", username);
    (void) putenv (env);

    env = xmalloc (sizeof "USER" + strlen (username));
    (void) sprintf (env, "USER=%s", username);
    (void) putenv (env);
  }
endif /* HAVE_PUTENV */

endif

ifdef AUTH_SERVER_SUPPORT

extern char *crypt_PROTO(const char *, const char *);

  /*
  * 0 means no entry found for this user.
  * 1 means entry found and password matches.
  * 2 means entry found, but password does not match.
  * 3 means entry found, but password does not match.
  * If success, host_user_ptr will be set to point at the system
  * username (i.e., the "real" identity, which may or may not be the
  * CVSS username) of this user; caller may free this. Global
  * CVSS_username will point at an allocated copy of cvs username (i.e.,
  * the username argument below).
  */

  static int
  check_repository_password (username, password, repository, host_user_ptr)
  char *username, *password, *repository, *host_user_ptr;
  
  {
    int retval = 0;
    FILE *fp;
char *filename;
char *linebuf = NULL;
size_t linebuf_len;
int foundat = 0;
int namelen;

/* We don't use /usr/local directory because it hasn't been set yet
   * - our 'repository' argument came from the authentication
   * protocol, not the regular CVS protocol.
   */
filename = xmalloc(strlen(repository) + 1 + strlen(CVSRootADM) + 1 + strlen(CVSRootADM_PASSWD) + 1);

(void) sprintf (filename, "%s/Ta/La", repository, CVSRootADM, CVSRootADM_PASSWD);

fp = CVS_FOPEN (filename, "r");
if (fp == NULL) {
    if (fexist_error (errno))
        error (0, errno, "cannot open %s", filename);
    return 0;
}

/* Look for a relevant line - one with this user's name. */
namelen = strlen (username);
while (getline (&linebuf, &linebuf_len, fp) >= 0) {
    if (strncmp (linebuf, username, namelen) != 0)
        continue;
    if (linebuf[namelen] == ':')
        foundat = 1;
    break;
}

if (ferror (fp))
    error (0, errno, "cannot read %s", filename);
if (fclose (fp) < 0)
    error (0, errno, "cannot close %s", filename);

/* If foundat != 0, then linebuf contains the information we need. */
if (foundat) {
    char *found_password, *host_user_tmp;
    strtok (linebuf, ":");
    found_password = strtok (NULL, ": \n");
    host_user_tmp = strtok (NULL, ": \n");
    if (host_user_tmp == NULL)
        host_user_tmp = username;

    if (strcmp (found_password, crypt (password, found_password)) == 0)
        { /* Give host_password its permanent storage. */
            *host_user_ptr = xsstrdup (host_user_tmp);
            retval = 1;
        }
    else
        { *host_user_ptr = NULL;
            retval = 2;
        }
    else
        { *host_user_ptr = NULL;
            retval = 0;
        }
    free (filename);
if (linebuf)
    free (linebuf);
    return retval;
}

/* Return a hosting username if password matches, else NULL. */
static char *
check_password (username, password, repository)
char *username, *password, *repository;
{ int rc;
    char *host_user = NULL;

First we see if this user has a password in the CVS-specific password file. If so, that’s enough to authenticate with. If not, we’ll check /etc/passwd.

```c
rc = check_repository_password(username, password, repository, &host_user);
if (rc == 2)
    return NULL;
```

```c
else if (rc == 1)
{
    /* host_user already set by reference, so just return */
    goto handle_return;
}
else if (rc == 0 && system_auth)
{
    /* No cvs password found, so try /etc/passwd. */
    const char *found_passwd = NULL;
    struct passwd *pw;
    #ifdef HAVE_GETSPNAM
    struct spwd *spw;
    spw = getspnam(username);
    if (spw != NULL)
    {
        found_passwd = spw->sp_pwdp;
    }
    #endif
    if (found_passwd == NULL && (pw = getpwnam(username)) != NULL)
    {
        found_passwd = pw->pw_passwd;
    }
    if (found_passwd == NULL)
    {
        printf("E Fatal error, aborting.\n"

        error 0 %s: no such user
"), username);
    } else if (password && *password)
    {
        /* user exists and has a password */
        host_user = (((stricmp(found_passwd,
crypt(password, found_passwd)))

        ? username : NULL),
        goto handle_return;
    }
    else if (password && *password)
    {
        /* user exists and has no system password, but we got
one as parameter */
        host_user = username;
        goto handle_return;
    }
else
    {
        /* user exists but has no password at all */
        host_user = NULL;
        goto handle_return;
    }
}
else if (rc == 0)
{
    /* Note that the message *does* distinguish between the case in
which we check for a system password and the case in which
we do not. It is a real pain to track down why it isn’t
letting you in if it won’t say why, and I am not convinced
that the potential information disclosure to an attacker
outweighs this. */
    printf("error 0 no such user %s in %s",

    error 0 %s: no such user %s in CVSROOT/passwd", username);
    /* I’m doing this manually rather than via error_exit() because I’m not
sure whether we want to call server_cleanup.
```
Needs more investigation.... */

ifdef SYSTEM_CLEANUP
    /* Hook for OS-specific behavior, for example socket subsystems on
     * NT and OS2 or dealing with windows and arguments on Mac. */
    SYSTEM_CLEANUP ();
endif

exit(EXIT_FAILURE);

else
    host_user = NULL;
    goto handle_return;

handle_return:
    if (host_user)
    {
        /* Set CVS_Username here, in allocated space.
         * It might or might not be the same as host_user. */
        CVS_Username = xmalloc(strlen(username) + 1);
        strcpy(CVS_Username, username);
    }

return host_user;
endif

/* AUTH_SERVER_SUPPORT */

ifdef (AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI) || defined (HAVE_KERBEROS)

/* Read username and password from client (i.e., stdin).
If correct, then switch to run as that user and send an ACK to the
client via stdout, else send NACK and die. */

void server_authenticate_connection ()
{
    char *tmp = NULL;
    size_t tmp_allocated = 0;
    char *tmpcopy = NULL;
    char *current = NULL;
    char *next = NULL;
    int verify_and_exit = 0;
    int method;
    int success = 0;
    int result;

    /* The Authentication Protocol. Client sends:
     * "BEGIN AUTH REQUEST <METHOD>\n <REPOSITORY>\n <USERNAME>\n <PASSWORD>\n END AUTH REQUEST"
     * Method is one of ps server/kserver/gserver
     * Server uses above information to authenticate, then sends
     * "I LOVE YOU\n"
     * if it grants access, else
     * "I HATE YOU\n"
     * if it denies access (and it exits if denying).
     * When the client is "cvs login", the user does not desire actual
     * repository access, but would like to confirm the password with
     * the server. In this case, the start and stop strings are
     * "BEGIN VERIFICATION REQUEST <METHOD>\n and
     * END VERIFICATION REQUEST"
     * On a verification request, the server’s responses are the same
     * (with the obvious semantics), but it exits immediately after
     * sending the response in both cases.
     * Why is the repository sent? Well, note that the actual
     * client/server protocol can’t start up until authentication is
     * successful. But in order to perform authentication, the server
     * needs to look up the password in the special CVS passwd file,
     * before trying /etc/passwd. So the client transmits the
     * repository as part of the “authentication protocol”. The
     * repository will be redundantly retransmitted later, but that’s no
     * big deal.

5180

5190

5200

5210

5215

5220

5225

5230

5235

5240

5245

5250

5255

5260

5265

5300

5305

5310

5315

5320

5325

5330

5335

5340

5345

5350

5355

5360

5365

5370

5375

5380

5385

5390

5395

5400

5405

5410

5415

5420

5425

5430

5435

5440

5445

5450

5455

5460

5465

5470

5475

5480

5485

5490

5495

5500

5505

5510

5515

5520

5525

5530

5535

5540

5545

5550

5555

5560

5565

5570

5575

5580

5585

5590

5595

5600

5605

5610

5615

5620

5625

5630

5635

5640

5645

5650

5655

5660

5665

5670

5675

5680

5685

5690

5695

5700

5705

5710

5715

5720

5725

5730

5735

5740

5745

5750

5755

5760

5765

5770

5775

5780

5785

5790

5795

5800

5805

5810

5815

5820

5825

5830

5835

5840

5845

5850

5855

5860

5865

5870

5875

5880

5885

5890

5895

5900

5905

5910

5915

5920

5925

5930

5935

5940

5945
*/

ifdef SO_KEEPALIVE
    /* Set SO_KEEPALIVE on the socket, so that we don't hang forever
       if the client dies while we are waiting for input. */
    {
        int on = 1;

        (void) setsockopt (STDIN_FILENO, SOL_SOCKET, SO_KEEPALIVE,
               (char *) &on, sizeof on);
    }
#endif

    /* Make sure the protocol starts off on the right foot... */
    if (getline (&tmp, &tmp_allocated, stdin) < 0)
        /* FIXME: what? We could try writing error eof, but chances
           are the network connection is dead bidirectionally. log it
           somewhere? */
        ;

    tmpcopy = xstrdup (tmp);

    error (1, 0, "Out of memory");

    next = tmpcopy;

    current = next;
    next = strchr (current, ' ');
    if (next == NULL) {
        error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    *next = '\0';
    next++;

    if (strcmp (current, "BEGIN") != 0) {
        error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    current = next;
    next = strchr (current, ' ');
    if (next == NULL) {
        error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    *next = '\0';
    next++;

    if (strcmp (current, "GSSAPI") == 0) {
        method = AUTH_GSSAPI;
    } else if (strcmp (current, "KERBEROS_V4") == 0) {
        method = AUTH_KERBEROS_V4;
    } else if (strcmp (current, "PASSWORD") == 0) {
        method = AUTH_PASSWORD;
    } else {
        error (1, 0, "Bad authentication method: %s", current);
    }

    current = next;
    next = strchr (current, ' ');
    if (next == NULL) {
        error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    *next = '\0';
    next++;

    if (strcmp (current, "AUTHENTICATION") == 0) {
        verify_and_exit = 0;
    } else if (strcmp (current, "VERIFICATION") == 0) {
        verify_and_exit = 0;
    } else {
        error (1, 0, "Bad authentication request: %s", current);
    }

    current = next;
    next = strchr (current, 'n');
    if (next != NULL) *
        next = '\0';

    if (strcmp (current, "REQUEST") != 0) {
        error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    free (tmpcopy);

    if (method == AUTH_GSSAPI) {
        ifdef HAVE_GSSAPI
        success = gserver_authenticate_connection (verify_and_exit);
        else
            error (1, 0, "GSSAPI authentication not supported by this server");
    }
# Source code

```c
#else if (method == AUTH_KERBEROS_V4) {
    ifdef HAVE_KERBEROS
        success = kserver_authenticate_connection(verify_and_exit);
    endif
    exit (1, 0, "Kerberos v4 authentication not supported by this server");
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
```
won't say why, and I am not convinced that the potential information disclosure to an attacker outweighs this). */
        result = 0;    /* failure */
        goto failure;
    }
/* OK, now parse the config file, so we can use it to control how to check passwords. If there was an error parsing the config file, parse_config already printed an error. We keep going. Why? Because if we didn't, then there would be no way to check in a new CVSROOT/config file to fix the broken one! */
parse_config (repository);
/* We need the real cleartext before we hash it. */
descrambled_password = descramble (password);
host_user = check_password (username, descrambled_password, repository);
memset (descrambled_password, 0, strlen (descrambled_password));
if (host_user)
    {
        printf (“I LOVE YOU”);
        fflush (stdout);
    }
else
    {
        printf (“I HATE YOU”);
        fflush (stdout);
        /* I'm doing this manually rather than via error_exit () because I'm not sure whether we want to call server_cleanup. Needs more investigation.... */
    }
#endif SYSTEM_CLEANUP
    /* Hook for OS-specific behavior, for example socket subsystems on NT and OS2 or dealing with windows and arguments on Mac. */
    SYSTEM_CLEANUP ();
#endif
exit (EXIT_FAILURE);
}
/* Don't go any farther if we're just responding to "cvs login". */
if (verify_and_exit)
    {
        SYSTEM_CLEANUP ();
    }
#ifdefined SYSTEM_CLEANUP
    /* Hook for OS-specific behavior, for example socket subsystems on NT and OS2 or dealing with windows and arguments on Mac. */
    SYSTEM_CLEANUP ();
#endif
if (host_user)
    {
        result = 0;    /* failure */
        goto failure;
    }
/* Set Pserver_repos so that we can check later that the same repository is sent in later client/server protocol. */
Pserver_repos = xmalloc (strlen (repository) + 1);
strcpy (Pserver_repos, repository);
/* Switch to run as this user. */
if (verify_and_exit)
    {
        switch_to_user (host_user);
        result = 1;    /* success */
    }
failure:
    free (tmp);    
    free (repository);
    free (username);
    free (password);
return result;
#endif /* AUTH_SERVER_SUPPORT */
#endif

HAVE_KERBEROS
static int
kserver_authenticate_connection (int verify_and_exit)
    {
        int status;
        char instance[INST_SZ];
        struct sockaddr_in peer;
        struct sockaddr_in ladder;
        int len;
        KTTEXTFT ticket;    
        AUTHDAT auth;
        char version[KRIBENDAUTHYLEN];
        char user[ANAME_SZ];
strncpy (instance, "*"),
len = sizeof peer;
if (getpeername (STDIN_FILENO, (struct sockaddr *) &peer, &len) < 0
 || getsockname (STDIN_FILENO, (struct sockaddr *) &laddr,
 &len) < 0 )
{  
    print("F Fatal error, aborting.
    
error In getpeername or getsockname failed! (", stderr (errno));
    return 0;  /* failure */
}

#define SO_KEEPALIVE
/*@ Set SO_KEEPALIVE on the socket, so that we don't hang forever
if the client dies while we are waiting for input. */
{  
    int on = 1;
    (void) setsockopt (STDIN_FILENO, SOL_SOCKET, SO_KEEPALIVE,
    (char *) &on, sizeof on);
}
#endif

status = krb_recoauth (KOPT.DO_MUTUAL, STDIN_FILENO, &ticket, "read",
instance, &peer, &laddr, &auth, "", sched, version);
if (status != KSUCCESS)
{  
    print("F Fatal error, aborting.
    
error In kerberos: kerberos: status", stderr (status));
    return 0;  /* failure */
}

/* Get the local name. */
status = krb_kutoh (&auth, user);
if (status != KSUCCESS)
{  
    print("F Fatal error, aborting.
    
error In kerberos: can't get local name: kerberos: status", stderr (status));
    return 0;  /* failure */
}

/* Switch to run as this user. */
if (!verify_and_exit)
    switch_to_user (user);
return 1;  /* success */
#endif  /* HAVE_KERBEROS */
#endif

#define MAXHOSTNAMELEN
#define MAXHOSTNAMELEN (256)
#endif

static int
gserverAuthenticateConnection (int verify_and_exit)
{
    char hostname[MAXHOSTNAMELEN];
    struct hostent *hp;
    gss_buffer_t tok_in, tok_out;
    char buffer[1024];
    OM_uint32 statmin, ret;
    gss_name_t server_name, client_name;
    gss_cred_t server_creds;
    int slyten;
    gss_QID mechid;

    gethostname (hostname, sizeof hostname);
    hp = gethostbyname (hostname);
    if (hp == NULL)
        error (1, 0, "can't get canonical hostname");
    sprintf (buffer, "c\0\0", hp->h_name);
    tok_in.value = buffer;
    tok_in.length = strlen (buffer);

    if (gss_import_name (&statmin, &tok_in, GSS_C_NT_HOSTBASED_SERVICE,
    &server_name) != GSS_C_COMPLETE)
        error (1, 0, "could not import GSSAPI service name \%s", buffer);

    /* Acquire the server credential to verify the client's
authentication. */
    if (gss_acquire_creds (&statmin, server_name, 0, GSS_C_NT_NULL_OID_SET, GSS_C_ACCEPT, &server_creds,
5580         error (1, 0, "could not acquire GSSAPI server credentials");
5590         gss_release_name (&stat_min, &server_name);
5600         /* The client will send us a two byte length followed by that many bytes. */
5610         if (fread (buf, 1, nbytes, stdin) != 2)
5620             error (1, errno, "read of length failed");
5630         nbytes = ((buf[0] & 0xff) << 8) | (buf[1] & 0xff);
5640         assert (nbytes <= sizeof buf);
5650         if (fread (buf, 1, nbytes, stdin) != nbytes)
5660             error (1, errno, "read of data failed");
5670         gcontext = GSS_C_NO_CONTEXT;
5680         tok_in.length = nbytes;
5690         tok_in.value = buf;
5700         if (gss_accept_sec_context (&stat_min,
5710             &gcontext, /* context handle */
5720             &tok_in, /* input token */
5730             &tok_out, /* output token */
5740             NULL, /* channel bindings */
5750             &client_name, /* src name */
5760             &mechid, /* mech type */
5770             NULL, /* ignore time sec */
5780             NULL)/* ignore del/rew handle */
5790             != GSS_S_COMPLETE)
5800             { error (1, 0, "could not verify credentials");
5810             }
5820             /* FIXME: Use Kerberos v5 specific code to authenticate to a user. */
5830             /* We could instead use an authentication to access mapping. */
5840             { krb5g_context kc;
5850                 krb5_principal p;
5860                 struct buffer desc;
5870                 krb5_init_context (&kc);
5880                 krb5_free_principal (kc, p);
5890                 gss_buffer_delete (desc);
5900                 if (gss_display_name (&stat_min, client_name, &desc,
5910                     &mechid) != GSS_S_COMPLETE
5920                     || krb5_parse_name (kc, ((gss_buffer_t) &desc)->value, &p) != 0
5930                     || krb5_uname_to_localname (kc, p, &src, buf) != 0
5940                     || krb5_kuserok (kc, p, buf) != TRUE)
5950                     { error (1, 0, "access denied");
5960                     krb5_free_principal (kc, p);
5970                     krb5_free_context (kc);
5980                     }
5990                     if (tok_out.length != 0)
6000                     { char cbuf[2];
6010                     cbuf[0] = (tok_out.length >> 8) & 0xff;
6020                     cbuf[1] = tok_out.length & 0xff;
6030                     if (fwrite (cbuf, 1, 2, stdout) != 2
6040                         || (fwrite (tok_out.value, 1, tok_out.length, stdout)
6050                         != tok_out.length))
6060                         error (1, errno, "fwrite failed");
6070                     }
6080             } if (verify_and_exit) switch_to_user (buf);
6090             return 1; /* success */
6100             }
6110          sendif /* HAVE_GSSAPI */
6120          sendif /* SERVER_SUPPORT */
6130          endif defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT)
6140          /* This global variable is non-zero if the user requests encryption on
6150              the command line. */
6160          int cvssencrypt;
6170          /* This global variable is non-zero if the users requests stream
6180              authentication on the command line. */
6190          int cvsauthenticate;
6200          ifdef HAVE_GSSAPI
/* An buffer interface using GSSAPI. This is built on top of a packetizing buffer. */

/* This structure is the closure field of the GSSAPI translation routines. */

struct cvgsapai_wrap_data
{
    /* The GSSAPI context. */
    gcontext *gcontext;
};

static int cvgsapai_wrap_input PROTO((void *, const char *, char *, int));
static int cvgsapai_wrap_output PROTO((void *, const char *, char *, int, int *));

/* Create a GSSAPI wrapping buffer. We use a packetizing buffer with GSSAPI wrapping routines. */

struct buffer *
cvgsapai_wrap_buffer_initializer(buf, input, gcontext, memory)

struct buffer *buf;
int input;
gs_ctls_t gcontext;
void (*memory) PROTO((struct buffer *));

struct cvgsapai_wrap_data *gd = (struct cvgsapai_wrap_data *) xmalloc(sizeof *gd);
gd->gcontext = gcontext;

return (packetizing_buffer_initializer(buf, input ? cvgsapai_wrap_input : NULL, input ? NULL : cvgsapai_wrap_output, gd, memory));

/* Unwrap data using GSSAPI. */

static int
cvgsapai_wrap_input (fnclosure, input, output, size)

void *fnclosure;
const char *input;
char *output;
int size;

struct cvgsapai_wrap_data *gd = (struct cvgsapai_wrap_data *) fnclosure;
gs_buffer_desc inbuf, outbuf;

OM_uint32 stat_min;
int conf;
inbuf.value = (void *) input;
inbuf.length = size;
if (gs unwrap (stat_min, gd->gcontext, &inbuf, &outbuf, &conf, NULL) != GSS_C_COMPLET)
    error (1, 0, "gs unwrap failed");

if (outbuf.length > size)
    abort ();

memcpy (output, outbuf.value, outbuf.length);
/* The real packet size is stored in the data, so we don't need to remember outbuf.length. */

return 0;

/* Wrap data using GSSAPI. */

static int
cvgsapai_wrap_output (fnclosure, input, output, size, translated)

void *fnclosure;
const char *input;
char *output;
int size;
int *translated;

struct cvgsapai_wrap_data *gd = (struct cvgsapai_wrap_data *) fnclosure;
gs_buffer_desc inbuf, outbuf;

OM_uint32 stat_min;
gs_release_buffer (stat_min, &outbuf);

return 0;
int conf_req, conf;

inbuf.value = (void*) input;
inbuf.length = size;

ifdef ENCRYPTION
conf_req = cvp_gssap_encryption;
else
conf_req = 0;
endif

if (gss_wrap(&kdstat_min, gd->gcontext, conf_req, GSS_C_QOP_DEFAULT, &inbuf, &conf, &outbuf) != GSS_C_COMPLETE)
    error(1, 0, "gss_wrap failed");

/* The packetizing buffer only permits us to add 100 bytes.
   FIXME: I don't know what, if anything, is guaranteed by GSSAPI.
   This may need to be increased for a different GSSAPI
   implementation, or we may need a different algorithm. */
if (outbuf.length > size + 100)
    abort();

memcpy(output, outbuf.value, outbuf.length);

translated = outbuf.length;
gss_release_buffer(&kdstat_min, &outbuf);

return 0;
#endif /* HAVE_GSSAPI */

ifdef ENCRYPTION
endif /* HAVE_KERBEROS */

/* This structure is the closure field of the Kerberos translation
   routines. */

struct krb_encrypt_data
{
    /* The Kerberos key schedule. */
    Key_schedule sched;
    /* The Kerberos DES Block. */
    C_Block block;
};

static int krb_encrypt_input PROTO((void *, const char *, char *, int));
static int krb_encrypt_output PROTO((void *, const char *, char *, int, int *));

/* Create a Kerberos encryption buffer. We use a packetizing buffer
   with Kerberos encryption translation routines. */

struct buffer *
krb_encrypt_buffer_initialize (buf, input, sched, block, memory)
{
    struct buffer *buf;
    int input;
    Key_schedule sched;
    C_Block block;

    void (*memory) PROTO((struct buffer *));

    struct krb_encrypt_data *kd;

    kd = (struct krb_encrypt_data *) xmalloc (sizeof *kd);
    memcpy (kd->sched, sched, sizeof (Key_schedule));
    memcpy (kd->block, block, sizeof (C_Block));

    return packetizer_buffer_initialize (buf, input ? krb_encrypt_input : NULL, input ? NULL : krb_encrypt_output, kd, memory);
}

/* Decrypt Kerberos data. */

static int
krb_encrypt_output (fnclosure, input, output, size)
{
    void *fnclosure;
    const char *input;
    char *output;
    int size;

    struct krb_encrypt_data *kd = (struct krb_encrypt_data *) fnclosure;
    int tcount;

static int    

void cerr (const char *fmt, ...)
{
    va_list val;
    va_start (val, fmt);
    fprintf (stderr, fmt, val);
    va_end (val);
}

void error (int errnum, const char *fmt, ...)
{
    va_list val;
    va_start (val, fmt);
    fprintf (stderr, fmt, val);
    va_end (val);
    exit (errnum);
}

/* Output LEN bytes at STR. If LEN is zero, then output up to (not including) the first '|\n' byte. */

void

csvwoutput (str, len)
{
    const char *str;
    size_t len;
    
    if (len == 0)
        len = strlen (str);

    if (error_use_protocol)
    {
        bufoutput (saved_output, str, len);
        bufcopy_lines (buf, bufnet, saved_output, '\n');
    }
    else if (server_active)
    {
        bufoutput (saved_output, str, len);
        bufcopy_lines (protocol, saved_output, '\n');
        bufsend_counted (protocol);
    }
    else
    
    size_t written;
    size_t to_write = len;
    const char *p = str;

    /* For symmetry with csvwoutput we would call fprintf (stderr) */
Here, I guess the assumption is that stderr will be unbuffered, so we don't need to. That sounds like a sound assumption from the manpage I looked at, but if there was something fishy about it, my guess is that calling fflush would not produce a significant performance problem. */

while (towrite > 0)
{
    written = fwrite (p, 1, towrite, stdout);
    if (written == 0)
        break;
    p += written;
    towrite -= written;
}

/* Output LEN bytes at STR in binary mode. If LEN is zero, then output zero bytes. */

void

if (error_use_protocol)
   🔞
else
    struct buffer *buf;
    char *text[40];

if (error_use_protocol)
    buf = buf_to_hex;
else
    buf = protocol;

if (supported_response ("Mbinary"))
    error (0, 0, ":

this client does not support writing binary files to stdout");
    return;

buf_output0 (buf, "Mbinary012");
sprintf (size_text, "%lu\012", (unsigned long) len);
buf_output0 (buf, size_text);

/* Not sure what would be involved in using buf_append_data here without stepping on the toes of our caller (which is responsible for the memory allocation of STR). */
buf_output (buf, str, len);

if (error_use_protocol)
    buf_send_counted (protocol);

else#endif
{
    size_t written;
    size_t towrite = len;
    const char *p = str;

#include "server_support"

if (error_use_protocol)
    fflush (stdout);
else
    /* For symmetry with cvs_output we would call fflush (stderr) here. I guess the assumption is that stderr will be unbuffered, so we don't need to. That sounds like a sound assumption from the manpage I looked at, but if there was something fishy about it, my guess is that calling fflush would not produce a significant performance problem. */
#endif

int oldmode;

/* It is possible that this should be the same ifdef as USE_SETMODE_BINARY but at least for the moment we keep them separate. Mostly this is just laziness and/or a question of what has been tested where. Also there might be an issue of setmode vs. setmode. */
/* The Windows doc says to call setmode only right after startup. I assume that what they are talking about can also be helped by flushing the stream before changing the mode. */

if (oldmode < 0)
    error (0, errno, "failed to setmode on stdout");
#endif

while (towrite > 0)
{
    written = fwrite (p, 1, towrite, stdout);
    if (written == 0)
        break;
}
ifdef USE_SETMODESTDOUT
flush (stdout);
if (setmode (fileno (stdout), oldmode) != OPEN_BINARY)
  error (0, errno, "failed to setmode on stdout");
endif
}
#endif

Like CVS OUTPUT but output is for stderr not stdout.

void cvsouterr (str, len)
  const char *str;
  size_tlen;
  {
    if (len == 0)
      len = strlen (str);
    ifdef SERVER_SUPPORT
      if (error_use_protocol)
        {
          buf_output (savedouterr, str, len);
          buf_copy_lines (buf_to_net, savedouterr, "E");
        }
      else if (server_active)
        {
          buf_output (savedouterr, str, len);
          buf_copy_lines (protocol, savedouterr, 'E');
          buf_sendcounted (protocol);
        }
    }else
      sendif
      {
        size_t written;
        size_t to_write = len;
        const char *p = str;
        /* Make sure that output appears in order if stdout and stderr
         * point to the same place. For the server case this is taken
         * care of by the fact that savedouterr always holds less
         * than a line. */
        fflush (stdout);
        while (to_write > 0)
          {
            written = fwrite (p, 1, to_write, stderr);
            if (written == 0)
              break;
            p += written;
            to_write -= written;
          }
      }
    endif

    /* Flush stderr. stderr is normally flushed automatically, of course,
    * but this function is used to flush information from the server back
    * to the client. */
    void
cvssflusherr ()
    {
      ifdef SERVER_SUPPORT
        if (error_use_protocol)
          {
            /* Flush what we can to the network, but don't block. */
            bufflush (buf_to_net, 0);
          }
        else if (server_active)
          {
            /* Send a special count to tell the parent to flush. */
            buf sendspecialcount (protocol, -1);
          }
        else
          sendif
            flush (stderr);
        }
      endif
    }
  /* Make it possible for the user to see what has been written to
  * stdout (it is up to the implementation to decide exactly how far it
  * should go to ensure this). */
  void
cvssflushout ()
  {
    ifdef SERVER_SUPPORT
      if (error_use_protocol)
        {
          /* Flush what we can to the network, but don't block. */
        }
// Output TEXT; tagging it according to TAG. There are lots more
details about what TAG means in cvs.client.texi but for the simple
case (e.g. non-client/server), TAG is just "newline" to output a
newline (in which case TEXT must be NULL), and any other tag to
output normal text.

Note that there is no way to output either \0 or \n as part of TEXT. */

void
cvs_output_tagged (tag, text)
  char *tag;
  char *text;
{
  if (text != NULL && strchr (text, '\n') != NULL)
      /* Uh oh. The protocol has no way to cope with this. For now
       we dump core, although that really isn't such a nice
       response given that this probably can be caused by newlines
       in filenames and other causes other than bugs in CVS. Note
       that we don't want to turn this into "MT newline" because
       this case is a newline within a tagged item, not a newline
       as extraneous sugar for the user. */
      assert (0);

  /* Start and end tags don't take any text, per cvs.client.texi. */
  if (tag[0] == '+' || tag[0] == '-')
      assert (text == NULL);

  if (tag[0] == 'NEWLINE')
      cvs_output (tag, 0);
  else if (text != NULL)
      cvs_output (text, 0);
}

#else

if (server_active && supported_response ("MT"))
{
    struct buffer *buf;
    if (error_use_protocol)
        buf = buf_to_net;
    else
        buf = protocol;

    buf_output0 (buf, "MT");
    buf_output0 (buf, tag);
    if (text != NULL)
    {
        buf_output (buf, "\", 1);
        buf_output (buf, text);
    }
    buf_output (buf, "\n", 1);
}
#else

if (error_use_protocol)
    buf_end_counted (protocol);
else

if (strcmp (tag, "newline") == 0)
    cvs_output ("\n", 1);
else if (text != NULL)
    cvs_output (text, 0);

#endif
A.56 server.h

/* Interface between the server and the rest of CVS. */

/* Miscellaneous stuff which isn't actually particularly server-specific. */

#define STDIN_FILENO
#define STDOUT_FILENO
#define STDERR_FILENO
#define STDOUT_FILENO
#define STDRH_FILENO

#ifndef SERVER_SUPPORT
#endif

extern int server_register;
extern int server_expanding;

/* Server functions exported to the rest of CVS. */

/* Run the server. */
extern int server_PROTO((int argc, char **argv));

/* See server.c for description. */
extern void server_pathname_check PROTO ((char *));

/* We have a new Entries line for a file. TAG or DATE can be NULL. */
extern void server_PROTO((char *name, char *version, char *timestamp,
                        char *options, char *tag, char *date, char *conflict, char *repository));

/* Set the modification time of the next file sent. This must be followed by a call to server_PROTO. */
extern void server_PROTO((struct file_info *finfo, Ver_TS *vers));

/* We want to make the Entries line for a file, and (unless server_PROTO is subsequently called) the file itself. */
extern void server_PROTO((char *name));

/* The file which just had server_PROTO called on it needs to have only the Entries line removed, not the file itself. */
extern void server_PROTO((void));

/* We just successfully checked in FILE (which is just the bare filename, with no directory). REPOSITORY is the directory for the repository. */
extern void server_PROTO((char *file, char *updatefile, char *repository));

extern void server_PROTO((char *file, char *updatefile, char *repository, char *newfile));

/* Send the appropriate responses for a file described by FINFO and VERS. This is called after server_PROTO or server_PROTO. In the latter case the file is to be removed (and VERS can be NULL). In the former case, VERS must be non-NULL, and UPDATED indicates whether the file is now up to date (SERVER_PROTO, yes, SERVER_PROTO, no, SERVER_PROTO, yes, but file is a diff from user version to repository version, SERVER_PROTO, yes, like SERVER_PROTO but with an RCS style diff). MODE is the mode the file should get, or (mode) -1 if this should be obtained from the file itself. CHECKSUM is the MD5 checksum of the file, or NULL if this need not be sent. If FILEBUF is not NULL, it holds the contents of the file, in which case the file itself may not exist. If FILEBUF is not NULL, server_PROTO will free it. */
enum server_PROTO((struct file_info *finfo, Ver_TS *vers, 
                  enum server_PROTO,arg4 updated, mode, 
                  unsigned char *checksum, struct buffer *filebuf));
/* Whether we should send RCS format patches. */
extern int server_use_rcs_diff PROTO(void);
/* Set the Entries.Static flag. */
extern void server_setstat PROTO((char *update_dir, char *repository));
/* Clear it. */
extern void server_clearstat PROTO((char *update_dir, char *repository));
/* Set or clear a per-directory sticky tag or date. */
extern void server_setsticky PROTO((char *update_dir, char *repository,
char *tag, char *date, int nonbranch));
/* Send Template response. */
extern void server_sendtemplate PROTO ((char *, char *));
extern void server_updateentries PROTO((char *file, char *update_dir, char *repository,
enum server_updateflags updated));
/* Pointer to a malloc'd string which is the directory which
the server should prepend to the pathnames which it sends
to the client. */

extern char *server_wdir;
enum progs [PROC_CHECKIN, PROC_UPDATE];
extern void server_prog PROTO((char *, char *, enum progs));
extern void server_cleanup PROTO((int sig));

#ifdef SERVER_FLOWCONTROL
/* Pause if it's convenient to avoid memory blowout */
extern void server_pause_check PROTO(void);
#endif /* SERVER_FLOWCONTROL */

#ifdef AUTH_SERVER_SUPPORT
extern char *CSERVER_username;
extern int system_auth;
#endif /* AUTH_SERVER_SUPPORT */

extern void server_output_nonbranch PROTO((struct file_info * info, server info,
char * tag, char * rev, char * server, char * root, char * repository);
#endif /* SERVER_SUPPORT */

/* Stuff shared with the client. */
struct request {
  /* Name of the request. */
  char *name;
}
#ifdef SERVER_SUPPORT
/* Function to carry out the request. ARGS is the text of the command
* after name and, if present, a single space, have been stripped off.
*/
void (*func) PROTO((char *args));
#endif /* SERVER_SUPPORT */

/* Stuff for use by the client. */
enum {
  /* Failure to implement this request can imply a fatal
   * error. This should be set only for commands which were in the
   * original version of the protocol; it should not be set for new
   * commands.
   */
  req_essential,
  /* Some servers might lack this request. */
  req_optitional,
  /* Set by the client to one of the following based on what this
   * server actually supports.
   */
  req_supported,
  req_not_supported,
  /* If the server supports this request, and we do too, tell the
   * server by making the request.
   */
  req_enableme
} status;

/* Table of requests ending with an entry with a NULL name. */
extern struct request request requests[];
#ifdef GZIP
extern int op_add extname PROTO((int char *, unsigned char *, size_t));
extern void gunzip PROTO((int char *, unsigned char **, size_t *),
extern void gzip PROTO((int char *, unsigned char **, size_t *,
extern void read_gzip PROTO((int char *, unsigned char **, size_t *,
extern void write_gzip PROTO((int char *, unsigned char **, size_t *,
extern void write_gunzip PROTO((int char *, unsigned char **, size_t *,
#endif /* GZIP */
size_t * int);
# include "cvs.h"

static Dtype status_dirproc PROTO ((void ∗callerdat, char ∗dir, char ∗repos, char ∗update_dir, List ∗entries));

static int status_fileprocPROTO ( (void ∗callerdat, struct fileinfo ∗finfo));

static int taglistprocPROTO ( (Node ∗p, void ∗closure));

static int local = 0;
static int long_format = 0;
static RCSNode ∗xrcsnode;

static const char ∗const status_usage[] = {
"Usage: %s %s [-vlR] [files...]
\t-v	Verbose format; includes tag information for the file
\t-l	Process this directory only (not recursive).
\t-R	Process directories recursively.
(Specify the --help global option for a list of other help options)

NULL
};

int cvsstatus (argc, argv)
int argc;
char ∗∗argv;
{
int c;
int err = 0;
40
if (argc == −1)
usage (status_usage);

optind = 0;
while ((c = getopt (argc, argv, "+vlR")) != −1)
{
switch (c)
{
case 'v':
long_format = 1;
b nreak;
break;
case 'l':
local = 1;
b nreak;
case 'R':
local = 0;
b nreak;
case '?':
default:
50
usage (status_usage);
b nreak;
}
}
argc = = optind;
argv += optind;
wrap_setup ();

 ifdef CLIENT_SUPPORT
70
if (client_active) {
start_server ();
ign_setup ();

if (long_format)
send_arg("-v");
if (local)
send_arg("-1");

80
send_filenames (argc, argv, SEND_EXPAND_WILD);

 /* For a while, we tried setting SEND_NO_CONTENTS here so this
 could be a fast operation. That prevents the
 server from updating our timestamp if the timestamp is
 changed but the file is unmodified. Worse, it is user-visible
 (shows "locally modified" instead of "up to date" if
 timestamp is changed but file is not). And there is no good
 workaround (you might not want to run "cvs update"; "cvs-n
"
update" doesn't update CVS/Entries; "cvs diff --brief" or
something perhaps could be made to work but somehow that
seems nonintuitive to me even if so). Given that timestamps
seem to have the potential to get munged for any number of
reasons, it seems better to not rely too much on them. */

sendFiles (argc, argv, local, 0, 0);
sendToServer (*status*\012, 0);
err = getResponse and Close ();

return err;
}

/* start the recursion processor */

err = startRecursion (*status*Fileproc, (FILESDONEPROC) NULL,
statusDirproc, (DIRLEAVEPROC) NULL, NULL,
argc, argv, local,
W\_LOCAL, 0, 1, (char *) NULL, 1);

return (err);

/* display the status of a file */

static int statusFileproc (callerdat, finfo)

    void * callerdat;

    struct fileInfo * finfo;

    CType status;

    char * sstat;

    VersTS * vers;

    status = ClassifyFile (finfo, (char *) NULL, (char *) NULL, (char *) NULL,
1, 0, &vers, 0);

    sstat = "Classify Error";

    switch (status) { 

    case T\_UNKNOWN:

        sstat = "Unknown";

        break;

    case T\_CHECKOUT:

        sstat = "Needs Checkout";

        break;

    #ifdef SERVER\_SUPPORT

    case T\_PATCH:

        sstat = "Needs Patch";

        break;

    #endif

    case T\_CONFLICT:

        /* I think that "unresolved" is correct; that if it has
been resolved then the status will change. But I'm not
sure about that. */

        sstat = "Unresolved Conflict";

        break;

    case T\_ADDED:

        sstat = "Locally Added";

        break;

    case T\_REMOVED:

        sstat = "Locally Removed";

        break;

    case T\_MODIFIED:

        if (vers->ts\_conflict)

            sstat = "File had conflicts on merge";

        else

            sstat = "Locally Modified";

        break;

    case T\_REMOVE\_ENTRY:

        sstat = "Entry Invalid";

        break;

    case T\_UPTODATE:

        sstat = "Up-to-date";

        break;

    case T\_NEEDS\_MERGE:

        sstat = "Needs Merge";

        break;

    case T\_TITLE:

        /* I don't think this case can occur here. Just print
"Classify Error". */

        break;

    }

cvsOutput ("\n
===================================================================
" ,0);

if (vers->ts\_user == NULL) {
cvs_output("File: no file ", 0);
cvs_output(fininfo->file, 0);
cvs_output("\n\Status: ", 0);
cvs_output(fininfo->file, 0);
cvs_output("\n\n", 0);
}
else
{
  char *buf;
  buf = xmalloc(strlen(fininfo->file) + strlen(fininfo->file) + 80);
  sprintf(buf, "File: %s\Status: %s\n\n", fininfo->file, fininfo->file, fininfo->file);
  cvs_output(buf, 0);
  free(buf);
}
if (vers->vn_user == NULL)
{
  cvs_output(" Working revision:\No entry for ", 0);
cvs_output(fininfo->file, 0);
cvs_output("\n", 0);
}
else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '0')
{ defs SERVER_SUPPORT
  
else if (server_active)
  {
    cvs_output(" Working revision:\No file\n", 0);
cvs_output(fininfo->vn_user, 0);
cvs_output("\n", 0);
  }
else
{
  cvs_output(" Working revision:\n", 0);
cvs_output(vers->vn_user, 0);
cvs_output("\n", 0);
  }
if (vers->vn_rcs == NULL)
{
  cvs_output(" Repository revision:\No revision control file\n", 0);
  
else
{
  cvs_output(" Repository revision:\n", 0);
cvs_output(vers->vn_rcs, 0);
cvs_output("\n", 0);
cvs_output(vers->arcfile->path, 0);
cvs_output("\n", 0);
  }
if (vers->entdata)
{
  Entnode *edata;
  edata = vers->entdata;
  if (edata->tag)
  {
    if (vers->vn_rcs == NULL)
    {
      cvs_output(" Sticky Tag:\n", 0);
cvs_output(edata->tag, 0);
cvs_output(" Missing from RCS file\n", 0);
    }
    else
    {
      if (isdigit(edata->tag[0]))
      {
        cvs_output(" Sticky Tag:\n", 0);
cvs_output(edata->tag, 0);
cvs_output("\n", 0);
      }  
else
  { char *branch = NULL;
    if (RCS_nodeisbranch(fininfo->rcs, edata->tag))
      branch = RCS_whatbranch(fininfo->rcs, edata->tag);
  
cvs_output(" Sticky Tag:\n", 0);
cvs_output(edata->tag, 0);
cvs_output("\n", 0);
cvs_output(branch ? "branch" : "revision", 0);
cvs_output("\n", 0);
cvs_output(branch ? branch : vers->vn_rcs, 0);
cvs_output("\n", 0);
    if (branch)
      free(branch);
    }
  }
}
else if (!really_quiet)
    cvs_output("Sticky Tag:\t\t(none)\n", 0);

if (edata->date)
{
    cvs_output("Sticky Date:\t\t",0);
    cvs_output((edata->date),0);
    cvs_output("\n", 0);
}
else if (!really_quiet)
    cvs_output("Sticky Date:\t\t(none)\n", 0);

if (edata->options && edata->options[0])
{
    cvs_output("StickyOptions:\t",0);
    cvs_output((edata->options[0]),0);
    cvs_output("\n", 0);
}
else if (!really_quiet)
    cvs_output("StickyOptions:\t(none)\n", 0);

if (long_format && vers->srcfile)
{
    List ∗symbols = RCS_symbols(vers->srcfile);
    cvs_output("Existing Tags:\n", 0);
    if (symbols)
    {
        xrcsnode = finfo->rcs;
        (void) walklist (symbols, tag_list_proc, NULL);
    }
    else
        cvs_output("\tNo Tags Exist\n", 0);
}
else
    cvs_output("\n", 0);
freevers(ts (&vers));
return(0);

} /* Print a warm fuzzy message */
/* ARGSUSED */

static Dtype status_dirproc (callerdat, dir, repos, update_dir, entries)
    void ∗callerdat;
    char ∗dir;
    char ∗repos;
    char ∗update_dir;
    List ∗entries;
{
    if (!quiet)
        error (0, 0, "Examine the %s, update_dir");
    return (R_PROCESS);
}

/* Print out a tag and its type */

static int tag_listproc (p, closure)
    Node ∗p;
    void ∗closure;
{
    char ∗branch = NULL;
    char ∗buf;
    if (RCSS_nodeisbranch (xrcsnode, p->key))
    {
        branch = RCS_isbranch (xrcsnode, p->key);
        buf = xmalloc (80 + strlen (p->key)
            + (branch ? strlen (branch) : strlen (p->data)));
        sprintf (buf, "\t%-25s (%s: %s)\n", p->key,
            branch ? "branch": "revision",
            branch ? branch : "branch")
            + strlen (p->data));
        cvs_output (buf, 0);
        free (buf);
    }
    if (branch)
        free (branch);
    return (0);
A.58 subr.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * Various useful functions for the CVS support code. */

#include "cvs.h"
#include "getline.h"

extern char *getlogin();

/* malloc some data and die if it fails */
char *xmalloc (bytes)
size_t bytes;
{
    char *cp;
    /* Parts of CVS try to xmalloc zero bytes and then free it. Some
     * systems have a malloc which returns NULL for zero byte
     * allocations but a free which can't handle NULL, so compensate. */
    if (bytes == 0)
        bytes = 1;
    cp = malloc (bytes);
    if (cp == NULL)
        error (1, 0, "out of memory; can not allocate %lu bytes",
              (unsigned long) bytes);
    return (cp);
}

/* realloc data and die if it fails [I've always wanted to have "realloc" do
 * a "malloc" if the argument is NULL, but you can't depend on it. Here, I
 * can "force" it. */
void *xrealloc (ptr, bytes)
void *ptr;
size_t bytes;
{
    char *cp;
    if (!ptr)
        cp = malloc (bytes);
    else
        cp = realloc (ptr, bytes);
    if (cp == NULL)
        error (1, 0, "can not reallocate %lu bytes", (unsigned long) bytes);
    return (cp);
}

/* Two constants which tune expandstring. Having MININC as large
 as 1024 might waste a bit of memory, but it shouldn't be too bad
 (CVS used to allocate arrays of, say, 3000, PATH_MAX (8192, often),
 or other such sizes). Probably anything which is going to allocate
 memory which is likely to get as big as MAXINC shouldn't be doing
 it in one block which must be contiguous, but since getrcskey does
 so, we might as well limit the wasted memory to MAXINC or so
 bytes. */
#define MININC 1024
#define MAXINC (2*1024+1024)

/* STRPTRP is a pointer returned from malloc (or NULL), pointing to N
 characters of space. Realocate it so that points to at least
 NEWSIZE bytes of space. Gives a fatal error if out of memory;
 if it returns it was successful. */
void expand_string (strptr, n, newsize)
char *strptr;
size_t n;
size_t newsize;
{
    if (*n < newsize)
    {
        while (*n < newsize)
        {
            if (*n < MININC)
                *n = MININC;
            else if (*n > MAXINC)
                *n = MAXINC;
            else
                *n = *n + MININC;
        }
        *n = newsize;
    }
}
Duplicate a string, calling xmalloc to allocate some dynamic space

```c
#include <stdlib.h>

char *xstrdup (const char *str)
{
    char *s;
    if (str == NULL)
        return (char *) NULL;
    s = xmalloc (strlen (str) + 1);
    strcpy (s, str);
    return (s);
}
```

Remove trailing newlines from STRING, destructively.

```c
#include <string.h>

void strip_trailing_newlines (char *str)
{
    int len;
    len = strlen (str) - 1;
    while (str[len] == '
')
        str[len--] = '\0';
}
```

Return the number of levels that path ascends above where it starts.

```c
#include <string.h>

int pathname_levels (char *path)
{
    char *p;
    int level;
    int max_level;
    max_level = 0;
    p = path;
    level = 0;
    do
    {
        q = strchr (p, '/');
        if (q != NULL)
            ++q;
        if (p[0] == '.' &
            --level;
        else if (p[0] == '.' &
            (p[1] == '\0' || p[1] == '/'))
            max_level = level;
        else
            ++level;
    } while (p != NULL);
    return max_level;
}
```

Free a vector, where (argv)[0], (argv)[1], . . . (argv)[*argc - 1] are malloc’d and so is argv itself. Such a vector is allocated by line2argv or expandwild, for example.

```c
void free_names (int argc, char **argv)
{
    int *pargc;
    char **argv;
    register int i;
    for (i = 0; i < *pargc; i++)
    {
        free (argv[i]);
    }
}
```
free (argv[0]);
}
free (argv);
*pargc = 0;  /* and set it to zero when done */

/* Convert LINE into arguments separated by SEPCHARS. Set *ARGC to the number of arguments found, and (*ARGV)[0] to the first argument, (*ARGV)[1] to the second, etc. *ARGV is malloc'd and so are each of (*ARGV)[0], (*ARGV)[1], ... Use free(names()) to return the memory allocated here back to the free pool. */

void line2argv (pargc, argv, line, sepchars)
int *pargc;
char ***argv;
char *line;
char *sepchars;
{
char *cp;
/* Could make a case for size_t or some other unsigned type, but we'll stick with int to avoid signed/unsigned warnings when comparing with *pargc. */

int argv_allocated;
/* Small for testing. */
/* argv_allocated must be at least 3 because at some places (-e.g. checkout(proc) cu alters argv[2]. */
argv_allocated = 4;
*argv = (char *)xmalloc (argv_allocated * sizeof (**argv));
*pargc = 0;
for (cp = strtok (line, sepchars); cp; cp = strtok ((char *)NULL, sepchars))
{
    if (*pargc == argv_allocated)
        {
            argv_allocated += 2;
            *argv = xrealloc (*argv, argv_allocated * sizeof (**argv));
        }
    (*argv)[*pargc] = xstrdup (cp);
    (*pargc)++;
}

} /* Returns the number of dots (\.) found in an RCS revision number */

int numdots (s)
const char *s;
{
    int dots = 0;
    for (; *s; s++)
        {
            if (*s == '.')
                dots++;
        }
    return (dots);
}

/* Compare revision numbers REV1 and REV2 by consecutive fields. Return negative, zero, or positive in the manner of strcmp. The two revision numbers must have the same number of fields, or else compare_revnms will return an inaccurate result. */

int compare_revnms (rev1, rev2)
const char *rev1;
const char *rev2;
{
    const char *s, *sp;
    const char *t, *tp;
    char *snext, *tnext;
    int result = 0;
    sp = s = rev1;
    tp = t = rev2;
    while (result == 0)
        {
            result = strtoul (sp, &snext, 10) - strtoul (tp, &tnext, 10);
            if (*snext == '\'0'" | *tnext == '\'0'"
                break;
            sp = snext + 1;
            tp = tnext + 1;
        }
    return result;
}

char *increment_revnms (rev)

const char *rev;

char *newrev, *p;
int lastfield;
size_t len = strlen (rev);

newrev = (char *) xmalloc (len + 2);
memcpy (newrev, rev, len + 1);
if (p == NULL)
{
    free (newrev);
    return NULL;
}

lastfield = atoi (++ p);
sprintf (p, "%d", lastfield + 1);
return newrev;

/* Return the username by which the caller should be identified in
   CVS, in contexts such as the author field of RCS files, various
   logs, etc. */
char *
getcaller ()
{
    ifndef SYSTEM_GETCALLER
        static char *cache;
        struct passwd *pw;
        uid_t uid;
    endif

    /* If there isa CVS usename, return it. */
    ifndef AUTH_SERVER_SUPPORT
        if (CVS_USERNAME != NULL)
            return CVS_USERNAME;
    endif
    ifndef SYSTEM_GETCALLER
        return SYSTEM_GETCALLER () ;
    endif
    else
        /* Get the caller’s login from his uid. If the real uid is “root”
           try LOGNAME USER or getlogin(). If getlogin() and getpwuid()
           both fail, return the uid as a string. */
        if (cache == NULL)
            return cache;

        uid = getuid ()
        if (uid == (uid_t) 0)
        {
            char *name;
            /* super-user; try getlogin() to distinguish */
            if (((name = getlogin ()) || (name = getenv("LOGNAME"))) ||
                (name = getenv("USER"))) && (name)
            {
                cache = xstrdup (name);
                return cache;
            }
        }

        if ((pw = (struct passwd *) getpwuid (uid)) == NULL)
        {
            char uidname[20];

            (void) sprintf (uidname, "uid%d", (unsigned long) uid);
            cache = xstrdup (uidname);
            return cache;
        }
        cache = xstrdup (pw->pw_name);
        return cache;
    endif

    endif
}

#ifndef GNUC
/* ARGUSED */
time_t
getdate (date, now)
char *date;
struct timeb *now;
{
    time_t foo = 0;
    return (foo);
}
two input revisions exist, then RCS guarantees that the gca will exist. */

char *
gca (rev1, rev2)
const char *rev1;
const char *rev2;
{
    int dots;
    char *gca;
    const char *p[2];
    int j[2];
    char *retval;
    if (rev1 == NULL || rev2 == NULL) {
        error (0, 0, "sanity failure in gca");
        abort();
    }
    /* The greatest common ancestor will have no more dots, and numbers of digits for each component no greater than the arguments. Therefore this string will be big enough. */
    gca = xmalloc (strlen (rev1) + strlen (rev2) + 100);
    /* walk the strings, reading the common parts. */
    gca[0] = "\0";
p[0] = rev1;
p[1] = rev2;
do {
    int i;
    char c[2];
    char *s[2];
    for (i = 0; i < 2; ++i) {
        /* swap out the dot */
        s[i] = strchr (p[i], '.');
        if (s[i] != NULL) {
            c[i] = *s[i];
        } else {
            /* or mark us at the end */
            p[i] = NULL;
        }
    }
    /* use the lowest. */
    (void) sprintf (gca + strlen (gca), "%d.%",
} while (j[0] == j[1] || p[0] != NULL && p[1] != NULL);
/* back up over that last dot. */
gca[strlen(gca) - 1] = '\0';
/* numbers differ, or we ran out of strings. we're done with the common parts. */
dots = numdots (gca);
if (dots == 0) {
    /* revisions differ in trunk major number. */
    char *q;
    const char *s;
    s = (j[0] < j[1] ? p[0] : p[1]);
    if (s == NULL) {
        /* we only got one number. this is strange. */
        error (0, 0, "bad revisions %s or %s", rev1, rev2);
        abort();
    } else
elif ((dots & 1) == 0)
{
  /* if we have an even number of dots, then we have a branch, remove the last number in order to make it a revision. */
  char *s;
  s = strchr(gca, '•');
  *s = '•';
}
retval = xstrdup(gca);
free(gca);
return retval;

} /* Give fatal error if REV is numeric and ARGV imply we are planning to operate on more than one file. The current directory should be the working directory. Note that callers assume that we will only be checking the first character of REV; it need not have '0' at the end of the tag name and other niceties. Right now this is only called from admin.c, but if people like the concept it probably should also be called from diff -r, update -r, get -r, and log -r. */

void check_numeric (rev, argc, argv)
const char *rev;
int argc;
char **argv;
{
  if (rev == NULL || !isdigit(*rev))
    return;
  /* Note that the check for whether we are processing more than one file is (basically) syntactic; that is, we don't behave differently depending on whether a directory happens to contain only a single file or whether it contains more than one. I strongly suspect this is the least confusing behavior. */
  if (argc != 1 || (!wrap_name_has(argv[0], WRAP_TOCVS) && isdir(argv[0])))
    {
      error(0, 0, "while processing more than one file: ");
      error(1, 0, "attempt to specify a numeric revision ");
    }
} /* Sanity checks and any required fix-up on message passed to RCS via '-m'.
RCS 5.7 requires that a non-total-whitespace, non-null message be provided with '-m'.
* RCS 5.7 requires that a non-total-whitespace, non-null message be provided with '-m'. Returns a newly allocated, non-empty buffer with whitespace stripped from end of lines and end of buffer.
* TODO: We no longer use RCS to manage repository files, so maybe this nonsense about non-empty log fields can be dropped.
*/

char *make_message_rcslegal (message)
char *message;
{
  char *src, *dp, *mp;
  if (message == NULL) message = "";
  /* Strip whitespace from end of lines and end of string. */
  dp = src = (char *)xmalloc(strlen(message) + 1);
  for (mp = message; *mp != '\0'; ++mp)
    {
      if (*mp == ' ')
        {
          /* End of line: back to last non-space. */
          while (dp > src && ispace(dp[-1]) || dp[-1] == 't')
            --dp;
          *dp++ = *mp;
        }
    /* Backtrack to last non-space at end of string, and truncate. */
    while (dp > src && ispace(dp[-1]))
      --dp;
} /* Given fatal error if REV is numeric and ARGV imply we are planning to operate on more than one file. The current directory should be the working directory. Note that callers assume that we will only be checking the first character of REV; it need not have '0' at the end of the tag name and other niceties. Right now this is only called from admin.c, but if people like the concept it probably should also be called from diff -r, update -r, get -r, and log -r. */
after all that, if there was no non-space in the string, substitute a non-empty message. */
if (dst == '\0') {
    free (dst);
    dst = xstrdup("*** empty log message ***");
}
return dst;
}

/* Does the file FINFO contain conflict markers? The whole concept
of looking at the contents of the file to figure out whether there are
unresolved conflicts is kind of bogus (people do want to manage files
which contain those patterns not as conflict markers), but for now it
is what we do. */
int file_has_markers (finfo)
const struct file_info *finfo;
{
    FILE *fp;
    char *line = NULL;
    size_t line_allocated = 0;
    int result;
    result = 0;
    fp = CVS_FOPEN (finfo->file, "r");
    if (fp == NULL)
        error (1, errno, "cannot open %s", fullname);
    while (getline (&line, &line_allocated, fp) > 0)
    {
        if (strncmp (line, RCS_MERGE_PAT, sizeof RCS_MERGE_PAT - 1) == 0)
        {
            result = 1;
            goto out;
        }
    }
    if (ferror (fp))
        error (0, errno, "cannot read %s", fullname);
out:
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", fullname);
    if (line != NULL)
        free (line);
    return result;
}

/* Read the entire contents of the file NAME into *BUF.
If NAME is NULL, read from stdin. */
void getfile (name, fullname, mode, buf, bufsize, len)
const char *name;
const char *fullname;
const char *mode;
char **buf;
size_t *bufsize;
size_t *len;
{
    struct stat s;
    size_t aread;
    char *tobuf;
    FILE *e;
    size_t filesize;

    if (name == NULL)
    {
        e = stdin;
        filesize = 100; /* force allocation of minimum buffer */
    }
    else
    {
        if (CVS_LSTAT (name, &s) < 0)
            error (1, errno, "can't stat %s", fullname);
        /* Don't attempt to read special files or symbolic links. */
        if (S_ISREG (s.st_mode))
        {
            *len = 0;
            return;
        }
    }
/* Convert from signed to unsigned. */
```c
filesize = s.st_size;

e = open_file(name, mode);
}
if (*bufsize < filesize)
{
    *bufsize = filesize;
    *buf = realloc(*buf, *bufsize);
}

while (1)
{
    size_t got;
    got = fread(tobuf, 1, *bufsize - (tobuf - *buf), e);
    if (ferror(e))
        error(1, errno, "can't read %s", fullname);
    nread += got;
    tobuf += got;
    if (feof(e))
        break;

    /* It's probably paranoid to think S.ST_SIZE might be too small to hold the entire file contents, but we handle it just in case. */
    if (tobuf == *buf + *bufsize)
    {
        int c;
        long off;
        c = getc(e);
        if (c == EOF)
            break;
        off = tobuf - *buf;
        expand_string(buf, bufsize, bufsize + 100);
        tobuf = *buf + off;
        *tobuf++ = c;
        ++nread;
    }
}
if (e != stdin && fclose(e) < 0)
    error(0, errno, "cannot close %s", fullname);

*len = nread;

/* Force BUF to be large enough to hold a null terminator. */
if (*buf != NULL)
{
    if (nread == *bufsize)
        expand_string(buf, bufsize, bufsize + 1);
    (*buf)[nread] = '0';
}
```
A.59  tag.c

/*@  

Copyright (c) 1992, Brian Berliner and Jeff Polk  
Copyright (c) 1989-1992, Brian Berliner  
You may distribute under the terms of the GNU  
General Public License as  
specified in the README file that comes with the  
CVS source distribution.  
*  
Tag  
*  
Add or delete a symbolic name to an RCS file, or a collection of RCS files.  
*  
Uses the checked out revision in the current directory.  
*/

#include "cvs.h"
#include "sarsced.h"

static int check_fileproc (void *callerdat, struct file_info *info);
static int check_filedoneproc (void *callerdat, int err,  
char *repos, char *update_dir,  
List *entries);

static int pretag_proc (char *repository, char *filter);
static void masterlist_delproc (PROTO((Node *p)));
static void tag_delproc (PROTO((Node *p)));
static int pretend_proc (PROTO((Node *p, void *closure)));
static Dtype tag_dirproc (PROTO ((void *callerdat, char *dir,  
char *repos, char *update_dir,  
List *entries)));
static int tag_fileproc (PROTO ((void *callerdat, struct file_info *info)));
static int tag_filedoneproc (PROTO ((void *callerdat, int err,  
char *repos, char *update_dir,  
List *entries)));

static char *numtag;
static char *date = NULL;
static char *syntag;
static int deletetag;  /* adding a tag by default */
static int branchmode;  /* make an automagic "branch" tag */
static int local;  /* recursive by default */
static int force_match = 1;  /* force tag to match by default */
static int force_tagmove;  /* don't force tag to move by default */
static int check_uptodate;  /* no uptodate-check by default */

struct tag_info  
{
  Ctype status;
  char *rev;
  char *tag;
  char *options;
};

struct master_lists  
{
  List *list;
};

static List *ntlist;
static List *tlist;

static const char *const tag_usage[] =  
{
  "Usage: %s [-RF] [-b] [-d] [-c] [-r tag|-D date] tag [files...]\n",  
  "\t-V\tLocal directory only, not recursive.\n",  
  "\t-f\tProcess directories recursively.\n",  
  "\t-d\tDelete the given tag.\n",  
  "\t-c\tExisting revision/tag.\n",  
  "\t-c\tExisting date.\n",  
  "\t-F\tForce a head revision if specified tag not found.\n",  
  "\t-B\tMake the tag a \"branch\" tag, allowing concurrent development.\n",  
  "\t-c\tMove tag if it already exists.\n",  
  "\t-c\tCheck that working files are unmodified.\n",  
  "\t-F\t\n"  
"(Specify the --help global option for a list of other help options)\n",  
  NULL  
};

int cvstag (argc, argv)
int argc;
char **argv;

80 {
    int c;
    int err = 0;
    if (argc == 1)
        usage (tag_usage);
    optind = 0;
    while ((c = getopt (argc, argv, "+FQldr:bf:bt:cd:")) == -1)
```c
{ switch (c)
{
    case 'Q':
    case 'q':
        
#ifdef SERVER_SUPPORT
        /* The CVS 1.5 client sends these options (in addition to
         * Global option requests), so we must ignore them. */
        if ((server_active)
          error (1, 0,
         "-q or -Q must be specified before \"%s\",
            command_name);
        break;
    case 'l':
        local = 1;
        break;
    case 'R':
        local = 0;
        break;
    case 'd':
        delete_flag = 1;
        break;
    case 'c':
        check_uptodate = 1;
        break;
    case 'r':
        numtag = optarg;
        break;
    case 'D':
        if (date)
            free (date);
        date = Make_Date (optarg);
        break;
    case 'f':
        force_tag_match = 0;
        break;
    case 'b':
        branch_mode = 1;
        break;
    case 'F':
        force_tag_move = 1;
        break;
    case '?':
        default:
        usage (tag_usage);
        break;
}
  }
  argc -= optind;
  argv += optind;

100 if (argc == 0)
    usage (tag_usage);
  symtag = argv[0];
  argc--;  argv++;

  if (date && numtag)
    error (1, 0, "-r and -D options are mutually
            exclusive");
  if (delete_flag && branch_mode)
    error (0, 0, "warning: -b ignored with -d options");
#include (tag_move (symtag));

#endif CLIENT_SUPPORT

if (client_active)
{
    /* We're the client side. Fire up the remote server. */
    start_server ();
    ign_setup ();

160 if (force_tag_match)
    send_arg ("-m");
  if (local)
    send_arg("-i");
  if (delete_flag)
    send_arg("-d");
  if (check_uptodate)
    send_arg("-c");
  if (branch_mode)
    send_arg("-b");
  if (force_tag_move)
    send_arg("-m");
  if (numtag)
    option_with_arg ("-r", numtag);
  if (date)
    client_senddate (date);
```
send(arg (symtag);
send_file_names (arg, argv, SEND_EXPAND_WILD);

/* SEND_NO_CONTENTS has a mildly bizarre interaction with
check_uptodate; if the timestamp is modified but the file
is unmodified, the check will fail, only to have "cvs diff"
show no differences (and one must do "update" or something to
reset the client's notion of the timestamp). */

send_files (arg, argv, local, 0, SEND_NO_CONTENTS);
send_server ("tag%012", 0);
return get_responses send_close ();
}

sendif
if (numtag != NULL)
tag_check_valid (numtag, argv, local, 0, **);
/* check to make sure they are authorized to tag all the
specified files in the repository */

mntlist = getlist();
err = start_recursion (check_fileproc, check_filedoneproc,
(DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL, NULL,
argv, argv, local, W_LOCAL, 0, 1,
(char *) NULL, 1);
if (err)
{
  error (1, 0, "correct the above errors first!");
}
/* start the recursion processor */
err = start_recursion (tag_fileproc, tag_filedoneproc, tag_dirproc,
(DIRLEAVEPROC) NULL, NULL, argv, argv, local,
W_LOCAL, 0, 0, (char *) NULL, 1);
dellist(&mntlist);
return (err);
}
/* check file that is to be tagged */
/* All we do here is add it to our list */

static int
check_fileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;
{
  char *xdir;
  Node->p;
  Ver_SFS vers;
  if (check_uptodate)
  {
    CType status = ClassifyFile (finfo, (char *) NULL, (char *) NULL,
      (char *) NULL, 1, 0, &vers, 0);
    if ((status != T_UPTODATE) && (status != T_CHECKOUT))
      {
      error (0, 0, "%s is locally modified", finfo->fullname);
      return (1);
      }
  }

  if (finfo->update-dir[0] == 'L')
xdir = *p;
else
xdir = finfo->update-dir;
if (!((p = findnode (mntlist, xdir)) != NULL))
  {
    tlist = ((struct master_lists *) p->data)->tlist;
  }
else
  {
    struct master_lists *ml;
    tlist = getlist ();
p = getnode ();
p->key = xstrdup (xdir);
p->type = UPDATE;
ml = (struct master_lists *)
xmalloc (sizeof (struct master_lists));
ml->tlist = tlist;
p->data = (char *) ml;
p->delproc = masterlist_delproc;
(void) addnode (mntlist, p);
  }
/* do list */
p = getnode ();
p->key = xstrdup (finfo->file);
p->type = UPDATE;
p->delproc = tag_delproc;

vers = VersionTS (finfo, NULL, NULL, NULL, 0, 0);
if (vers->srcfile == NULL)
{
  if (really_quiet)
    error (0, 0, "nothing known about %s", finfo->file);
  return (1);
}

/* Here we duplicate the calculation in tag_fileproc about which
version we are going to tag. There probably are some subtle races
(e.g. numtag is "foo" which gets moved between here and
tag_fileproc). */
if (numtag == NULL && date == NULL)
  p->data = xstrdup (vers->vn_user);
else
  p->data = RCS_getversion (vers->srcfile, numtag, date,
                 force_tag_match, NULL);

if (p->data != NULL)
{
  int addit = 1;
  char *oversion;

  oversion = RCS_getversion (vers->srcfile, symtag, (char *) NULL, 1, (int *) NULL);
  if (oversion == NULL)
  {
    if (!force_tag_move)
      addit = 0;
  }
  else if (strcmp (oversion, p->data) == 0)
  {
    addit = 0;
  }
  else if (force_tag_move)
  {
    addit = 0;
  }
  else if (oversion != NULL)
  {
    free (version);
    p->data = NULL;
  }

  if (oversion != NULL)
  {
    free (oversion);
    if (!addit)
      free(p->data);
    p->data = NULL;
  }
  freevers_ts(&vers);
  (void) addnode (tlist, p);
  return (0);
}

static int
tag_fileproc_done (callerdat, err, repos, update_dir, entries)

  void *callerdat;
  int err;
  char *repos;
  char *update_dir;
  List *entries;
{
  int n;
  Node *p;

  p = findnode (mtlist, update_dir);
  if (p != NULL)
  {
    tlist = ((struct masterlists *) p->data)->tlist;
  }
  else
  {
    tlist = (List *) NULL;
  }
  if ((tlist == NULL) || (tlist->next == tlist->list))
  {
    return (err);
  }

  if (n = Parse_Info (CVSROOTADM_TAGINFO, repos, pretag_proc, 1)) > 0
  {
    error (0, 0, "Pre-tag check failed");
    err += n;
  }
  return (err);
}

static int
```c
pretag_proc(repository, filter)
    char *repository, *filter;
    {
        if (filter[0] == 'r')
            char *s, *cp;
        s = xstrdup(filter);
        for (cp = s; *cp; cp++)
            {
                if (isspace(*cp))
                    { 
                        *cp = '\0';
                        break;
                    } 
                if (!isfile(s))
                    { 
                        error(0, errno, "cannot find pre-tag filter '%s'", s);
                        free(s);
                    }
                run_setup(filter);
                run_arg(symtag);
                run_arg(delete_tag ? "del" : force_tag ? "mov" : "add");
                run_arg(repository);
                walklist(tlist, pretag_list_proc, NULL);
                return (run_exec(RUNTTY, RUNTTY, RUNTTY, RUNNORMAL));
            }
        free(ml);
        return;
    }
}

tag_delproc(p)
    Node *p;
    {
        struct master_lists *ml;
        ml = (struct master_lists *)&p->data;
        dellist(&ml->tlist);
        free(ml);
        return;
    }

tag_delproc(p)
    Node *p;
    {
        if (p->data != NULL)
            { 
                free(p->data);
                p->data = NULL;
            }
        return;
    }

tag_list_proc(p, closure)
    Node *p, void *closure;
    {
        if (p->data != NULL)
            { run_arg(p->key);
              run_arg(p->data);
            }
        return (0);
    }
}
/* Called to tag a particular file (the currently checked out version is
 * tagged with the specified tag - or the specified tag is deleted).
 * ARGUSED */
static int
tag_filproc(callerdat, finfo)
    void *callerdat;
    struct fileinfo *finfo;
    {
        char *version, *oversion;
        char *version = NULL;
        char *rev;
        VersTS *vers;
        int retcode = 0;
        /* Lock the directory if it is not already locked. We can't rely
         * on tagdirproc because it won't handle the case where the user
         * specifies a list of files on the command line. */
```
/* We do not need to acquire a full write lock for the tag operation:
the revisions are obtained from the working directory, so we do not
require consistency across the entire repository. However, we do
need to prevent simultaneous tag operations from interfering with
each other. Therefore, we write lock each directory as we enter
it, and unlock it as we leave it. */

lock(dir for write (finfo->repository));

vers = VersionTS (finfo, NULL, NULL, 0, 0);

if ((numtag != NULL) || (date != NULL))
{
    version = RCS_getversion (vers->srcfile, numtag, date,
        force_tag_match, NULL);
    if (version == NULL)
    {
        freevers (vers);
        return (0);
    }
}

if (delete_flag)
{
    /* If -d is specified, "force_tag_match" is set, so that this call to
    RCS_getversion() will return a NULL version string if the symbolic
    tag does not exist in the RCS file.
    * This is done here because it's MUCH faster than just blindly calling
    * "rcs" to remove the tag... trust me.
    */

    version = RCS_getversion (vers->srcfile, symtag, NULL, 1,
        (int*) NULL);
    if (version == NULL || vers->srcfile == NULL)
    {
        freevers (vers);
        return (0);
    }
}

free (version);

if ((retcode = RCS_deltag (vers->srcfile, symtag)) != 0)
{
    if (!quiet)
        error (0, retcode == -1 ? errno : 0,
            "failed to remove tag %s from %s", symtag, vers->srcfile->path);
    freevers (vers);
    return (1);
}

RCS_rewrite (vers->srcfile, NULL, NULL);

/* warm fuzzies */
if (!really_quiet)
{
    cvs_output ("D ", 2);
    cvs_output (finfo->fullname, 0);
    cvs_output ("nn", 1);
}

freevers (vers);
return (0);

/* If we are adding a tag, we need to know which version we have checked
out and we'll tag that version. */

if (inversion == NULL)
{
    version = vers->vn_user;
}
else
{
    version = inversion;
}
if (version == NULL)
{
    freevers (vers);
    return (0);
}
else if (strcmp (version, "0") == 0)
{
    if (quiet)
        error (0, 0, "couldn't tag added but un-commited file ", 0, file);
    freevers (vers);
    return (0);
*/
* As an enhancement for the case where a tag is being re-applied to a
* large number of files, make one extra call to RCS_getversion to see
* if the tag is already set in the RCS file. If so, check to see if it
* needs to be moved. If not, do nothing. This will likely save a lot of
*/
rev = branch_mode ? RCS_magicrev (vers->srcfile, version) : version;
oversion = RCS_getversion (vers->srcfile, symtag, (char *) NULL, 1,
                            (int *) NULL);
if (oversion != NULL)
{
    int isbranch = RCS_nodeisbranch (finfo->rcs, symtag);

    /*
    * if versions the same and neither old or new are branches don't have
    * to do anything
    */
    if (strcmp (version, oversion) == 0 && !branch_mode && !isbranch)
    {
        free (oversion);
        freevers-> (vers);
        return (0);
    }
}

if (!force_tag_move)
{
    /* we're NOT going to move the tag */
cvsa_output (* "W", 2);
cvsa_output (finfo->fullname, 0);
cvsa_output (* "version": 0);
cvsa_output (symtag, 0);
cvsa_output (* already exists on ", 0);
cvsa_output (isbranch ? "branch": "version", 0);

cvsa_output (* "W", 0);
cvsa_output (oversion, 0);
cvsa_output (* NOT MOVING tag to ", 0);
      cvsa_output (branch_mode ? "branch": "version", 0);
cvsa_output (* "W", 0);
cvsa_output (rev, 0);
cvsa_output (*"\n", 1);
        free (version);
        freevers-> (vers);
        return (0);
    }
}

int tagged = 0;

/* Check if this is a remote tag, and if so, tag the file and queue a request
 to tag it on the other side as well */
if (symtag[0] == ':' )
{
    /* format of the remote tag is host:/root:tag */

    char* remotetag = strchr (remotetag + 1, ':');
    if (remotetag != NULL) 
    {
        remotetag = strchr (remotetag + 1, ':');
        if (remotetag != NULL) 
            char* remote_location;

tagged++;  
    }  
    /* found the tag */
    remotetag++;  
    /* if we got this end of the branch, tell the client to go to the other server
and create the other end of the branch */
char *server = remote_location;
char *repository = strchr (server, ':');
char *remote_path;
*repository = ' \0';
repository += 4;
remote_path = xmalloc (strlen (vers) + strlen (repository) + 1);
sprintf (remote_path, "\%s\", repository);
*strchr (remote_path, '/' ) = '\0';

cv_output("Create-remote-branch ", 0);
cv_output(info->fullname, 0);
cv_output("\n", 0);
cv_output(server, 0);
cv_output(vers, 0);
cv_output(repository, 0);
cv_output("\n", 0);
cv_output(remote_path, 0);
cv_output("\n", 0);
cv_output(vers, 0);
cv_output(vers, 0);
}
tagged = 1;
}
}

if (!tagged) {
retcode = RCS_settag (vers->srcfile, symtag, rev);
}
}

if (retcode != 0) {
error (1, retcode, " failed to set tag %s to revision %s in %s",
rev, vers->srcfile->path);
freevers (vers);
return (1);
}
RCS_rewrite (vers->srcfile, NULL, NULL);

/* more warm fuzzies */
if (really_quiet) {

cvs_output ("T ", 2);
cvs_output (info->fullname, 0);
cvs_output ("\n", 1);
}

if (revision != NULL) {
free (revision);
}
freevers (vers);
return (0);
}

/* Clear any lock we may hold on the current directory. */
static int
tag_filesdoneproc (callerdat, err, repos, update, entries)
void *callerdat;
int err;
char *repos;
char *update;
List *entries;
{
Lock_Cleanup ();
return (err);
}

/* Print a warm fuzzy message */
/* ARGUSED */
static DirProc
tag_dirproc (callerdat, dir, repos, update, entries)
void *callerdat;
char *dir;
char *repos;
char *update;
List *entries;
{
if (quiet) error (0, 0, "Is 3a", delete_flag ? "Un-tagging" : "Tagging", update, dir);
return (R_OK);
*/ Code relating to the val-tags file. Note that this file has no way
of knowing when a tag has been deleted. The problem is that there
is no way of knowing whether a tag still exists somewhere, when we
delete it some places. Using per-directory val-tags files (in
CVSREP) might be better, but that might slow down the process of
verifying that a tag is correct (maybe not, for the likely cases,
if carefully done), and/or be harder to implement correctly. */

struct val_args {
    char *name;
    int found;
};

static int val_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
val_fileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;

    RCSNode *rcsdata;
struct val_args *args = (struct val_args *)callerdat;
char *tag;

    if ((rcsdata = finfo->rsc) == NULL)
        /* Not sure this can happen, after all we passed only
          RCSNODE | W_ARTIFICIAL */
        return 0;

    tag = RCS_gettag (rcsdata, args->name, 1, (int *) NULL);
    if (tag != NULL) {
        /* FIXME: should find out a way to stop the search at this point. */
        args->found = 1;
        free (tag);
    }
    return 0;

static Dtype val_direntproc PROTO ((void *, char *, char *, char *, List *));

static Dtype
val_direntproc (callerdat, dir, repository, update_dir, entries)
void *callerdat;
char *dir;
char *repository;
char *update_dir;
List *entries;

    /* This is not quite right—it doesn’t get right the case of “cvs
       update -d -r foobar” where foobar is a tag which exists only in
files in a directory which does not exist yet, but which is
about to be created. */
    if ((dir) (dir))
        return 0;
    return RCS_KIP_ALL;

    /* Check to see whether NAME is a valid tag. If so, return. If not
print an error message and exit. ARGC, ARGV, LOCAL, and AFLAG specify
which files we will be operating on.

REPOSITORY is the repository if we need to cd into it, or NULL if we
are already there, or "" if we should do a W_LOCAL recursion.
Sorry for three cases, but the "" case is needed in case the
working directories come from diverse parts of the repository, the
NULL case avoids an unnecessary cd, and the non-NULL, non-""
case is needed for checkout, where we don’t want to cd if the
tag is found in CVSROOTADM_VALTAGS, but there is not (yet) any
local directory. */

    void
tag_check_valid (name, argc, argv, local, aflag, repository)
char *name;
int argc;
char **argv;
int local;
int aflag;
char *repository;

    { DBM *db;
        char *valtags_filename;
        int err;
        datum mytag;
        struct val_args theval_args;
        struct savedcwd cwd;
        int which;

        /* Numeric tags require only a syntactic check. */
if (isdigit (name[0]))
{
    char *p;
    for (p = name; *p != '\0'; ++p)
        error (1, 0, "%s contains characters other than digits and ", name);
    return;
}

/* Special tags are always valid. */
if (strcmp (name, TAG_BASE) == 0)
    return;

/* FIXME: This routine doesn't seem to do any locking whatsoever
   (and it is called from places which don't have locks in place).
   If two processes try to write val-tags at the same time, it would
   seem like we are in trouble. */
mytag.tp = name;
mytag.dsize = strlen (name);
valtags_filename = xmalloc (strlen (CVSROOT_directory) + sizeof CVSROOTADM + sizeof CVSROOTADM_VALTAGS + 20);
strcpy (valtags_filename, CVSROOT_directory);
strcat (valtags_filename, ".");
strcat (valtags_filename, CVSROOTADM);
strcat (valtags_filename, ".");
strcat (valtags_filename, CVSROOTADM_VALTAGS);
db = dbmopen (valtags_filename, O_RDONLY, 0666);
if (db == NULL)
{
    if (errno == ENOENT)
        error (1, errno, "cannot read %s", valtags_filename);
    /* If the file merely fails to exist, we just keep going and create
     * it later if need be. */
}
else
{
    datum val;
    val = dbm_fetch (db, mytag);
    if (val.dp == NULL)
    {
        /* Found. The tag is valid. */
        dbm_close (db);
        free (valtags_filename);
        return;
    }
    /* FIXME: should check errors somehow (add dbm_error to mydbs.c)? */
}

/* We didn't find the tag in val-tags, so look through all the RCS files
   to see whether it exists there. Yes, this is expensive, but there
   is no other way to cope with a tag which might have been created
   by an old version of CVS, from before val-tags was invented.

Since we need this code anyway, we also use it to create
entries in val-tags in general (that is, the val-tags entry
will get created the first time the tag is used, not when the
tag is created). */

thevalargs.name = name;
thevalargs.count = 0;
which = W_REPOS | W_ATTIC;
if (repository != NULL)
{
    if (repository[0] == '\0')
        which = W_LOCAL;
    else
        if (strcmp (repository, (const char *) kwdw))
            errorexit (1);
        else
            if ( CVSrootCHDIR (repository) < 0)
                error (1, errno, "cannot change to %s directory", repository);
    err = startrecursion (valfileproc, (FILESDONEPROC) NULL, valrdirentproc, (DIRLEAVEPROC) NULL, (void *)&thevalargs, argc, argv, local, which, aflag, 1, NULL, 1);
if (repository != NULL && repository[0] != '\0')
{
    if (restorecwd((cwd, NULL))
        exit (EXIT_FAILURE);
    free_cwd((cwd));
}

if (!the_val_args.found)
    error (1, 0, "no such tag %s", name);
else
{
    /* The tags is valid but not mentioned in val-tags. Add it. */
    datumvalue;
    if (noexec)
        {
            if (db != NULL)
                dbm_close (db);
            free (valtags_filename);
            return;
        }
    if (db == NULL)
        {
            modb_omask;
            omask = umask (cvsumask);
            db = dbm_open (valtags_filename, O_RDWR | O_CREAT | O_TRUNC, 0666);
            (void) umask (omask);
            if (db == NULL)
                {
                    error (0, errno, "cannot create %s", valtags_filename);
                    free (valtags_filename);
                    return;
                }
            value.dptr = '\0';
            value.dsize = 1;
            if (dbm_store (db, mytag, value, DBM_REPLACE) < 0)
                error (0, errno, "cannot store %s into %s", name, valtags_filename);
        }
    dbm_close (db);
    free (valtags_filename);
}

/* Check whether a join tag is valid. This is just like */
/* tag_check_valid, but we must stop before the colon if there is one. */

void
tag_check_valid_join (join_tag, argc, argv, local, aflag, repository)
    char *join_tag;
    int argc;
    char **argv;
    int local;
    int aflag;
    char *repository;
{
    char *c, *s;

c = xstrdup (join_tag);
s = stcrch (c, '\0');
    if (s != NULL)
        {
            if (isdigit (join_tag[0]))
                error (1, 0,
                    "Numeric join tag %s may not contain a date specifier", 
                    join_tag);
            *s = '\0';
        }

tag_check_valid (c, argc, argv, local, aflag, repository);
    free (c);
A.60 update.c

/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 * "update" updates the version in the present directory with respect to the RCS
 * repository. The present version must have been created by "checkout". The
 * user can keep up-to-date by calling "update" whenever he feels like it.
 * The present version can be committed by "commit", but this keeps the version
 * in fact.
 * Arguments following the options are taken to be file names to be updated,
 * rather than updating the entire directory.
 * Modified or non-existent RCS files are checked out and reported as U
 * <user_file>
 * Modified user files are reported as M <user_file>. If both the RCS file and
 * the user file have been modified, the user file is replaced by the result
 * of rcsmerge, and a backup file is written for the user in .file.version.
 * If this throws up irreconcilable differences, the file is reported as C
 * <user_file>, and as M <user_file> otherwise.
 * Files added but not yet committed are reported as A <user_file>. Files
 * removed but not yet committed are reported as R <user_file>.
 * If the current directory contains subdirectories that hold concurrent
 * versions, these are updated too. If the -d option was specified, new
 * directories added to the repository are automatically created and updated
 * as well.
 */

#include "cvs.h"
#include "savedud.h"
#define SERVER_SUPPORT
#include "mif.h"
#endif

#define "watch.h"
#define "fileattr.h"
#include "edit.h"
#endif
#endif

static int checkout_file PROTO ((struct update *user_fileinfo, int adding, int merging, int update_server));

#define SERVER_SUPPORT
static void checkout_file_buffer PROTO ((void *, const char *, size_t));
#endif
#define SERVER_SUPPORT
static int patchfile PROTO ((struct file_info *fileinfo, VersTS *versTS, int *dosedcheckout, struct stat *filestat info, unsigned char *checksum));
static void patchfile_write PROTO ((void *, const char *, size_t));
#endif

static int merge_file PROTO ((struct file_info *fileinfo, VersTS *versTS));
static int scratchfile PROTO ((struct file_info *fileinfo));
static int D-type update_datetime PROTO ((void *callerdata, char *dir, char *repository, char *update_dir, List *entries));
static int update_entry PROTO ((void *callerdata, struct file_info *fileinfo));
static int update_file proc PROTO ((void *callerdata, struct file_info *fileinfo, int err, char *update_dir, List *entries));
static int update_fileproc PROTO ((void *callerdata, struct file_info *fileinfo));
static int update_filesdone proc PROTO ((void *callerdata, int err, char *repository, char *update_dir, List *entries));
#endif

static char *options = NULL;
static char *tag = NULL;
#if 0
 This is a bit of a kludge. We call WriteTag at the beginning
 before we know whether nonbranch is set or not. And then at the
 end, once we have the right value for nonbranch, we call WriteTag
 again. I don't know whether the first call is necessary or not.
 rename_tag is nonzero if we are going to have to make that second
 call. */
static int rewrite_tag;
static int nonbranch;
If we set the tag or date for a subdirectory, we use this to undo the setting. See update dirent proc.

static char *tagupdate_dir;
static char *join_revl, *date_revl;
static char *join_revl2, *date_revl2;
static int aflag = 0;
static int force_tag_match = 1;
static int update_build_dirs = 0;
static int update_prune_dirs = 0;
static int pipeout = 0;
#endif
static int patches = 0;
static int rcs_diff_patches = 0;
#endif
static List *ignlist = (List *) NULL;
static time_t last_registers_time;
static const char *const update_usage[] = {
  "Usage: %s %s [-APdflRp] [-k kopt] [-r rev|-D date] [-j rev]
  [-I ign] [-W spec] [files...]
  -A Reset any sticky tags/date/kopts.
  -P Prune empty directories.
  -d Build directories, like checkout does.
  -f Force a head revision match if tag/date not found.
  -l Local directory only, no recursion.
  -R Process directories recursively.
  -p Send updates to standard output (avoids stickiness).
  -k kopt Use RCS kopt -k option on checkout.
  -r rev Update using specified revision/tag (is sticky).
  -D date Set date to update from (is sticky).
  -j rev Merge in changes made between current revision and rev.
  (Specify the --help global option for a list of other help options)
  NULL
};
int update(argc, argv)
int argc;
char **argv;
{ int c, err;
int local = 0; /* recursive by default */
int which; /* where to look for files and dirs */
if (argc == 1) usage (update_usage);
ign_setup ();
wrap_setup ();
/* parse the args */
optind = 0;
  switch (c)
  { case 'A':
    aflag = 1;
    break;
  case 'I':
    ign_add(optarg, 0);
    break;
  case 'W':
    wrap_add(optarg, 0);
    break;
  case 'k':
    if (options)
      free (options);
    options = RCS_check_kflag (optarg);
    break;
  case 'd':
    local = 1;
    break;
  case 'r':
    local = 0;
    break;
  case 'D':
    break;
  case 'j':
    break;
#define SERVER_SUPPORT
  /* The CVS 1.5 client sends these options (in addition to
   * Global option requests), so we must ignore them. */
  if (server_active)
error (1, 0, "-q or -Q must be specified before \\"%s\", command_name);
break;
case 'd':
    update_build_dirs = 1;
    break;
case 'f':
    force_tag_match = 0;
    break;
case 'p':
    force_tag_match = 0;
    break;
case 'D':
    tag = optarg;
    break;
case 'p':
    pipeout = 1;
    /* so no locks will be created */
    break;
case 'j':
    if (join_rev2)
        error (1, 0, "only two -j options can be specified");
    if (join_rev1)
        join_rev2 = optarg;
    else
        join_rev1 = optarg;
    break;
case 'u':
    #ifdef SERVER_SUPPORT
    if (server_active)
        { 
        patches = 1;
        rcs_diff_patches = server_rcs_diff ();
        }
    else
    #endif
    usage (update_usage);
    break;
case '?':
    default:
    usage (update_usage);
    break;
}
/* Don't mess with this if we are fetching a remote */
if (client_active && ! handling_remotes)
    first_file_arg = argc;
argc -= optind;
argv += optind;
#endif
if (client_active)
    { 
    int pass;
    /* The first pass does the regular update. If we receive at least 
    one patch which failed, we do a second pass and just fetch 
    those files whose patches failed. */
    pass = 1;
    do
    { 
    int status;
    start_server ();
    if (local)
        send_arg("-l");
    if (update_build_dirs)
        send_arg("-d");
    if (pipeout)
        send_arg("-p");
    if (force_tag_match)
        send_arg("-f");
    if (aflag)
        send_arg("-A");
    if (update_prune_dirs)
        send_arg("-P");
    client_prune_dirs = update_prune_dirs;
    option_with_arg ("-r", tag);
    if (options && options[0] != '\0')
        send_arg (options);
    if (date)
        client_senddate (date);
    if (join_rev1)
        option_with_arg ("-j", join_rev1);
        if (join_rev2)
option with arg "-j", join wv2);

wrap_send ();

/* If the server supports the command "update-patches", that means
that it knows how to handle the -a argument to update, which
means to send patches instead of complete files.*/

We don't send -a if failed_patches != NULL, so that the
server doesn't try to send patches which will just fail
again. At least currently, the client also clobbers the
file and tells the server it is lost, which also will get
a full file instead of a patch, but it seems clean to omit
it. */

if (failed_patches != NULL)
{  
    if (supported_request("update-patches"))
        send_wv ("-a");
    }

if (failed_patches == NULL)
{
    send_files (argc, argv, local, aflag,
            update_build_dirs ? SEND_BUILD_DIRS : 0);
}

else
{
    int i;

    (void) printf ("%s client: refetching unpatchable files\n",
            program_name);

    if (toplevel != NULL
        && CVSCHDIR (toplevel) < 0)
    {  
        error (1, errno, "could not chdir to \"", toplevel);
    }

    for (i = 0; i < failed_patches_count; i++)
        {  
            (void) unlink_file (failed_patches[i]);

            send_files (failed_patches_count, failed_patches, 0,
                    aflag, update_build_dirs ? SEND_BUILD_DIRS : 0);
        }

    failed_patches = NULL,
    failed_patches_count = 0;

    send_to_server ("update\012", 0);

    status = get_responses_and_close ();

    /* If there are any conflicts, the server will return
a non-zero exit status. If any patches failed, we still
want to run the update again. We use a pass count to
avoid an endless loop. */

    /* Notes: (1) assuming that status != 0 implies a
potential conflict is the best we can cleanly do given
the current protocol. I suppose that trying to
re-fetch in cases where there was a more serious error
is probably more or less harmless, but it isn't really
ideal. (2) it would be nice to have a testsuite case for the
conflict-and-patch-failed case. */

    if (status != 0
        && (failed_patches != NULL || pass > 1))
    {  
        return status;
    }

    ++pass;
    } while (failed_patches != NULL);

return 0;
}

sendif

if (tag != NULL)
tag_check_valid (tag, argv, local, aflag, "");
if (join_rev1 != NULL)
tag_check_valid (join_rev1, argv, local, aflag, "");
if (join_rev2 != NULL)
tag_check_valid (join_rev2, argv, local, aflag, "");

/*
* If we are updating the entire directory (for real) and building dirs
* as we go, we make sure there is no static entries file and write the
The code snippet is as follows:

```c
360 */
361 if (argc <= 0 && !pipeout)
362 {
363     if (update || build || dirs)
364     {
365         if (unlink_file (CVSADM::ENTSTAT) < 0 && !existence_error (errno))
366             error (errno, "full remove file \"%s\". CV$ADM::ENTSTAT");
367     }
368     /* keep the CVS/Tag file current with the specified arguments */
369     if (aflag || tag || date)
370     {
371         WriteTag ((char *) NULL, tag, date, 0,
372                     "", /*, NameRepository (NULL, NULL));
373         rewrite_tag = 1;
374         nonbranch = 0;
375     }
376     /* look for files/dirs locally and in the repository */
377     which = W_LOCAL | W_REPOS;
378     /* look in the attic too if a tag or date is specified */
379     if (tag != NULL || date != NULL || joining())
380     {
381         which |= W_ATTIC;
382     }
383     /* call the command line interface */
384     err = do_update (argc, argv, options, tag, date, force_tag_match,
385                     local, update, build, aflag, update_prune, dirs,
386                     pipeout, which, join revv1, join revv2, (char *) NULL);
387     /* free the space MakeDate allocated if necessary */
388     if (date != NULL) free (date);
389     if (options != NULL) {
390         free (options);
391         options = NULL;
392     }
393     return (err);
394 }
395 /* */
396 /* Command line interface to update (used by checkout) */
397 /*
398 int
do_update (argc, argv, xoptions, xtag, xdate, xforce, local, xbuild, xaflag,
399         xprune, xpipeout, which, xjoin revv1, xjoin revv2, preload, update dir)
400     int argc;
401     char *argv;
402     char *xoptions;
403     char *xtag;
404     char *xdate;
405     int xforce;
406     int local;
407     int xbuild;
408     int xaflag;
409     int xprune;
410     int xpipeout;
411     int which;
412     char *xjoin revv1;
413     char *xjoin revv2;
414     char *preload, update dir;
415     {
416         int err = 0;
417         char *cp;
418         /* fill in the statics */
419         options = xoptions;
420         tag = xtag;
421         date = xdate;
422         force_tag_match = xforce;
423         update = build || dirs = xbuild;
424         aflag = xaflag;
425         update_prune || dirs = xprune;
426         pipeout = xpipeout;
427         /* setup the join support */
428         join revv1 = xjoin revv1;
429         join revv2 = xjoin revv2;
430         if (join revv1 && (cp = strchr (join revv1, '\'))) != NULL)
431         {
432             *cp++ = '\0';
433             date revv1 = MakeDate (cp);
434         }
435     }
```
else
date_rev1 = (char *) NULL;
if (join_rev2 && (cp = strchr (join_rev2, ' '))) != NULL)
{
    cp++ = '\0';
    date_rev2 = MakeDate (cp);
}
else
date_rev2 = (char *) NULL;

#define PRESERVE_PERMISSIONS_SUPPORT
if (preserveperms)
{
    /* We need to do an extra recursion, bleh. It’s to make sure
     * that we know as much as possible about file linkage. */
    hardlist = getlist(); /* save top-level working dir */

    /* FIXME: the arguments to start_recursion make me dizzy. This
     * function call was copied from the update_fileproc call that
     * follows it; someone should make sure that I did it right. */
    err = start_recursion (get_linkinfo_proc, (FILESDONEPROC) NULL,
                       (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                       argc, argv, local, which, aflag, 1,
                       preload_update_dir, 1);
    if (err)
        return (err);

    /* FIXME: at this point we should walk the hardlist
     * and update the ‘links’ field of each hardlinkinfo struct
     * to list the files that are linked on dist. That would make
     * it easier & more efficient to compare the disk linkage
     * with the repository linkage (a simple strcmp). */
}
#endif

/* call the recursion processor */
err = start_recursion (update_fileproc, update_filedone_proc,
                       update_dirent_proc, update_dirleave_proc, NULL,
                       argc, argv, local, which, aflag, 1,
                       preload_update_dir, 1);

#if defined PRESERVE_PERMISSIONS_SUPPORT
/* The get_linkinfo_proc callback adds each file to the hardlist
 * (see hardlink.c).
 */
#endif

static int
get_linkinfo_proc (callerdat, finfo)
void *callerdat;
struct fileinfo *finfo;
{
    char *fullpath;
    Node *linkp;
    struct hardlink_info *hlinfo;

    /* Get the full pathname of the current file. */
    fullpath = xmalloc (strlen (working_dir) +
                       strlen (finfo->fullname) + 2);
    sprintf (fullpath, "%s/%s", working_dir, finfo->fullname);

    /* To permit recursing into subdirectories, files
     * are keyed on the full pathname and not on the basename. */
    linkp = lookup_linkdirnode (fullpath);
    if (linkp == NULL)
    {
        /* The file isn’t on disk; we are probably restoring
         * a file that was removed. */
        return 0;
    }

    /* Create a new, empty hardlink_info node. */
    hlinfo = (struct hardlink_info *)
             xmalloc (sizeof (struct hardlink_info));
hlinfo->status = (Ctype) 0; /* is this dumb? */
hlinfo->checked_out = 0;
linkp->data = (char *) hlinfo;

return 0;

#endif

/*
 * This is the callback proc for update. It is called for each file in each directory by the recursion code. The current directory is the local instantiation. file is the file name we are to operate on. update is set to the path relative to where we started (for pretty printing).
 * repository is the repository. entries and srcfiles are the pre-parsed entries and source control files.
 * This routine decides what needs to be done for each file and does the appropriate magic for checkout
 */

static int
560 update_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    int retval;
    CType status;
    Version FS vers;
    int resurrecting;

    resurrecting = 0;

    status = Classify_Files (finfo, tag, date, options, force_tag_match, aflag, &vers, pipeout);

    /* Keep track of whether TAG is a branch tag. Note that if it is a branch tag in some files and a nonbranch tag in others, treat it as a nonbranch tag. It is possible that case 580 should elicit a warning or an error. */
    if (rewriting)
        &k tag != NULL
        &k fino->rcs != NULL)
    {
        char *rev = RCS_getversion (finfo->rcs, tag, NULL, 1, NULL);
        if (rev != NULL)
            &k RCS_nodeisbranch (finfo->rcs, tag))
            nonbranch = 1;
        if (rev != NULL)
            free (rev);
    }

    if (pipeout)
    {
        /*
         * We just return success without doing anything if any of the really funky cases occur
         * If there is still a valid RCS file, do a regular checkout type operation
         */
        switch (status)
        {
        case T_UNKNOWN:
            /* unknown file was explicitly asked about */
            retval = 0;
            break;
        case T_REMOVED_ENTRY:
            /* needs to be un-registered */
            retval = 0;
            break;
        case T_ADDED:
            /* added but not committed */
            retval = 0;
            break;
        case T_CONFLICT:
            /* old punt-type errors */
            retval = 1;
            break;
        case T_UPTODATE:
            /* file was already up-to-date */
            retval = 1;
            break;
        case T_NEEDS_MERGE:
            /* needs merging */
            retval = 1;
            break;
        case T_MODIFIED:
            /* locally modified */
            retval = 1;
            break;
        case T_REMOVED:
            /* removed but not committed */
            retval = 1;
            break;
        case T_CHECKOUT:
            /* needs checkout */
            retval = checkout_file (finfo, vers, 0, 0, 0);
            break;
        default:
            /* can't ever happen :-) */
            error (0, 0,
                   "unknown file status \d for file \n", status, fino->file);
            retval = 0;
            break;
        }
    }
    else


```c
{
    switch (status)
    {
        case T_UNKNOWN: /* unknown file was explicitly asked about */
        case T_UPTODATE: /* file was already up-to-date */
            retval = 0;
            break;
        case T_CONFLICT: /* old punt-type errors */
            retval = 1;
            write_letter (finfo, 'C');
            break;
        case T_NEEDS_MERGE: /* needs merging */
            if (active)
            {
                retval = 1;
                write_letter (finfo, 'C');
            }
            else
            {
                retval = merge_file (finfo, vers);
            }
            break;
        case T_MODIFIED: /* locally modified */
            retval = 0;
            if (vers->ts_conflict)
            {
                char *filestamp;
                int retcode;
                /* If the timestamp has changed and no conflict indicators
                 * are found, it isn't a 'C' any more. */
                ifdef SERVER_SUPPORT
                    if (server_active)
                        filestamp = vers->ts_conflict[0] == 's';
                    else
                        {
                            filestamp = time_stamp (finfo->file);
                            retcode = strcmp (vers->ts_conflict, filestamp);
                            free (filestamp);
                        }
                else
                    filestamp = time_stamp (finfo->file);
                    retcode = strcmp (vers->ts_conflict, filestamp);
                    free (filestamp);
                endif
                if (retcode)
                {
                    /* The timestamps differ. But if there are conflict
                     * markers print 'C' anyway. */
                    retcode = file_has_markers (finfo);
                }
                if (retcode)
                {
                    write_letter (finfo, 'C');
                    retval = 1;
                }
                else
                {
                    /* Reregister to clear conflict flag. */
                    Register (finfo->entries, finfo->file, vers->vn_rcs, vers->ts_rcs,
                        vers->options, vers->tag, vers->date, (char *)&0, CVSroot_directory, finfo->repository);
                }
            }
            if (retval)
            {
                write_letter (finfo, 'M');
            }
            retval = 0;
            break;
        endif
    }
```
static void update ignproc PROTO ((char *, char *));

static void
update ignproc (file, dir)
char *file;
char *dir;
{
    struct file_info finfo;

    memset (&finfo, 0, sizeof (finfo));
    finfo.file = file;
    finfo.update_dir = dir;
    if (dir[0] == '\0')
        finfo.fullname = xstredup (file);
    else
        finfo.fullname = xmalloc (strlen (file) + strlen (dir) + 10);
    strcpy (finfo.fullname, dir);

    /* only try to join if things have gone well thus far */
    if (retval == 0 && join_errno)
        join_file (finfo, vers);
    /* if this directory has an ignore list, add this file to it */
    if (ignorelist)
        {
            Node *p;
            p = getnode ();
            p->type = FILES;
            p->key = xstredup (finfo->file);
            if (addnode (ignorelist, p) != 0)
                freenode (p);
        }
    freevers (&vers);
    return (retval);
}

710 – Source code

ifdef SERVER_SUPPORT
endif

ifdef SERVER
endif

? SERVER
RCSDIFF
SERVER_PATCHED,
file_info.stmode, checksum,
(struct buffer *) NULL);

break;
}
/* If we're not running as a server, just check the
file out. It's simpler and faster than producing
and applying patches. */
/* Full through. */

730endif

case T_CHECKOUT: /* needs checkout*/
    retval = checkout_file (finfo, vers, 0, 0, 1);
    break;

740case T_ADDED: /* added but not committed*/
    write_file (finfo, 'A');
    retval = 0;
    break;

750case T_REMOVED: /* removed but not committed*/
    write_file (finfo, 'R');
    retval = 0;
    break;

760case T_REMOVE_ENTRY: /* needs to be un-registered*/
    retval = scratch_file (finfo);
    SERVER_UPDATED, (mode &) \-1,
    (unsigned char *) NULL,
    (struct buffer *) NULL);

break;

770case T_REMOTE: {
    server_output_server_arrived_for_file (finfo, vers);
    retval = 0;
    break;
}

default: /* can't ever happen :-) */
    error (0, 0,
        "missing file status \%d for file \%s", status, finfo->file);
    retval = 0;
    break;
}

/* only try to join if things have gone well thus far */
if (retval == 0 && join_errno)
    join_file (finfo, vers);
/* if this directory has an ignore list, add this file to it */
if (ignorelist)
    { Node *p;
      p = getnode ();
      p->type = FILES;
      p->key = xstredup (finfo->file);
      if (addnode (ignorelist, p) != 0)
          freenode (p);
    }
    freevers (&vers);
    return (retval);
```c
810  strcat (info.fullname, "/");
811  strcat (info.fullname, file);
812  }
813  write_message (&info, ?>> ");
814  free (info.fullname);
815  }
816
817 */ ARGSUSED */
818 static int
819 update_fileone_dir (callerdat, err, repository, update_dir, entries)
820     void *callerdat;
821     int err;
822     char *repository;
823     char *update_dir;
824     List *entries;
825 {
826     if (rewrite_tag)
827     {
828         WriteTag (NULL, tag, date, nonbranch, update_dir, repository);
829         rewrite_tag = 0;
830     }
831
832     if (/* if this directory has an ignore list, process it then free it */
833        if (ignlist)
834        {
835            ignore_files (ignlist, entries, update_dir, update_gnproc);
836            dellist (&ignlist);
837        }
838     }
839
840     /* Clean up CVS admin dirs if we are export */
841     if (strcmp (command_name, "export") == 0)
842     {
843         /* I'm not sure the existence of CVS dirs is actually possible (except
844            in cases where we really should print a message), but since
845            this code used to ignore all errors, I'll play it safe. */
846         if (unlinkfile (CVSADM) < 0 && !existence_error (errno))
847             error (0, errno, "cannot remove %s directory", CVSADM);
848     }
849     else if (SERVER_SUPPORT)
850     
851     else if (pipeout)
852     { /* SERVER_SUPPORT */
853         /* If there is no CVS/Root file, add one */
854         if ((file (CVSADM_ROOT))
855             CreateRoot ((char *) NULL, CVSSroot_original);
856     }
857     return (err);
858  }
859
860 */
861     + update_dirent_proc () is called back by the recursion processor before a
862     + sub-directory is processed for update. In this case, update_dirent proc
863     + will probably create the directory unless -d isn't specified and this is a
864     + new directory. A return code of 0 indicates the directory should be
865     + processed by the recursion code. A return of non-zero indicates the
866     + recursion code should skip this directory.
867 */
868 static Dtype
869 update_direntproc (callerdat, dir, repository, update_dir, entries)
870     void *callerdat;
871     char *dir;
872     char *repository;
873     char *update_dir;
874     List *entries;
875 {
876     if (!ignore_directory (update_dir))
877     {
878         /* print the warm fuzzy message */
879         if (quiet)
880             error (0, 0, "Ignoring %s", update_dir);
881         return R_SKIP_ALL;
882     }
883     if (!isdir (dir))
884     {
885         /* if we aren't building dirs, blow it off */
886         if (update_direntfiles)
887             return (R_SKIP_ALL);
888     }
889     if (noexec)
890     {
891         /* ignore the missing dir if -n is specified */
892         error (0, 0, "new directory '%s' -- ignored", update_dir);
893         return (R_SKIP_ALL);
894     }
895     else
896         ...
We use TAG_UPDATE_DIR to undo the tag setting in update_directory. If we did not do this, we would not correctly handle a working directory with multiple tags (and maybe we should prohibit such working directories, but they work now and we shouldn't make them stop working without more thought). */
if (tag == NULL && date == NULL && !aflag)
{
    ParseTag (&tag, &date, &nonbranch);
    if (tag != NULL || date != NULL)
        tag_update_dir = x_strdup (update_dir);
}

make_directory (dir);
Create_Admin (dir, update_dir, repository, tag, date,
    /* This is a guess. We will rewrite it later */
    via WriteTag. */
    0, 0);
    rewrite_tag = 1;
    nonbranch = 0;
Subdir_Register (entries, (char *) NULL, dir);
}
/* Do we need to check noexec here? */
else if (!pipeout)
{
    char *cvsadmdir;
    /* The directory exists. Check to see if it has a CVS */
    /* subdirectory. */

cvsadmdir = xmalloc (strlen (dir) + 80);
strcpy (cvsadmdir, dir);
strcat (cvsadmdir, "CVSADM");
strcat (cvsadmdir, CVSADM);
if (isdir (cvsadmdir))
{
    /* We cannot successfully recurse into a directory without a CVS */
    /* subdirectory. Generally we will have already printed */
    /* "? foo". */
    free (cvsadmdir);
    return R_SKIP_ALL;
}
free (cvsadmdir);
}
/* If we are building dirs and not going to stdout, we make sure there is */
/* no static entries file and write the tag file as appropriate */
if (pipeout)
{
    if (update_hnilo_dir)
    {
        char *tmp;
        tmp = xmalloc (strlen (dir) + sizeof (CVSADM_ENTSTAT) + 10);
        Владимір (tmp, "%a.%d", dir, CVSADM_ENTSTAT);
        if (unlink (tmp) < 0 && !errno || exists (error, errno))
            error (1, errno, "cannot remove file %s", tmp);
        if (server_active)
            server_who_entstat (update_dir, repository);
        free (tmp);
    }
    /* keep the CVS/Tag file current with the specified arguments */
    if (aflag || tag || date)
    {
        WriteTag (dir, tag, date, 0, update_dir, repository);
        rewrite_tag = 1;
        nonbranch = 0;
    }
    /* Initialize the ignore list for this directory */
    ignlist = getlist();
}
if (quiet) error (0, 0, "Upating %s", update_dir);

return (RUNPROCESS);
}
/*
 * update_dirleaveproc () is called back by the recursion code upon leaving
 * a directory. It will prune empty directories if needed and will execute
 * any appropriate update programs.
 */

/*
 * ARGSUSED */
static int
update_dirleaveproc (callerdat, dir, err, update_dir, entries)
    void *callerdat;
    char *dir;
    int err;
    char *update_dir;
    List *entries;
{
    FILE *fp;

    /* If we set the tag or date for a new subdirectory in
     * update_dirproc, and we're now done with that subdirectory,
     * undo the tag/date setting. Note that we know that the tag and
date were both originally NULL in this case. */
    if (tag_update_dir != NULL && strcmp(update_dir, tag_update_dir) == 0) {
        if (tag != NULL) {
            free (tag);
            tag = NULL;
        }
        if (date != NULL) {
            free (date);
            date = NULL;
        }
        nonbranch = 0;
        free (true_update_dir);
        tag_update_dir = NULL;
    }

    /* run the update_prog if there is one */
    /* FIXME: should be checking for errors from CVS_FOPEN and printing
     * them if not existence_error. */
    if (err == 0 && pipeout && noexec &&
        (fp = CVS_FOPEN (CVSADM_UPROG, "r")) != NULL) {
        char *cp;
        char *repository;
        char *line = NULL;
        size_t line_allocated = 0;
        repository = NameRepository ((char *) NULL, update_dir);
        if (getline (&line, &line_allocated, fp) >= 0) {
            if ((cp = strchr (line, '"')) != NULL)
                *cp = '\0';
            run_setup (line);
            run_arg (repository);
            cvs_output (program_name, 0);
            cvs_output (*, 1);
            cvs_output (command_name, 0);
            cvs_output (*: Executing "", 0);
            run_print (stdout);
            cvsn_output (*", 0);
            (void) run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL);
        } else if (error (fp))
            error (0, errno, "cannot read %s", CVSDADM_UPROG);
        else
            error (0, errno, "unexpected end of file on %s", CVSDADM_UPROG);
    if (fclose (fp) < 0) error (0, errno, "cannot close %s", CVSDADM_UPROG);
    if (line != NULL)
        free (line);
    free (repository);
}
}
if (strchr (dir, '/') == NULL) {
    /* FIXME: chdir (".") loses with symlinks. */
    /* Prune empty dirs on the way out - if necessary */
    (void) CVSDADM_CHDIR (", ");
    if (update_dir=runs && isemptydir (dir, 0)) {
        /* I'm not sure the existence error is actually possible (except
in cases where we really should print a message), but since this code used to ignore all errors, I'll play it safe. */
if (unlink_file_dir ((dir) < 0 && existence_error (errno))
    error (0, errno, "cannot remove %s directory", dir);
    Subdir_Deregister (entries, (char *) NULL, dir);
}
return (err);
}

static int isremoved PROTO ((Node *, void *));

/* Returns 1 if the file indicated by node has been removed. */
static int
isremoved (node, closure)
Node *node;
void *closure;
{
  Entnode *entdata = (Entnode *) node->data;

  /* If the first character of the version is a ' - ', the file has been
   * removed. */
  return (entdata->version && entdata->version[0] == ' - ') ? 1 : 0;
}

/* Returns 1 if the argument directory is completely empty, other than the
existence of the CVS directory entry. Zero otherwise. If might_NOT_EXIST
and the directory doesn't exist, then just return 0. */

int
isemptydir (dir, might_not_exist)
char *dir;
int might_not_exist;
{
  DIR *dirp;
  struct dirent *dp;

  if ((dirp = CVS_OPENDIR (dir)) == NULL)
    if (might_not_exist && existence_error (errno))
      return 0;
  error (0, errno, "cannot open directory %s for empty check", dir);
  return (0);
}

errno = 0;
while ((dp = readdir (dirp)) != NULL)
{
  if (strcmp (dp->d_name, ".") != 0)
    if (strcmp (dp->d_name, CVSADM) != 0)
      {
        /* An entry other than the CVS directory. The directory
         * is certainly not empty. */
        (void) closedir (dirp);
        return (0);
      }
    else
      {
/* The CVS directory entry. We don't have to worry about
this unless the Entries file indicates that files have
been removed, but not committed, in this directory.
(Removing the directory would prevent people from
committing the fact that they removed the files!) */
List *l;
int files_removed;
struct saved_wd cwd;

  if (save_wd (&cwd))
    error_exit (1);

  if (CVS_CHDIR (dir) < 0)
    error (1, errno, "cannot change directory to %s", dir);
  l = Entries_Open (0, NULL);
  files_removed = walklist (l, isremoved, 0);
  Entries_Close (l);

  if (restore_wd (&cwd, NULL))
    error_exit (1);
  free_wd (&cwd);

  if (files_removed != 0)
    {
      /* There are files that have been removed, but not
       * committed! Do not consider the directory empty. */
    (void) closedir (dirp);
    return (0);
    }
}
errno = 0;
}
if (errno && 0)
{
    error (0, errno, "cannot read directory ", dir);
    (void) closedir (dirp);
    return (0);
}
(void) closedir (dirp);
return (1);

/*
 * scratch the Entries file entry associated with a file
 */
static int
scratch_file (finfo)
struct fileinfo *finfo;
{
    history_write ("W", info->update_dir, "", info->file, info->repository);
    Scratch_Entry (info->entries, info->file);
    if (unlinkinfo (info->file) < 0 && !existenceerror (errno))
        error (0, errno, "unable to remove ", info->fullname);
    return (0);
}

/*
 * Check out a file.
 */
static int
checkout_file (finfo, vers, adding, merging, update_server)
struct fileinfo *finfo;
struct_TSVers_TS vers;
int adding;
int merging;
int update_server;
{
    char *backup;
    int settime, retval = 0;
    int status;
    int filewritable;
    struct buffer *revbuf;

    backup = NULL;
    revbuf = NULL;

    /* Don't screw with backup files if we're going to stdout, or if
     * we are the server. */
    if (pipeout)
        ifdef SERVER_SUPPORT
        else server_active
        endif
    else
    {
        /* If -f>-t wrappers are being used to wrap up a directory,
         * then backup might be a directory instead of just a file. */
        if (unlinkinfo (backup) < 0)
        {
            /* Not sure if the existence error check is needed here. */
            if (existenceerror (errno))
                /* FIXME: should include update_server in message. */
                error (0, errno, "error removing ", backup);
        }
        free (backup);
        backup = NULL;
    }

    filewritable = RCS_isdead (vers->srcfile, vers->version);
    if (filewritable)
    {
        /* if we are checking out to stdout, print a nice message to
         * stderr, and add the -p flag to the command */
        if (pipeout)
            { }
Checking out * 0;
cva_outerr (finfo->fullname, 0);
cva_outerr (*\n)
RCS: * 0;
cva_outerr (vers->srcfile->path, 0);
cva_outerr (*\n)
VERS: * 0;
cva_outerr (vers->vnrcs, 0);
cva_outerr (*\n
}

#define SERVER_SUPPORT
#endif

#define CLIENT_SUPPORT
#endif

if (client->active && handling_remotes && fetch_remote (finfo)) {
    /* If we are the client, and this checkout is a part of
     * fetching a remote revision for the purpose of feeding
     * it back to a server, then we check it out to a different
     * location (inside CVS/). See client.c.
     */
    int len;
    char *realfile = NULL;
    char *lastslash;
    len = strlen (finfo->file) + strlen (CVSADM) + strlen (vers->vn_tag) +
        strlen (CVSADM_CACHED_REMOTE_SEPARATOR) + 1;
    realfile = xmalloc (len);
    if (realfile == NULL) {
        error (1, errno, "checkout failed");
    }
    lastslash = strrchr (finfo->file, '/');
    if (lastslash == NULL) {
        lastslash = fninfo->file;
    } else {
        lastslash += 1;
    }
    sprintf (realfile, "%s/" + lastslash, filepath, realfile, lastslash -= info->file, CVSADM, lastslash, CVSADM_CACHED_REMOTE_SEPARATOR, vers->vn_tag);
    printf ("Checking out to %s instead of %s/\n", realfile, filepath, lastslash -= info->file);
    status = RCS_checkout (vers->arcfile, realfile, vers->vnrcs, vers->vn_tag, vers->options, RUN_TTY, NULL, NULL);
    if (status == 0) {
        FILE *remotes_file = fopen (CVSADM_REMOTES, "a+");
        if (remotes_file == NULL) {
            fprintf (remotes_file, "%s/\n", filepath, lastslash -= info->file, vers->vnrcs, strrchr (realfile, '/') + 1);
            fclose (remotes_file);
        }
    }
}
else
    status = RCS_checkout (vers->arcfile, filepath, info->file, vers->vnrcs, vers->vn_tag, vers->options, RUN_TTY, NULL, (void *) NULL);
}
if (file_is_dead || status == 0)
{
    mode = mode;
    mode = (mode > 1);
}
if (pipeout)
{
    VersTS*xvers;
    if (revbuf != NULL)
    {
        struct stat sb;
/* FIXME: We should have RCS:checkout return the mode. */
if (stat (vers_ts->arcfile->path, &sb) < 0)
    error (1, errno, "cannot stat %s",
          vers_ts->arcfile->path);
mode = statmode & ((S_IWWRITE | S_IWGRP | S_IWOTH);
}

if (cverywrite
    & & file_is_dead
    & & fileattribute (info->file, "_watched"))
{
    if (revbuf == NULL)
        chmod (info->file, 1);
    else
        /* We know that we are the server here, so
           although chmod checks umask, we don't bother. */
        mode = (((mode & S_IRUSR) ? S_IWUSR : 0)
                 || ((mode & S_IRGRP) ? S_IWGRP : 0)
                 || ((mode & S_IROTH) ? S_IWOTH : 0));
}

/* A newly checked out file is never under the spell
   of "cv edit". If we think we were editing it
   from a previous life, clean up. Would be better to
   check for the same the working directory instead of
   same user, but that is hairy. */
struct addremove_args args;
editor_set (info->file, getcaller (), NULL);
memset (&args, 0, sizeof (args);
args.remove_temp = 1;
watch_modify_watchers (info->file, &args);

/* set the time from the RCS file iff it was unknown before */
settime =
    (noexec
     & & (vers_ts->vnuser == NULL ||
          strncmp (vers_ts->vcs, "Initial", 7) == 0)
     & & file_is_dead);
wrap_fromcvs_process_file (info->file);

xvers_ts = Version_Ts (info, options, tag, date,
force_tag_match, set_time);
if (strncmp (xvers_ts->options, "-V4", 4) == 0)
    xvers_ts->options[0] = '\0';

if (revbuf != NULL)
{
    /* If we stored the file data into a buffer, then we
       didn't create a file at all, so xvers_ts->tv.user
       is wrong. The correct value is to have it be the
       same as xvers_ts->tv.cs, meaning that the working
       file is unchanged from the RCS file.

       FIXME: We should tell Version_Ts not to waste time
       statting the nonexistent file.

       FIXME: Actually, I don't think the tv.user value
       matters at all here. The only use I know of is
       that it is printed in a trace message by
       Server_register. */
    if (xvers_ts->tv.user != NULL)
        free (xvers_ts->tv.user);
    xvers_ts->tv.user = xstrdup (xvers_ts->tv.cs);
}

(void) time (&last_register_time);

if (file_is_dead)
{
    if (xvers_ts->vnuser != NULL)
    {
        error (5, 0,
               "warning: Ts is not (any longer) pertinent",
               info->fullname);
    }
    Scratch_Entry (info->entries, info->file);
#endif SERVERSUPPORT
    if (server_active & & xvers_ts->tv.user == NULL)
        server_scratch_entry_only ();
#endif /* Server支持 */

    */ FIXME: Rather than always unlink'ing, and ignoring the
existence_error, we should do the unlink only if
verified_t->user is non-NULL. Then there would be no
need to ignore an existence_error (for example, if the
user removes the file while we are running). */
if (unlinkfile (find->file) < 0 & & ! existence_error (errno))
{
    error (0, errno, "cannot remove %s", find->fullname);
}
else {
    /* Since this is a fresh checkout, we give it the appropriate repository string */
    /* mecroh firmware: add the root, not just the repository location */
    Register (find->entries, find->file, adding ? "": xversTs->vn_user, xversTs->options, xversTs->tag, xversTs->date,
        (char *)X, CVSroot_original, CVSroot_directory); /* Clear conflict flag on fresh checkout */
}
/* fix up the vers structure, in case it is used by join */
if (join_rev1)
{
    if (versTs->vn_user != NULL)
        free (versTs->vn_user);
    if (versTs->vn_rcs != NULL)
        free (versTs->vn_rcs);
    versTs->vn_user = xstrdup (xversTs->vn_user);
    versTs->vn_rcs = xstrdup (xversTs->vn_rcs);
}
/* If this is really Update and not Checkout, recode history */
if (strcmp (command_name, "update") == 0)
    history_write (vP, find->update_dir, xversTs->vn_rcs, find->file,
        find->repository);
freeversTs (&xversTs);
if (really_quiet & & file_is_upload)
    { write_letter (find, "!"); }
#endif

#ifndef SERVER_SUPPORT
if (update_server & & server_active)
    server_updated (find, versTs, merging ? SERVER_MERGED : SERVER_UPDATED,
        mode, (unsigned char *) NULL, revbuf);
#endif
else
{
    if (backup != NULL)
        { rename_file (backup, find->file);
            free (backup);
            backup = NULL;
        }
    error (0, 0, "could not check out %s", find->fullname);
}
retval = status;
}
if (backup != NULL)
{
    /* If ./f-t wrappers are being used to wrap up a directory,
    then backup might be a directory instead of just a file. */
    if (unlink_file (backup) < 0)
    {
        /* Not sure if the existence_error check is needed here. */
        if (existence_error (errno))
            /* FIXME: should include update_failure in message. */
            error (0, errno, "error removing %s", backup);
        }
    free (backup);
}
return (retval);
#endif

#ifndef SERVER_SUPPORT
/* This function is used to write data from a file being checked out
into a buffer. */

static void
checkout_to_buffer (callerdat, data, len)
    callerdat;
    const char *data;
    }
```c
size len;

struct buffer *buf = (struct buffer *) callerdat;
    buf->output (buf, data, len);
}
#endif /
∗ SERVER_SUPPORT ∗/

#define SERVER_SUPPORT

/* This structure is used to pass information between patchfile and patchfilewrite. ∗/

struct patchfiledata {
    /* File name, for error messages. ∗/
    const char *filename;
    /* File to which to write. ∗/
    FILE *fp;
    /* Whether to compute the MD5 checksum. ∗/
    int computechecksum;
    /* Data structure for computing the MD5 checksum. ∗/
    struct MD5Context context;
    /* Set if the file has a final newline. ∗/
    int finalnl;
};

/* Patch a file. Runs diff. This is only done when running as the ∗
    server. The hope is that the diff will be smaller than the file ∗
    itself. ∗/

static int patchfile (finfo, versets, docheckout, fileinfo, checksum)
struct fileinfo *finfo;
VersTS *versets;
int *docheckout;
struct stat *fileinfo;
unsigned char *checksum;
{
    char *backup;
    char *file1;
    char *file2;
    int retval = 0;
    int retcode = 0;
    int fail;
    FILE *e;
    struct patchfiledata data;
    *docheckout = 0;

    if (noexec || pipeout || joining())
    {
        *docheckout = 1;
        return 0;
    }

    /* If this file has been marked as being binary, then never send a ∗
        patch. ∗/
    if (strcmp (versets->options, "-vb") == 0)
    {
        *docheckout = 1;
        return 0;
    }

    /* First check that the first revision exists. If it has been nuked ∗
        by cvs admin -o, then just fall back to checking out entire ∗
        revisions. In some sense maybe we don't have to do this, after ∗
        all cvs.texinfo says “Make sure that no-one has checked out a ∗
        copy of the revision you outdated” but then again, that advice ∗
        doesn't really make complete sense, because “cvs admin” operates ∗
        on a working directory and so someone will almost always have ∗
        someone revision checked out. ∗/
    {
        char *rev;
        rev = RCS_gettag (finfo->rcs, versets->vn_user, 1, NULL);
        if (rev == NULL)
        {
            *docheckout = 1;
            return 0;
        }
        else
            free (rev);
    }

    /* If the revision is dead, let checkoutfile handle it rather ∗
        than duplicating the processing here. ∗/
    if (RCS_isdead (versets->archive, versets->vn_rcs))
    {
```
```c
+docheckout = 1;
    return 0;
}
backup = xmalloc (strlen (finfo->file)
    + sizeof (CVSADM)
    + sizeof (CVSPREFIX)
    + 10);

(void) sprintf (backup, "%a/%a", CVSADM, CVSPREFIX, finfo->file);
if (fwrite (finfo->file, filename, finfo->file, backup));
else
    (void) unlink (backup);

file1 = xmalloc (strlen (finfo->file)
    + sizeof (CVSADM)
    + sizeof (CVSPREFIX)
    + 10);

(void) sprintf (file1, "%a/%a-1", CVSADM, CVSPREFIX, finfo->file);
file2 = xmalloc (strlen (finfo->file)
    + sizeof (CVSADM)
    + sizeof (CVSPREFIX)
    + 10);

(void) sprintf (file2, "%a/%a-2", CVSADM, CVSPREFIX, finfo->file);

fail = 0;

/* We need to check out both revisions first, to see if either one
has a trailing newline. Because of this, we don't use readdir,
but just use diff */
if (e = CVS_FOPEN (file1, "a"));
    if (e == NULL)
        error (1, errno, "cannot open %a", file1);

data.filename = file1;
data.fp = e;
data.filename = NULL;
data.compute_checksum = 0;

    retcode = RCS_checkout (vers->srcfile, (char *) NULL,
        vers->srcuser, (char *) NULL,
        vers->srcoptions, RUNTTY,
        patchfilewrite, (void *) &data);

    if (fclose (e) < 0)
        error (1, errno, "cannot close %a", file1);

    if (retcode != 0 || ! data.filename)
        fail = 1;

if (! fail)
{
    e = CVS_FOPEN (file2, "a");
    if (e == NULL)
        error (1, errno, "cannot open %a", file2);

data.filename = file2;
data.fp = e;
data.filename = NULL;
data.compute_checksum = 1;

    MD5Init (&data.context);

    retcode = RCS_checkout (vers->srcfile, (char *) NULL,
        vers->srcuser, (char *) NULL,
        vers->srcoptions, RUNTTY,
        patchfilewrite, (void *) &data);

    if (fclose (e) < 0)
        error (1, errno, "cannot close %a", file2);

    if (retcode != 0 || ! data.filename)
        fail = 1;
    else
        MD5Final (checksum, &data.context);
}
retcode = 0;
if (! fail)
{
    char *diffoptions;

    /* If the client does not support the Rcs-diff command, we
    send a context diff, and the client must invoke patch.
    That approach was problematical for various reasons. The
    new approach only requires running diff in the server; the
    client can handle everything without invoking an external
    program. */
    if (! RCS_diffpatches)
        {
```
/* We use -c, not -u, because that is what CVS has traditionally used. Kind of a moot point, now that RCS-diff is preferred, so there is no point in making the compatibility issues worse. */
diffoptions = "-a";
} else {
  /* Now that diff is librarified, we could be passing -a if we wanted to. However, it is unclear to me whether we would want to. Does diff -a, in any significant percentage of cases, produce patches which are smaller than the files it is patching? I guess maybe text files with character sets which diff regards as 'binary'. Conversely, do they tend to be much larger in the bad cases? This needs some more thought/investigation, I suspect. */
diffoptions = "-n";
}
retcode = diff_exec (file1, file2, diff_options, finfo->file);

/* A retcode of 0 means no differences. 1 means some differences. */
if (retcode != 0) {
  & retcode != 1)
    fail = 1;
} else
#define BINARY "Binary"

unsigned int c;

/* Stat the original RCS file, and then adjust it the way that RCS(checkout would. FIXME: This is an abstraction violation. */
if ((CVS_STAT (versa->srcfile->path, fileinfo) < 0)
    error (1, errno, "could not stat %s", versa->srcfile->path);
if (chmod (finfo->file, fileinfo->filemode & ~(S_IREAD | S_IWGRP | S_IWRITE)) < 0)
  error (0, errno, "cannot change mode of file %s", finfo->file);
if (c IRequest & fileattr_get (fininfo->file, ".watched"))
  chmod (fininfo->file, 1);
/* Check the diff output to make sure patch will be handle it. */
e = CVS=fopen (fininfo->file, "r*");
if (e == NULL) {
  error (1, errno, "cannot open diff output file %s", fininfo->fullname);
  fininfo->fullname);
c = fread (buf, 1, sizeof BINARY - 1, e);
buf[0] = '\0';
if (strcmp (buf, BINARY) == 0) {
  /* These are binary files. We could use diff -a, but patch can't handle that. */
    fail = 1;
  }
fclose (e);
}

if (! fail) {
  versa->srcfile;
  /* This stuff is just copied blindly from checkout.c. I don't really know what it does. */
  versa->ts = Version_TS (fininfo, options, tag, date,
    force, &ag_match, 0);  
  if (strcmp (versa->ts->options, "-f") == 0)
    versa->ts->options[0] = '\0';
  Register (fininfo->entries, fininfo->file, versa->ts->vn_res,
    versa->ts->ts_user, versa->options, versa->ts->tag, versa->ts->date, NULL, CVSroot, directory, fininfo->repository);
  if (CVS_STAT (fininfo->file, fileinfo) < 0)
    error (1, errno, "could not stat %s", fininfo->file);
/* If this is really Update and not Checkout, recode history */
if (strcmp (command_name, "update") == 0)
  historywrite (*P", fininfo->update_dir, versa->ts->vn_res, fininfo->file, 
    fininfo->repository);
freeversa (&versa->ts);
if (really) {
static void write_letter (finfo, 'P');
}
else
{
  int old_errno = errno; /* save errno value over the rename */
  if (isfile (backup))
    rename_file (backup, finfo->file);
  if (retcode != 0 && retcode != 1)
    error (retcode == -1 ? 1 : 0, retcode == -1 ? old_errno : 0,
          "could not diff %s", finfo->fullname);
  docheckout = 1;
  retval = retcode;
}
(void) unlink_file (backup);
(void) unlink_file (file1);
(void) unlink_file (file2);
free (backup);
free (file1);
free (file2);
return (retval);

/* Write data to a file. Record whether the last byte written was a newline. Optionally compute a checksum. This is called by patch_file via RCS_checkout. */
static void
patch_file_write (callerdat, buffer, len)
{
  void *callerdat;
  const char *buffer;
  size len;

  struct patch_file_data *data = (struct patch_file_data *) callerdat;

  if (fwrite (buffer, 1, len, data->fp) != len)
    error (1, errno, "cannot write %s", data->filename);
  data->final_ml = (buffer[len - 1] == 'n');
  if (data->compute_checksum)
    MD5Update (&data->context, (unsigned char *) buffer, len);
}

endif /* SERVER_SUPPORT */

/* * Several of the types we process only print a bit of information consisting * of a single letter and the name. */
static void
write_letter (finfo, letter)
{
  struct file_info *finfo;
  int letter;

  if (!really_quiet)
  {
    char *tag = NULL; /* Big enough for "+updated" or any of its ilk. */
    char buf[80];

    switch (letter)
    {
      case 'U':
        tag = "updated";
        break;
      default:
        /* We don't yet support tagged output except for "U". */
        break;
    }

    if (tag != NULL)
    {
      sprintf (buf, "%s", tag);
      cvs_output_tagged (buf, NULL);
    }
  }

  buf[0] = letter;
  buf[1] = ' ';
  buf[2] = '0';
  cvs_output_tagged ("text", buf);
  cvs_output_tagged ("fname", finfo->fullname);
  cvs_output_tagged ("newline", NULL);
  if (tag != NULL)
  {

static int
merge_file (finfo, vers)
struct file_info *finfo;
VersionTS vers;
{
    char *backup;
    int status;
    int retcode = 0;
    int retval;

    /*
     * The users currently modified file is moved to a backup file name
     * "$filename.version", so that it will stay around for a few days
     * before being automatically removed by some cron daemon. The "version"
     * is the version of the file that the user was last up-to-date with
     * before the merge.
     */
    backup = xmalloc (strlen (finfo->file) + strlen (VERS yup) + BAKPREFIX + 10);

    (void) sprintf (backup, "%s\%s.%s", BAKPREFIX, finfo->file, vers->vn_user);

    (void) unlink (finfo->file);
    copy_file (finfo->file, backup);
    chmod (finfo->file, 1);

    if (strcmp (vers->options, "-s") == 0
        || wrapmerge & copy (finfo->file)
        || special & mismatch (finfo, NULL, vers->vn_user))
    {
        /*
         * For binary files, a merge is always a conflict. Same for
         * files whose permissions or linkage do not match. We give the
         * user the two files, and let them resolve it. It is possible
         * that we should require a "touch foo" or similar step before
         * we allow a checkin. *
         */

        /*
         * TODO: it may not always be necessary to regard a permission
         * mismatch as a conflict. The working file and the RCS file
         * have a common ancestor 'A'; if the working file's permissions
         * match A's, then it's probably safe to overwrite them with the
         * RCS permissions. Only if the working file, the RCS file, and
         * A all disagree should this be considered a conflict. But more
         * thought needs to go into this, and in the meantime it is safe
         * to treat any such mismatch as an automatic conflict. -tup */
         */

        symlink SERVER_SUPPORT
        if (server_active)
            servercopy_file (finfo->file, finfo->update_dir, 
                            finfo->repository, backup);
        endif

    status = checkout_file (finfo, vers, 0, 1, 1);

    /* Is there a better term than "nonmergeable file"? What we
    really mean is, not something that CVS cannot or does not
    want to merge (there might be an external manual or
    automatic merge process). */
    error (0, 0, "nonmergeable file needs merge");
    error (0, 0, "revision %s from repository is now in %s", 
           vers->vn_rcs, finfo->fullname);
    error (0, 0, "file from working directory is now in %s", 
           backup);
    write_letter (finfo, "C");
    history_write ("C", finfo->update_dir, vers->vn_rcs, finfo->file, 
                  finfo->repository);
    retval = 0;
    goto out;
}

status = RCS_merge (finfo->rcs, vers->srcfile->path, finfo->file, 
                    vers->options, vers->vn_user, vers->vn_rcs);
if (status != 0 & & status != 1)
{
    error (0, status == -1 ? 0 : 0,
           "could not merge revision %s of %s", vers->vn_user, finfo->fullname);
    error (status == -1 ? 1 : 0, 0, "restoring %s from backup file %s", 
           finfo->fullname, backup);
    rename_file (backup, finfo->file);
    retval = 1;
}
goto out;
}

if (strcmp (vers->options, "-T") == 0)
    vers->options[0] = '\t';

/* This file is the result of a merge, which means that it has been modified. We use a special timestamp string which will not compare equal to any actual timestamp. */
{
    char *cp = 0;

1990
    if (status)
    {
        (void) time (&last_write_time);
        cp = timestamp (finfo->file);
    }

    Register (finfo->entries, finfo->file, vers->vn_base,
        "result of merge", vers->options, vers->tag,
        vers->date, cp, CVSroot_directory, finfo->repository);
    if (cp)
        free (cp);
    }

/* fix up the vers structure, in case it is used by join */
if (join_now)
    {
        if (vers->vn_user != NULL)
            free (vers->vn_user);
        vers->vn_user = xstrdup (vers->vn_base);
    }

2000

siendif SERVER_SUPPORT

/ Send the new contents of the file before the message. If we wanted to be totally correct, we would have the client write the message only after the file has safely been written. */
if (server_active)
    {
        server_copyfile (finfo->file, finfo->update_dir, finfo->repository, backup);
        server_updated (finfo, vers, SERVER_MERGED.);
        (mode - 1, (unsigned char *) NULL, (struct buffer *) NULL);
    }

sendif

if (!noexec && bcmp (backup, finfo->file))
    {
        printf ("%s already contains the differences between %s and %s\n", 
            finfo->fullname, vers->vn_user, vers->vn_base);
        history_write ('G', finfo->update_dir, vers->vn_base, finfo->file,
            finfo->repository);
        retval = 0;
        goto out;
    }

2030
    if (status == 1)
    {
        if (noexec)
            error (0, 0, "conflicts found in %s", finfo->fullname);
        write_letter (finfo, 'C');
        history_write ('C', finfo->update_dir, vers->vn_base, finfo->file, 
            finfo->repository);
    }
    else if (retcode == -1)
    {
        error (1, errno, "fork failed while examining update of %s", 
            finfo->fullname);
    }
2040
    else
    {
        write_letter (finfo, 'M');
        history_write ('M', finfo->update_dir, vers->vn_base, finfo->file, 
            finfo->repository);
    } 

    retval = 0;
out:
    free (backup);
    return retval;

2050
}

/*
 * Do all the magic associated with a file which needs to be joined
 * (-j option)
 */

static void
join_file (finfo, vers)
    struct file *info
;
{ VersFS *vers;
  char *backup;
  char *options;
  int status;
  char *rev1;
  char *rev2;
  char *jrev1;
  char *jrev2;
  char *jdate1;
  char *jdate2;
  jrev1 = join(rev1);
  jrev2 = join(rev2);
  jdate1 = date(rev1);
  jdate2 = date(rev2);

  /* Determine if we need to do anything at all. */
  if (vers->arcfile == NULL ||
      vers->arcfile->path == NULL)
    return;

  /* Only one join tag is specified, it becomes the second revision. */
  if (jrev2 == NULL)
    { jrev2 = jrev1;
      jrev1 = NULL;
      jdate2 = jdate1;
      jdate1 = NULL;
    }

  /* Convert the second revision, walking branches and dates. */
  rev2 = RCSgetversion(vers->arcfile, jrev2, jdate2, 1, (int *) NULL);

  /* If this is a merge of two revisions, get the first revision.
   * If only one join tag was specified, then the first revision is
   * the greatest common ancestor of the second revision and the
   * working file. */
  if (jrev1 != NULL)
    rev1 = RCSgetversion(vers->arcfile, jrev1, jdate1, 1, (int *) NULL);
  else
    { /* Note that we use vn_rcs here, since vn_user may contain a
       special string such as "nn". */
      if (vers->vn_rcs == NULL)
        rev1 = NULL;
      else if (rev2 == NULL)
        { /* This means that the file never existed on the branch.
           It does not mean that the file was removed on the
           branch: that case is represented by a dead rev2. If
           the file never existed on the branch, then we have
           nothing to merge, so we just return. */
          return;
        } else
        rev1 = gca(vers->vn_rcs, rev2);
      }

  /* Handle a nonexistent or dead merge target. */
  if (rev2 == NULL || RCSisdead(vers->arcfile, rev2))
    { char *mrev;
      if (rev2 != NULL)
        free(rev2);
      /* If the first revision doesn't exist either, then there is
       * no change between the two revisions, so we don't do
       * anything. */
      if (rev1 != NULL || RCSisdead(vers->arcfile, rev1))
        { if (rev1 != NULL)
            free(rev1);
            return;
        }
    }

  /* If we are merging two revisions, then the file was removed
   * between the first revision and the second one. In this case
   * we want to mark the file for removal.
   * If we are merging one revision, then the file has been
   * removed between the greatest common ancestor and the merge
   * revision. From the perspective of the branch on to which
   * we are merging, which may be the trunk, either 1) the file
   * does not currently exist on the target, or 2) the file has
726 – Source code

not been modified on the target branch since the greatest common ancestor, or 3) the file has been modified on the target branch since the greatest common ancestor. In case 1 there is nothing to do. In case 2 we mark the file for removal. In case 3 we have a conflict.

Note that the handling is slightly different depending upon whether one or two join targets were specified. If two join targets were specified, we don’t check whether the file was modified since a given point. My reasoning is that if you ask for an explicit merge between two tags, then you want to merge in whatever was changed between those two tags. If a file was removed between the two tags, then you want it to be removed. However, if you ask for a merge of a branch, then you want to merge in all changes which were made on the branch. If a file was removed on the branch, that is a change to the file. If the file was also changed on the main line, then that is also a change. These two changes—the file removal and the modification—must be merged. This is a conflict.

If the user file is dead, or does not exist, or has been marked for removal, then there is nothing to do.

If the user file has been marked for addition, or has been locally modified, then we have a conflict which we can not resolve. No Difference will already have been called in this case, so comparing the timestamps is sufficient to determine whether the file is locally modified.

If only one join tag was specified, and the user file has been changed since the greatest common ancestor (rev1), then there is a conflict we can not resolve. See above for the rationale.

The user file exists and has not been modified. Mark it for removal. FIXME: If we are doing a checkout, this has the effect of first checking out the file, and then
removing it. It would be better to just register the removal. */

ifdef SERVER_SUPPORT

if (server->active)
{
server->scratch (info->file);
server->updated (info, vers, SERVER_UPDATED, (mode & RCS_dead) != 0,
(unsigned char *) NULL, (struct buffer *) NULL);
}
endif

mrev = malloc (strlen (vers->vn_user) + 2);
sprintf (mrev, "%s", vers->vn_user);
Register (info->entries, info->file, mrev, vers->ts, vers->options, vers->tag, vers->date, vers->conflict, CVSroot->directory, info->repository);
free (mrev);
/* We need to check existence error here because if we are
running as the server, and the file is up to date in the
working directory, the client will not have sent us a copy. */
if (unlinkfile (info->file) <= 0 & & ! existence error (errno))
error (0, errno, "cannot remove file ", info->fullname);

ifdef SERVER_SUPPORT

if (server->quiet)
server->checked_in (info->file, info->update_dir, info->repository);
endif

if (! really_quiet)
error (0, 0, "scheduling %s for removal", info->fullname);
return;
}

/* If the target of the merge is the same as the working file
revision, then there is nothing to do. */
if (vers->vn_user != NULL & & strcmp (rev2, vers->vn_user) == 0)
{
if (rev1 != NULL)
free (rev1);
free (rev2);
return;
}

/* If rev1 is dead or does not exist, then the file was added
between rev1 and rev2. */
if (rev1 == NULL || RCS_dead (vers->srcfile, rev1))
{
if (rev1 != NULL)
free (rev1);
free (rev2);
/* If the file does not exist in the working directory, then
we can just check out the new revision and mark it for
addition. */
if (vers->vn_user == NULL)
{
Vers_TS *xvers;

xvers = Version_TS (info, vers->options, jrev2, jdate2, 1, 0);
/* FIXME: If checkout_file fails, we should arrange to
return a non-zero exit status. */
status = checkout_file (info, xvers, 1, 0, 1);
freevers (xvers);
return;
}
/* The file currently exists in the working directory, so we
have a conflict which we can not resolve. Note that this
is true even if the file is marked for addition or removal. */
if (jdate2 != NULL)
error (0, 0,
"file \%s exists, but has been added in revision \%s as of \%s",
info->fullname, jrev2, jdate2);
else
error (0, 0,
"file \%s exists, but has been added in revision \%s",
info->fullname, jrev2);
return;
}

/* If the two merge revisions are the same, then there is nothing
to do. */
if (strcmp (rev1, rev2) == 0)
{
free (rev1);
free (rev2);
return;
}
2340 */ If there is no working file, then we can't do the merge. */
if (vers->vn_user == NULL)
{
  free (rev1);
  free (rev2);
  if (jdate2 != NULL)
    error (0, 0,
      "File %s is present in revision %s as of %s",
      file->fullname, jrev2, jdate2);
  else
    error (0, 0,
      "File %s is present in revision %s",
      file->fullname, jrev2);
  /* FIXME: Should we arrange to return a non-zero exit status? */
  return;
}

2360#endif

2380

2380 /* The user currently modified file is moved to a backup file name
2380  * "filename.version", so that it will stay around for a few days
2380  * while being automatically removed by some cron daemon. The "version"
2380  * is the version of the file that the user was most up-to-date with
2380  * before the merge.
2380  */
2380 backup = xmalloc (strlen (file->filename) + 10);
2380   + strlen (vers->vn_user) + xmalloc (BAKPREFIX) + 10);
2380   (void) fprintf (backup, "%s%s.%s",
2380                    file->filename, vers->vn_user, BAKPREFIX);,
2380   (void) unlink (backup);
2380   copy_file (file->filename, backup);
2380   chmod (file->filename, 1),
2380   options = vers->options;
2380   if (options == \0"]")
2380     options = "-kk"; /* to ignore keyword expansions */

2400 /* If the source of the merge is the same as the working file
2400  * revision, then we can just RCS_checkout the target (no merging
2400  * as such). In the text file case, this is probably quite
2400  * similar to the RCS_merge, but in the binary file case,
2400  * RCS_merge gives all kinds of trouble. */
2400 if (vers->vn_user == NULL
2400   && strcmp (rev1, vers->vn_user) == 0
2400   /* See comments above about how Nodiff has already been
2400   */
2400   && vers->ts_user != NULL
2400   && strcmp (vers->ts_user, vers->ts_rcs) == 0
2400
2420 /* This is because of the worry below about $Name. If that
2420  * isn't a problem, I suspect this code probably works for
2420  * text files too. */
2420 if (strcmp (options, "-回避") == 0
2420   || wrap_merge_in_copy (file->filename))
2420{
2420  /* FIXME: what about nametag? What does RCS_merge do with
2420     $Name? */
2420  if (RCS_checkout (file->rcs, file->filename, rev2, NULL, options,
2420                    RUN_TTY, (RCSCHECKOUTPROC)0, NULL) != 0)
2420   status = 2, /* OK, this is really stupid. RCS_checkout carefully removes
2420               write permissions, and we carefully put them back. But
2420               until someone gets around to fixing it, that seems like the
2420               easiest way to get what would seem to be the right mode.
2420 else
2420   status = 0;
I don't check CVSWRITE or watched; I haven't thought about that in great detail, but it seems like a watched file should be checked out (writable) after a merge. */
xchmod (finfo->file, 1);
/* Traditionally, the text file case prints a whole bunch of scary looking and verbose output which fails to tell the user what is really going on (it gives them rev1 and rev2 but doesn't indicate in any way that rev1 == vn(user). I think just a simple "U foo" is good here; it seems analogous to the case in which the file was added on the branch in terms of what to print. */
write_letter (finfo, "U");
} else if (strcmp (options, "-kb") == 0
  || write_merge (“copy” (finfo->file)
  || special_file_mismatch (finfo, rev1, rev2))
{
/* We are dealing with binary files, or files with a permission/linkage mismatch, and real merging would need to take place. This is a conflict. We give the user the two files, and let them resolve it. It is possible that we should require a "touch foo" or similar step before we allow a checkin. */
if (RCS_checkout (finfo->rcs, finfo->file, rev2, NULL, options,
  RUN_TTY, (RCSCHECKOUTPROC, NULL) != 0)
  status = 2;
else
  status = 0;
/* OK, this is really stupid. RCS_checkout carefully removes write permissions, and we carefully put them back. But until someone gets around to fixing it, that seems like the easiest way to get what would seem to be the right mode. I don't check CVSWRITE or watched; I haven't thought about that in great detail, but it seems like a watched file should be checked out (writable) after a merge. */
xchmod (finfo->file, 1);
/* Hmm. We don't give them REV1 anywhere. I guess most people probably don't have a 3-way merge tool for the file type in question, and might just get confused if we tried to either provide them with a copy of the file from REV1, or even just told them what REV1 is so they can get it themselves, but it might be worth thinking about. */
/* See comment in merge_file about the "nonmergeable file" terminology. */
error (0, 0, "nonmergeable file needs merge");
error (0, 0, "revision %s from repository is now in %s",
  rev2, info->fullname);
error (0, 0, "file from working directory is now in %s", backup);
write_letter (finfo, "C");
} else
  status = RCS_merge (finfo->rcs, vers->srcfile->path, finfo->file,
  options, rev1, rev2);
if (status != 0 || status != 1)
{
  error (0, status == -1 ? errno : 0,
    "could not merge revision %s of %s", rev2, finfo->fullname);
  error (status == -1 ? 1 : 0, 0, "restoring %s from backup file %s",
    finfo->fullname, backup);
  rename_file (backup, finfo->file);
  free (rev1);
  free (rev2);
/* The file has changed, but if we just checked it out it may still have the same timestamp it did when it was first registered above in checkout_file. We register it again with a dummy timestamp to make sure that later runs of CVS will recognize that it has changed.
We don't actually need to register again if we called RCS_checkout above, and we aren't running as the server. However, that is not the normal case, and calling Register again won't cost much in that case. */
  char *cp = 0;
if (status)
  {
    (void) time (&last_register_time);
    cp = time_stamp (finfo->file);
    Register (finfo->entries, finfo->file, vers->rcs, 
      "result of merge", vers->options, vers->tag,
      vers->date, cp, CVStree->directory, finfo->repository);
    if (cp)
free(cp);

#ifdef SERVER_SUPPORT
if (server_active)
{
    server_copyfile (finfo->file, info->file, info->repository, backup);
    server_updated (info, vers, SERVER_MERGED,
                    (mode) - 1, (unsigned char *) NULL,
                    (struct buffer *) NULL);
}
#endif
free (backup);
}

/*
 * Report whether revisions REV1 and REV2 of FINFO agree on:
 * . file ownership
 * . permissions
 * . major and minor device numbers
 * . mode
 * . hard links
 * . symbolic links
 * If either REV1 or REV2 is NULL, the working copy is used instead.
 * Return 1 if the files differ on these data.
 */
int specialfile_mismatch (finfo, rev1, rev2)
struct file_info *finfo,
char *rev1,
char *rev2;
#endif

#ifdef PRESERVE_PERMISSIONS_SUPPORT
struct stat sb;
RCSVers *vp;
Node *n;
uid_t rev1_uid, rev2_uid;
gid_t rev1_gid, rev2_gid;
mode_t rev1_mode, rev2_mode;
unsigned long dev_long;
dev_t rev1_dev, rev2_dev;
char rev1_symlink = NULL;
char rev2_symlink = NULL;
List *rev1_hardlinks;
List *rev2_hardlinks;
int check_uids, check_gids, check_modes;
int result;
#endif

/* If we don't care about special file info, then
 don't report a mismatch in any case. */
if (!preserve_perm)
    return 0;

/* When specialfile_mismatch is called from NoDifference, the
 RCS file has been only partially parsed. We must read the
 delta tree in order to compare special file info recorded in
 the delta nodes.  (I think this is safe. -twp) */
if (finfo->rcs->flags & PARTIAL)
    RCS_reparsefile (finfo->rcs, NULL, NULL);
    check_uids = check_gids = check_modes = 1;

/* Obtain file information for REV1.  If this is null, then stat
finfo->file and use that info. */
if (rev1 == NULL)
{
    if (islink (finfo->file))
        rev1_symlink = xreadlink (finfo->file);
    else
    {
        if (CVS_LSTAT (finfo->file, &sb) < 0)
            error (1, errno, "could not get file information for %a",
                   info->file);
        rev1_uid = sb->uid;
        rev1_gid = sb->gid;
        rev1_mode = sb->mode;
        if (S_ISBLK (rev1_mode) || S_ISCHR (rev1_mode))
            rev1_dev = sb->dev;
        else
            rev1_hardlinks = listlinked_files_on_link (finfo->file);
    }

    n = findnode (finfo->rcs->versions, rev1);

    /*
     * Compare the file status.
     */
    if (check_uids)
        if (rev1_uid != sb->uid)
            return 1;
    else
        return 0;

    if (check_gids)
        if (rev1_gid != sb->gid)
            return 1;
    else
        return 0;

    if (check_modes)
        if (rev1_mode != sb->mode)
            return 1;
    else
        return 0;
#endif

/* Compare the hard link count.
*/
if (check_hardlinks)
    if (rev1_hardlinks != sb->hardlinks)
        return 1;
else
    return 0;

/* Compare the symbolic link count.
*/
if (check_symlinks)
    if (rev1_symlink != sb->symlink)
vp = (RCSVers *) n->data;

n = findnode (vp->other, "symlink");
if (n != NULL)
    rev1->symlink = xstrdup (n->data);
else
    n = findnode (vp->other, "cswr");
    if (n == NULL)
        check_uids = 0; /* don't care */
    else
        rev1->uid = strtol (n->data, NULL, 10);

n = findnode (vp->other, "group");
if (n == NULL)
    check_gids = 0; /* don't care */
else
    rev1->gid = strtol (n->data, NULL, 10);

n = findnode (vp->other, "permissions");
if (n == NULL)
    check_modes = 0; /* don't care */
else
    rev1->mode = strtol (n->data, NULL, 8);

n = findnode (vp->other, "special");
if (n == NULL)
    rev1->mode |= S_IFREG;
else
    /* If the size of 'ftype' changes, fix the sscanf call also */
    char *ftype[16];
    if (sscanf (n->data, "%s %lu", ftype, &dev) < 2)
        error (1, 0, "%s: %s has bad 'special' neophase %s",
               finfo->file, rev1, n->data);
    rev1->dev = dev;
    if (strcmp (ftype[0], "character") == 0)
        rev1->mode |= S_IFCHR;
    else if (strcmp (ftype[0], "block") == 0)
        rev1->mode |= S_IFBLK;
    else
        error (1, 0, "%s: %s unknown file type '%s'",
               finfo->file, rev1, ftype);

    rev1->hardlinks = vp->hardlinks;
    if (rev1->hardlinks == NULL)
        rev1->hardlinks = getlist();
}

if (rev2 == NULL)
    if (islink (finfo->file))
        rev2->symlink = xreadlink (finfo->file);
    else
        if (CVS.PathVariable (finfo->file, &sb) < 0)
            error (1, errno, "could not get file information for %s",
                   finfo->file);
        rev2->uid = sb->uid;
        rev2->gid = sb->gid;
        rev2->mode = sb->mode;
        if (S_IFBLK (rev2->mode) || S_IFCHR (rev2->mode))
            rev2->dev = sb->dev;
    rev2->hardlinks = listlinked_file_con_link (finfo->file);
else
    n = findnode (finfo->rcs->versions, rev2);
    vp = (RCSVers *) n->data;

    n = findnode (vp->other, "symlink");
    if (n != NULL)
        rev2->symlink = xstrdup (n->data);
    else
        n = findnode (vp->other, "cswr");
        if (n == NULL)
            check_uids = 0; /* don't care */
        else
            rev2->uid = strtol (n->data, NULL, 10);

    n = findnode (vp->other, "group");
    if (n == NULL)
        check_gids = 0; /* don't care */
    else
        rev2->gid = strtol (n->data, NULL, 10);

    n = findnode (vp->other, "permissions");
    if (n == NULL)
        check_modes = 0; /* don't care */
    else
        rev2->mode = strtol (n->data, NULL, 8);

    n = findnode (vp->other, "special");
    if (n == NULL)
        rev2->mode |= S_IFREG;
    else
        /* If the size of 'ftype' changes, fix the sscanf call also */
        char *ftype[16];
        if (sscanf (n->data, "%s %lu", ftype, &dev) < 2)
            error (1, 0, "%s: %s has bad 'special' neophase %s",
                   finfo->file, rev2, n->data);
        rev2->dev = dev;
        if (strcmp (ftype[0], "character") == 0)
            rev2->mode |= S_IFCHR;
        else if (strcmp (ftype[0], "block") == 0)
            rev2->mode |= S_IFBLK;
        else
            error (1, 0, "%s: %s unknown file type '%s'",
                   finfo->file, rev2, ftype);

        rev2->hardlinks = vp->hardlinks;
        if (rev2->hardlinks == NULL)
            rev2->hardlinks = getlist();
}
rev2_sgid = strtol (n->data, NULL, 10);

n = findnode (vp->other_delta, "permissions");
if (n == NULL)
    check_modes = 0; /* don't care */
else
    rev2_mode = strtol (n->data, NULL, 8);

n = findnode (vp->other_delta, "special");
if (n == NULL)
    rev2_mode |= S_IFREG;
else
{
    /* If the size of 'ftype' changes, fix the scanf call also */
    char ftype[10];
    if (scanf (n->data, "%s %lu", ftype, &dev) < 2)
        error (1, 0, "%s: has bad 'special' newphrase %s",
               info->file, rev2, n->data);
    rev2_dev = dev;
    if (strcmp (ftype, "character") == 0)
        rev2_mode |= S_IFCHR;
    else if (strcmp (ftype, "block") == 0)
        rev2_mode |= S_IFBLK;
    else
        error (0, 0, "%s: unknown file type '%s'.
               info->file, rev2, ftype);

    rev2_hardlinks = vp->hardlinks;
    if (rev2_hardlinks == NULL)
        rev2_hardlinks = getlist();
}

/* Check the user/group ownerships and file permissions, printing
   an error for each mismatch found. Return 0 if all characteristics
   matched, or 1 otherwise. */
result = 0;

/* Compare symlinks first, since symlinks are simpler (don't have
any other characteristics). */
if (rev1_sypath != NULL && rev2_sypath == NULL)
{
    error (0, 0, "%s is a symbolic link",
           rev1 == NULL ? "working file" : rev1);
    result = 1;
}
else if (rev1_sypath == NULL && rev2_sypath != NULL)
{
    error (0, 0, "%s is a symbolic link",
           rev2 == NULL ? "working file" : rev2);
    result = 1;
}
else if (rev1_sypath != NULL && rev2_sypath != NULL)
    result = (strcmp (rev1_sypath, rev2_sypath) == 0);
else
{
    /* Compare user ownership. */
    if (check_uids && rev1_uid != rev2_uid)
        error (0, 0, "%s: user mismatch between %s and %s",
               info->file, (rev1 == NULL ? "working file" : rev1),
               (rev2 == NULL ? "working file" : rev2));
    result = 1;
}

/* Compare group ownership. */
if (check_gids && rev1_gid != rev2_gid)
{
    error (0, 0, "%s: group mismatch between %s and %s",
           info->file, (rev1 == NULL ? "working file" : rev1),
           (rev2 == NULL ? "working file" : rev2));
    result = 1;
}

/* Compare permissions. */
if (check_modes &&
     (rev1_mode & 07777) != (rev2_mode & 07777))
{
    error (0, 0, "%s: permission mismatch between %s and %s",
           info->file, (rev1 == NULL ? "working file" : rev1),
           (rev2 == NULL ? "working file" : rev2));
    result = 1;
}
/** Compare device file characteristics. */

if((rev1_mode & S_IFMT) != (rev2_mode & S_IFMT))
{
    error (0, 0, "3a: %s and %s are different file types",
      finfo->file,
      (rev1 == NULL ? "working file" : rev1),
      (rev2 == NULL ? "working file" : rev2));
    result = 1;
}
else if(S_ISBLK (rev1_mode))
{
    if((rev1_dev != rev2_dev))
    {
        error (0, 0, "3a: device numbers of %s and %s do not match",
          finfo->file,
          (rev1 == NULL ? "working file" : rev1),
          (rev2 == NULL ? "working file" : rev2));
        result = 1;
    }
}

/* Compare hard links. */
if(compare_linkage_lists (rev1_hardlinks, rev2_hardlinks) == 0)
{
    error (0, 0, "3a: hard linkage of %s and %s do not match",
      finfo->file,
      (rev1 == NULL ? "working file" : rev1),
      (rev2 == NULL ? "working file" : rev2));
    result = 1;
}

if (rev1_syalink != NULL)
    free (rev1_syalink);
if (rev2_syalink != NULL)
    free (rev2_syalink);
if (rev1_hardlinks != NULL)
    dellist (&rev1_hardlinks);
if (rev2_hardlinks != NULL)
    dellist (&rev2_hardlinks);

return result;
#else
    return 0;
#endif

int
joining ()
{
    return (join(rev1 != NULL);
}
A.61 update.h

/* Declarations for update.c.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

int do_update PROTO((int argc, char **argv[], char **options, char **xtag,
                    char **xdate, int xforce, int local, int xbuild,
                    int xtag, int xprune, int xpipeout, int which,
                    char **xjoin_rev1, char **xjoin_rev2, char *preload_update_dir));
int joining PROTO((void));
extern int isemptydir PROTO((char *dir, int might_not_exist));
A.62 vers_ts.c

/* Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"

#define SERVER_SUPPORT

static void timestamp(server Proto(char *, VersTS *, Entnode *));
#endif

/* Fill in and return a VersTS structure for the file FINFO. TAG and
 * DATE are from the command line. */

VersTS *
VersionTS (finfo, options, tag, date, force_tag_match, set_time)

struct fileinfo *finfo;

/* Keyword expansion options, I think generally from the command
 * line. Can be either NULL or "" to indicate none are specified
 * here. */
char *options;
char *tag;
char *date;
int force_tag_match;
int set_time;

{ /* get a new VersTS struct */
  VersTS = (VersTS *) xmalloc (sizeof (VersTS));
  memset ((char *) VersTS, 0, sizeof (*VersTS));
}

/* look up the entries file entry and fill in the version and timestamp
 * if entries is NULL, there is no entries file so don't bother trying to
 * look up the entries file entry and fill in the version and timestamp
 */
if (finfo->entries == NULL)
{
  sdtp = NULL;
  p = NULL;
}
else
{
  p = findnode_in (finfo->entries, finfo->file);
  sdtp = (struct stickydirtag *) finfo->entries->list->data; /* list-private */
}

entdata = NULL;
if (p != NULL)
{
  entdata = (Entnode *) p->data;
  if (entdata->type == ENT_SUBDIR)
  {
    /* According to cvs.texinfo, the various fields in the Entries
     * file for a directory (other than the name) do not have a
     * defined meaning. We need to pass them along without getting
     * confused based on what is in them. Therefore we make sure
     * not to set vn_user and the like from Entries, add.c and
     * perhaps other code will expect these fields to be NULL for
     * a directory. */
    /*
     */
    vers->entdata = entdata;
  }
  else
  #define SERVER_SUPPORT
  /* An entries line with "D" in the timestamp indicates that the
   * client sent ls-modified without sending Entry. So we want to
   * use the entries line for the sole purpose of telling
   * timestamp_server what is up; we don't want the rest of CVS
   * to think there is an entries line. */

  if (strcmp (entdata->timestamp, "D") != 0)
  #endif
  
  
  vers->vn_user = xstrdup (entdata->version);
  vers->ts_rcs = xstrdup (entdata->timestamp);
  vers->ts_conflict = xstrdup (entdata->conflict);
  if (tag)
  {
    if ((isdtp && sdtp->aflag))
vers_t->tag = xstrdup (entdata->tag);

if (date)
{
    if ((sdtp & sdtp->aflag))
        vers_t->date = xstrdup (entdata->date);
    vers_t->entdata = entdata;
}

/* Even if we don't have an "entries line" as such
   (vers_t->entdata), we want to pick up options which could
   have been from a Rept protocol request. */
if (options || (options & options == '\n'))
{
    if ((sdtp & sdtp->aflag))
        vers_t->options = xstrdup (entdata->options);
}

/* -k options specified on the command line override (and overwrite)
 * options stored in the entries file */
if (options & options != '\n')
    vers_t->options = xstrdup (options);
else if (vers_t->options || vers_t->options == '\n')
{
    if (finfo->rcs != NULL)
    {
        /* If no keyword expansion was specified on command line,
         * use whatever was in the rcs file (if there is one). This
         * is how we, if we are the server, tell the client whether
         * a file is binary. */
        char *rcsexpand = RCS_getexpand (finfo->rcs);
        if (rcsexpand != NULL)
        {
            vers_t->options = xmalloc (strlen (rcsexpand) + 3);
            strcpy (vers_t->options, "-k");
            strcat (vers_t->options, rcsexpand);
        }
    }
    if (vers_t->options)
        vers_t->options = xstrdup ("-k");
    /* if tags were specified on the command line, they override what is in
     * the Entries file */
    if (tag || date)
    {
        vers_t->tag = xstrdup (tag);
        vers_t->date = xstrdup (date);
    }
    else if (vers_t->entdata & sdtp & sdtp->aflag == 0)
    {
        if (vers_t->tag)
        {
            vers_t->tag = xstrdup (sdtp->tag);
            vers_t->nonbranch = sdtp->nonbranch;
        }
        else if (vers_t->date)
            vers_t->date = xstrdup (sdtp->date);
    }

    /* Now look up the info on the source controlled file */
    if (finfo->rcs != NULL)
    {
        rcsdata = finfo->rcs;
        rcsdata->reccount++;}
else if (finfo->repository != NULL)
    rcsdata = RCS_parse (finfo->file, finfo->repository);
else
    rcsdata = NULL;

    if (rcsdata != NULL)
    {
        /* squirrel away the rcsdata pointer for others */
        vers_t->arcfile = rcsdata;
    }

    if (vers_t->tag & & strcmp (vers_t->tag, TAG_BASE) == 0)
    {
        vers_t->vn_rcs = xstrdup (vers_t->vn_user);
        vers_t->vn_tag = xstrdup (vers_t->vn_user);
    }
    else
    {
        int simple;
        char *local_rév = NULL;
/* We try remote first, because if the version is remote, and branched 
ad at a local revision, we want to return remote, not local */

ifdef SERVER

versTs->vn_rcs = RCSgetversioninfo (info, rcsdata, versTs->tag, &localRev);

if (versTs->vn_rcs == NULL) {
  /* Not remote, try regular (local) */
  if (localRev != NULL) {
    versTs->tag = localRev;
  }

  versTs->vn_rcs = RCSgetversion (rcsdata, versTs->tag, 
                                 versTs->date, forced_match, 
                                 &simple);

  if (versTs->vn_rcs == NULL) {
    versTs->vn_tag = NULL;
  } else if (simple)
    versTs->vn_tag = xstrdup (versTs->tag);
  else
    versTs->vn_tag = xstrdup (versTs->vn_rcs);

  /* Found remote, forget about local */
  if (versTs->vn_rcs != NULL) {
    versTs->vn_remote = xstrdup (versTs->vn_rcs);
  }
}
endif

/* If the source control file exists and has the requested revision, 
* get the Date the revision was checked in. If "user" exists, set 
* its mtime.
*/

if (setTime & versTs->vn_rcs != NULL)
  defineserverSupport
  if (serverActive)
    serverModtime (info, versTs);
  else
    endif

  { struct utimbuf t;
  
    memset (&t, 0, sizeof (t));
    t.mtime = RCSgetmtime (rcsdata, versTs->vn_rcs, 0, 0);
    if (t.mtime != (time_t) − 1)
      { t.actime = t.mtime;
        /* This used to need to ignore existence_errors 
         * (for cases like where update.c now clears 
         * setTime if noexec, but didn't used to), 
         * think maybe now it doesn't (server_modtime does 
         * not like those kinds of cases). */
        (void) utime (info->file, &t);
      }
  }
endif

/* get user file time-stamp in ts_user */

if (info->entries != (List *) NULL)
  defineserverSupport
  if (serverActive)
    tsmtimeServer (info->file, versTs, entdata);
  else
    endif

  versTs->ts_user = timestamp (info->file);
}

return (versTs);
#endif /* SERVER_SUPPORT */

/* Set TS_USER to time stamp for FILE. */
/* Separate these out to keep the logic below clearer. */
#define markLost (V) (V)->ts_user = 0
#define markUnchanged (V) (V)->ts_user = xstrdup ((V)->ts_cio))

static void
    timestamp (file, versTs, entdata)
    { struct stat sb;
      char *file;
      versTs oversTs;
      Entnode *entdata;
      
      struct utimbuf t;
      memset (&t, 0, sizeof (t));
      t.mtime = RCSgetmtime (rcsdata, versTs->vn_rcs, 0, 0);
      if (t.mtime != (time_t) − 1)
        { t.actime = t.mtime;
      /* This used to need to ignore existence_errors  
         * (for cases like where update.c now clears  
         * setTime if noexec, but didn't used to),  
         * think maybe now it doesn't (server_modtime does  
         * not like those kinds of cases). */
        (void) utime (info->file, &t);
      }
      
      /* get user file time-stamp in ts_user */
      if (info->entries != (List *) NULL)
        defineserverSupport
          if (serverActive)
            tsmtimeServer (info->file, versTs, entdata);
          else
            endif

          versTs->ts_user = timestamp (info->file);
      }
      return (versTs);
    } /* timestamp */

    defineserverSupport
      /* Set TS_USER to time stamp for FILE. */
      /* Separate these out to keep the logic below clearer. */
      #define markLost (V) (V)->ts_user = 0
      #define markUnchanged (V) (V)->ts_user = xstrdup ((V)->ts_cio))

    static void
      timestamp (file, versTs, entdata)
      { struct stat sb;
        char *file;
        versTs oversTs;
        Entnode *entdata;
        
        struct utimbuf t;
        memset (&t, 0, sizeof (t));
        t.mtime = RCSgetmtime (rcsdata, versTs->vn_rcs, 0, 0);
        if (t.mtime != (time_t) − 1)
          { t.actime = t.mtime;
        /* This used to need to ignore existence_errors  
           * (for cases like where update.c now clears  
           * setTime if noexec, but didn't used to),  
           * think maybe now it doesn't (server_modtime does  
           * not like those kinds of cases). */
          (void) utime (info->file, &t);
        }
      
      /* get user file time-stamp in ts_user */
      if (info->entries != (List *) NULL)
        defineserverSupport
          if (serverActive)
            tsmtimeServer (info->file, versTs, entdata);
          else
            endif

          versTs->ts_user = timestamp (info->file);
      }
      return (versTs);
    } /* timestamp */
if (CVS_LSTAT (file, &sb) < 0)
{
    if (! existence || error (errno))
        error (1, errno, "cannot stat temp file");
/* Missing file means lost or unmodified; check entries
 file to see which.

 XXX FIXME - If there's no entries file line, we
 wouldn't be getting the file at all, so consider it
 lost. I don't know that that's right, but it's not
 clear to me that either choice is. Besides, would we
 have an RCS string in that case anyways? */
if (entdata == NULL)
    mark_lost (vers); //
else if (entdata->timestamp
    && entdata->timestamp[0] == '\n')
    mark_unchanged (vers); //
else if (entdata->timestamp != NULL
    && (entdata->timestamp[0] == 'M'.
    || entdata->timestamp[0] == 'D'.
    || entdata->timestamp[1] == 'D'.)
    vers_ts->ts_user = xstrdup ("is-modified");
else
    mark_lost (vers_ts); //
}
else if (sb.st_mtime == 0) //
{ /* We shouldn't reach this case any more! */
    abort (1);
}
else
{
    struct tm *tm_w,
    struct tm local_tm;

    vers_ts->ts_user = xmalloc (25); //
/* We want to use the same timestamp format as is stored in the
 st_mtime. For unix (and NT I think) this must be universal
 time (UT), so that files don't appear to be modified merely
 because the timezone has changed. For VMS, or hopefully other
 systems where gmtime returns NULL, the modification time is
 stored in local time, and therefore it is not possible to cause
 st_mtime to be out of sync by changing the timezone. */
    tm_w = gmtime ( &sb.st_mtime);
    if (tm_w) //
    {
        memcpy (&local_tm, tm_w, sizeof (local_tm));
        cp = asctime ( &local_tm); // copy in the modify time */
    }
else
    cp = ctime ( &sb.st_mtime);

    cp[24] = 0;
    (void) strcpy (vers_ts->ts_user, cp);
}
} /* SERVERSUPPORT */ //
/* Gets the time-stamp for the file "file" and returns it in space it
 allocates */
/*
 char *
 time_stamp (file)
 {*/
    struct stat sb;
    char *cp;
    char *ts;

    if (CVS_LSTAT (file, &sb) < 0) //
    {
        ts = NULL;
    }
else
{
    struct tm *tm_w;
    struct tm local_tm;
    ts = xmalloc (25); //
/* We want to use the same timestamp format as is stored in the
 st_mtime. For unix (and NT I think) this must be universal
 time (UT), so that files don't appear to be modified merely
 because the timezone has changed. For VMS, or hopefully other
 systems where gmtime returns NULL, the modification time is
 stored in local time, and therefore it is not possible to cause
 st_mtime to be out of sync by changing the timezone. */
    tm_w = gmtime ( &sb.st_mtime);
    if (tm_w)
    {
        memcpy (&local_tm, tm_w, sizeof (local_tm));
        cp = asctime ( &local_tm); // copy in the modify time */
    }
else
    cp = ctime ( &sb.st_mtime);

    cp[24] = 0;
    (void) strcpy (vers_ts->ts_user, cp);
}
{  
  memcpy (&local_tm, tm_p, sizeof (local_tm));
  cp = localtime (&local_tm); /* copy in the modify time */
}  
else
  cp = ctime(&sb.st_mtime);
  cp[24] = 0;
  (void) strcpy (ts, cp);
}

return (ts);
}

/* free up a Vers_TS struct */

void freevers_ts (versp)
  Vers_TS **versp;
{
  if (!versp->srcfile)
    freecnod (&(versp->srcfile));
  if (!versp->vnuser)
    free ((versp)->vn_user);
  if (!versp->vnrcs)
    free ((versp)->vn_rcs);
  if (!versp->vn_tag)
    free ((versp)->vn_tag);
  if (!versp->tuser)
    free ((versp)->tm_user);
  if (!versp->rcs)  
    free ((versp)->tm_rcs);
  if (!versp->options)
    free ((versp)->options);
  if (!versp->tag)
    free ((versp)->tag);
  if (!versp->date)
    free ((versp)->date);
  if (!versp->tagconflict)
    free ((versp)->tm_conflict);
  free ((char *) versp);
  *versp = (Vers_TS *) NULL;
}
A.63  vers_ts.c

/ *
  * Copyright (c) 1992, Brian Berliner and Jeff Polk
  * Copyright (c) 1989-1992, Brian Berliner
  * You may distribute under the terms of the GNU General Public License as
  * specified in the README file that comes with the CVS source distribution.
  */

#include "cvs.h"

#if defined SERVER_SUPPORT
static void timestamp(char * server, PROTO(char *, Vers_TS * , Entnode *))
#endif

/* Fill in and return a Vers_TS structure for the file FINFO. TAG and
 * DATE are from the command line. */

Vers_TS *
Version_TS (finfo, options, tag, date, force_tag_match, set_time)

struct file_info *finfo;

/* Keyword expansion options, I think generally from the command
 * line. Can be either NULL or " " to indicate none are specified
 * here. */

char *options;
char *tag;
char *date;
int force_tag_match;
int set_time;

{ Node *p;
RCSNode *resdata;
Vers_TS *vers Ts;
struct sticky_dirtag *sdtp;
Entnode *entdata;

/* get a new Vers_TS struct */
vers Ts = (Vers_TS *) malloc (sizeof (Vers_TS));
memset ((char *) vers Ts , 0 , sizeof (vers Ts));

*/
/* look up the entries file entry and fill in the version and timestamp
 * if entries is NULL, there is no entries file so don't bother trying to
 * look it up (used by checkout -P)
 */
if (finfo->entries == NULL)
{
  sdtp = NULL;
p = NULL;
}
else
{
p = (file_info *) finfo->entries->file;
sdtp = (struct sticky_dirtag *) finfo->entries->list->data; /* list-private */
entdata = NULL;
if (p != NULL)
{
  entdata = (Entnode *) p->data;
  if (entdata->type == ENT_SUBDIR)
  {
    /* According to cvs.texinfo, the various fields in the Entries
     * file for a directory (other than the name) do not have a
     * defined meaning. We need to pass them along without getting
     * confused based on what is in them. Therefore we make sure
     * not to set vn_user and the like from Entries, add.c and
     * perhaps other code will expect these fields to be NULL for
     * a directory. */
    vers Ts->entdata = entdata;
  }
  else
#endif

/* An entries line with "D" in the timestamp indicates that the
 * client sent Is-modified without sending Entry. So we want to
 * use the entries line for the sole purpose of telling
time stamp server what is up; we don't want the rest of CVS
to think there is an entries line. */

if (strcmp (entdata->timestamp, "D") != 0)
#endif

{ vers Ts->vn_user = xstrdup (entdata->version);
vers Ts->vn_user = xstrdup (entdata->timestamp);
vers Ts->vn_conflict = xstrdup (entdata->conflict);
if (tag)
  { if ((sdtp && sdtp->aflag)
vers->tag = xstrdup (entdata->tag);
}

if ((date)
    { }
    
    else if (vers->date = xstrdup (entdata->date);
    }
    
    /* Even if we don't have an "entries line" as such
     * /vers->entdata = entdata;
    }
    
    /* Even if we don't have an "entries line" as such
     * /vers->entdata = entdata;
    */

    if (options || (options & options == '\0'))
    { 
        if (!sdtp & sdtp->aflag)
            vers->options = xstrdup (entdata->options);
    }
}

/* * `-k` options specified on the command line override (and overwrite)
 * */

/* options stored in the entries file */
if (options & options != ' 

    vers->options = xstrdup (options);
else if (vers->options || vers->options == ' 

    if (finfo->rcs != NULL)
        { 
            /* If no keyword expansion was specified on command line,
             * use whatever was in the rcs file (if there is one). This
             * is how we, if we are the server, tell the client whether
             * a file is binary. */
            char *rcsexpand = RCSgetexpansion (finfo->rcs);
            if (rcsexpand != NULL)
                { 
                    vers->options = xmalloc (strlen (rcsexpand) + 3);
                    strcpy (vers->options, "*"); 
                    strcat (vers->options, rcsexpand);
                }
        }

if (vers->options)
    vers->options = xstrdup ("*");

/* if tags were specified on the command line, they override what is in
 * the Entries file */
if (!tag || date)
    { 
        vers->tag = xstrdup (tag);
        vers->date = xstrdup (date);
    }
else if (vers->entdata & (sdtp & sdtp->aflag == 0))
    { 
        if (vers->tag)
            { 
                vers->tag = xstrdup (sdtp->tag);
                vers->nonbranch = sdtp->nonbranch;
            }
    }

if (vers->date)
    vers->date = xstrdup (sdtp->date);

/* Now look up the info on the source controlled file */
if (finfo->rcs != NULL)
    { 
        rcsdata = finfo->rcs;
        rcsdata->refcount++;
    }
else if (finfo->repository != NULL)
    rcsdata = RCSparse (finfo->file, finfo->repository);
else
    rcsdata = NULL;

if (rcsdata != NULL)
    { 
        /* squash away the rcsdata pointer for others */
        vers->arcfile = rcsdata;

        if (vers->tag & & strcmp (vers->tag, TAG_BASE) == 0)
            { 
                vers->vn_rcs = xstrdup (vers->vn_user);
                vers->vn_tag = xstrdup (vers->vn_user);
            }
        else
            { 
                int simple;
                char* localประเม = NULL;
/* We try remote first, because if the version is remote, and branched
at a local revision, we want to return remote, not local */
versions->vr_cs = RCS_getversion ((info, rcsdata, versions->tag, &local_rev));
if (versions->vr_cs == NULL) {
    /* Not remote, try regular (local) */
    if (local_rev != NULL) {
        versions->tag = local_rev;
    }
    versions->vr_cs = RCS_getversion (rcsdata, versions->tag,
        versions->date, force_simple_match,
        &simple);
    if (versions->vr_cs == NULL) {
        versions->vr_tag = NULL;
    } else if (simple)
        versions->vr_tag = xstrdup (versions->tag);
    else
        versions->vr_tag = xstrdup (versions->vr_cs);
} else {
    /* Found remote, forget about local */
    if (versions->vr_cs != NULL) {
        versions->vr_remote = xstrdup (versions->vr_cs);
    }
}

#if defined SERVER_SUPPORT
if (server_active)
    server_modtime (info, versions);
else
#endif

    struct utimbuf t;
    memset (&t, 0, sizeof (t));
    t.modtime = RCS_getmtime (rcsdata, versions->vr_cs, 0, 0);
    if (t.modtime != (time_t) -1)
        { t.actime = t.modtime;
          /* This used to need to ignore existence errors
             (for cases like where update.c now clears
              set_time if noexec, but didn’t use ts). I
             think maybe now it doesn’t (server_modtime does
             not like those kinds of cases). */
          (void) utime (info->file, &t);
        }
/* get user file time-stamp in t_user */
if (info->entries != (List *) NULL) {
    #ifdef SERVER_SUPPORT
    if (server_active)
        time_tstamp_server (info->file, versions, entdata);
    else
        #endif
        versions->t_user = tstamp (info->file);
    }
#endif
#endif

/* Set TS_USER to time stamp for FILE. */
/* Separate these out to keep the logic below clearer. */
#define mark_lost(V)     ((V)->t_user = 0)
#define mark_unchanged(V) ((V)->t_user = xstrdup ((V)->ts_cc))

static void
time_tstamp_server (file, versions, entdata)
    char *file;
    Versions *versions;
    Entnode *entdata;
{
    struct stat sb;
    char *cp;

if (CVS_lSTAT (file, &sb) < 0)
{
  if (! exists || error (errno))
    error (1, errno, "cannot stat temp file");
/* Missing file means lost or unmodified; check entries
* file to see which.

XXX FIXME - If there's no entries file line, we
* wouldn't be getting the file at all, so consider it
* lost. I don't know that that's right, but it's not
clear to me that either choice is. Besides, would we
* have an RCS string in that case anyways? */
  if (entdata == NULL)
    mark_lost (vers_ts);
  else if (entdata->timestamp
    && entdata->timestamp[0] == 's')
    mark_unchanged (vers_ts);
  else if (entdata->timestamp != NULL
    && (entdata->timestamp[0] == 'm'
      || entdata->timestamp[0] == 'M')
    && entdata->timestamp[1] == '0')
    vers_ts->ts_user = xstrdup ("is-modified");
  else
    mark_lost (vers_ts);
}
else if (sb.st_mtime == 0)
{
  /* We shouldn't reach this case any more! */
  abort ();
}
else
{
  struct tm *tm_p;
  struct tm local_tm;

  vers_ts->ts_user = xmalloc (25);
  /* We want to use the same timestamp format as is stored in the
   * st_mtime. For Unix (and NT I think) this *must* be universal
   * time (UT), so that files don't appear to be modified merely
   * because the timezone has changed. For VMS, or hopefully other
   * systems where gmtime returns NULL, the modification time is
   * stored in local time, and therefore it is not possible to cause
   * st_mtime to be out of sync by changing the timezone. */
  tm_p = gmtime (&sb.st_mtime);

  if (tm_p)
    {
      memcpy (&local_tm, tm_p, sizeof (local_tm));
      cp = asctime (&local_tm); /* copy in the modify time */
    }
  else
    {
      cp = ctime (&sb.st_mtime);
      
      cp[24] = 0;
      (void) strcpy (vers_ts->ts_user, cp);
    }
}
#endif /* SERVERSUPPORT */

/* Gets the time-stamp for the file "file" and returns it in space it
* allocates */

char * time_stamp (file)
char *file;
{
  struct stat sb;
  char *cp;
  char *ts;

  if (CVS_lSTAT (file, &sb) < 0)
    {
      ts = NULL;
    }
  else
    {
      struct tm *tm_p;
      struct tm local_tm;
      ts = xmalloc (25);

      /* We want to use the same timestamp format as is stored in the
       * st_mtime. For Unix (and NT I think) this *must* be universal
       * time (UT), so that files don't appear to be modified merely
       * because the timezone has changed. For VMS, or hopefully other
       * systems where gmtime returns NULL, the modification time is
       * stored in local time, and therefore it is not possible to cause
       * st_mtime to be out of sync by changing the timezone. */
      tm_p = gmtime (&sb.st_mtime);
      if (tm_p)
{  memcpy (&local_tm, tm_p, sizeof (local_tm));
   cp = asctime (&local_tm); /* copy in the modify time */
 }  
 else
   cp = ctime (&sb.st_mtime);
   cp[24] = 0;
   (void) strcpy (ts, cp);
 }

return (ts);

/* free up a Vers_TS struct */
void
freevers_ts (versp)
Vers_TS **versp;
{
  if ((versp)->srcfile)
    freercsnode (&((versp)->srcfile));
  if ((versp)->vn_user)
    free ((versp)->vn_user);
  if ((versp)->vn_rcs)
    free ((versp)->vn_rcs);
  if ((versp)->vn_tag)
    free ((versp)->vn_tag);
  if ((versp)->ts_user)
    free ((versp)->ts_user);
  if ((versp)->ts_rcs)
    free ((versp)->ts_rcs);
  if ((versp)->options)
    free ((versp)->options);
  if ((versp)->tag)
    free ((versp)->tag);
  if ((versp)->date)
    free ((versp)->date);
  if ((versp)->tag_conflict)
    free ((versp)->tag_conflict);
  if ((char *) versp)
    *versp = (Vers_TS *) NULL;
}
A.64 version.c

#include "cvs.h"

/* NOTE: remember to remove 'Halibut' when patching this code. */
char *version_string = "Concurrent Versions System (CVS) 1.10 'Halibut'";

#define CLIENT_SUPPORT
#define SERVER_SUPPORT

char *config_string = "(client/server)\n";

#define SERVER_SUPPORT
char *config_string = "(server)\n";

#define CLIENT_SUPPORT
char *config_string = "(client)\n";

#endif
#endif
#endif
#endif
A.65 watch.c

/* Implementation for "cvs watch", "cvs watchers", and related commands */

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

#include "cvs.h"
#include "edit.h"
#include "fileattr.h"
#include "watch.h"

const char *const watch_usage[] =
{
    "Usage: %s %s [on|off|add|remove] [-lR] [-a action] [files...]
    on/off: turn on/off read-only checks of files
    add/remove: add or remove notification on actions
    -l (on/off/add/remove): Local directory only, not recursive
    -R (on/off/add/remove): Process directories recursively
    -a (add/remove): Specify what actions, one of
    edit, unedit, commit, all, none
    (Specify the --help global option for a list of other help options)
    NULL
};

static struct addremove args the_args;

void watch modify watchers (file, what)
char *file,
struct addremove_args *what;
{
    char *curattr = fileattr_get0 (file, "_watchers");
    char *p;
    char *pend;
    char *nextp;
    char *who;
    int who_len;
    char *mycurattr;
    char *mynewattr;
    size_t mynewattr_size;
    int add_edit_pending;
    int add_unedit_pending;
    int add_commit_pending;
    int remove_edit_pending;
    int remove_unedit_pending;
    int remove_commit_pending;
    int add_tedit_pending;
    int add_tunedit_pending;
    int add_tcommit_pending;
    who = getcaller ();
    who_len = strlen (who);
    /* Look for current watcher types for this user. */
    mycurattr = NULL;
    if (curattr != NULL) {
        p = curattr;
        while (1) {
            if (strncmp (who, p, who_len) == 0
                && p[who_len] == '>') {
                /* Found this user. */
                mycurattr = p + who_len + 1;
            } else {
                p = strchr (p, ',');
                if (p == NULL)
                    break;
                p++;
            }
        } if (mycurattr != NULL) {
            mycurattr = xstrdup (mycurattr);
            p = strchr (mycurattr, ',');
            if (p != NULL)
                *p = '0';
        } /* Now copy mycurattr to mynewattr, making the requisite modifications.
Note that we add a dummy '+' to the start of mynewattr, to reduce... */
special cases (but then we strip it off when we are done). */

mynewattr_size = sizeof "edit=unedit=commit=tedit=tunedit=tcommit";
if (mycurattr != NULL)
  mynewattr = strlen (mycurattr);
mynewattr = xmalloc (mynewattr_size);
mynewattr[0] = '\0';

add_edit_pending = what->adding && what->edit;
add_unedit_pending = what->adding && what->unedit;
add_commit_pending = what->adding && what->commit;
remove_edit_pending = !what->adding && what->edit;
remove_unedit_pending = !what->adding && what->unedit;
remove_commit_pending = !what->adding && what->commit;
add_tedit_pending = what->add_tedit;
add_tunedit_pending = what->add_tunedit;
add_tcommit_pending = what->add_tcommit;

/* Copy over existing watch types, except those to be removed. */

while (p != NULL)
  {
    pend = strchr (p, ' ');  
    if (pend == NULL)
      {
        pend = p + strlen (p);
        nextp = NULL;
      }
    else
      nextp = pend + 1;

    /* Process this item. */
    if (pend - p == 4 && strncmp (*edit*, p, 4) == 0)
      {
        if (isremove) pending = 0;
        if (isedit)
          {
            strcat (mynewattr, "+edit");
            add_edit_pending = 0;
          }
        else if (pend - p == 6 && strncmp (*unedit*, p, 6) == 0)
          {
            if (isremove)
              {
                strcat (mynewattr, "-unedit");
                add_unedit_pending = 0;
              }
            else if (pend - p == 6 && strncmp (*commit*, p, 6) == 0)
              {
                if (isremove)
                  {
                    strcat (mynewattr, "-commit");
                    add_commit_pending = 0;
                  }
                else if (pend - p == 5 && strncmp (*tedit*, p, 5) == 0)
                  {
                    if (isremove)
                      {
                        strcat (mynewattr, "-tedit");
                        add_tedit_pending = 0;
                      }
                    else if (pend - p == 7 && strncmp (*tunedit*, p, 7) == 0)
                      {
                        if (isremove)
                          {
                            strcat (mynewattr, "-tunedit");
                            add_tunedit_pending = 0;
                          }
                        else if (pend - p == 7 && strncmp (*tcommit*, p, 7) == 0)
                          {
                            if (isremove)
                              {
                                strcat (mynewattr, "-tcommit");
                                add_tcommit_pending = 0;
                              }
                          }
                      }
                    else
                      {
                        char *mp;

                        /* Copy over any unrecognized watch types, for future
                         expansion. */
                        mp = mynewattr + strlen (mynewattr);
                        *mp++ = '"';
                        strcat (mp, p, pend - p);
                        *(mp + (pend - p)) = '"';
                      }
                  }
                else
                {
                  char *mp;

                  /* Set up for next item. */
                  p = nextp;
                }
          }
      }
  }
statt (mynewattr, **commit**);

if (add_tedit_pending),

statt (mynewattr, **tedit**);
if (add_tunedit_pending)

statt (mynewattr, **tunedit**);
if (add_commit_pending)

statt (mynewattr, **commit**);

{
    char *curattrnew =

    fileattr_modify (curattr,
who, mynewattr[0] == \"\"? NULL : mynewattr + 1,
> \");

    /*! If the attribute is unchanged, don't rewrite the attribute file. */
    if (!(curattrnew == NULL && curattr == NULL))
        (curattrnew != NULL
      && strlen (curattrnew, curattr) == 0))
    fileattr_set (file, 
"_watchers",
        curattrnew);

    if (curattr_new != NULL)
    free (curattr_new);
}
if (curattr != NULL)
    free (curattr);
if (mycurattr != NULL)
    free (mycurattr);
if (mynewattr != NULL)
    free (mynewattr);

static int addremovelfileproc PROTO ((void *callerdat,
 struct file_info *finfo));

static int
addremovelfileproc (callerdat, finfo)
void *callerdat;
struct file_info *finfo;
{
    watch_modify_watchers (finfo->file, &the_args);
    return 0;
}

static int addremovelfiledoneproc PROTO ((void *, int, char *, char *,
 List *));

static int
addremovelfiledoneproc (callerdat, err, repository, update_count, entries)
void * callerdat;
int err;
char * repository;
char *update_count;
List *entries;
{
    if (the_args.setting == default)
    watch_modify_watchers (NULL, &the_args);
    return err;
}

static int watch_addremové (int argc, char **argv);

static int
watch_addremové (argc, argv)
int argc;
char **argv;

{
    int c;
    int local = 0;
    int err;
    int a_omitted;
    a_omitted = 1;
    the_args.commit = 0;
    the_args.edit = 0;
    the_args.unedit = 0;

    optind = 0;
    while ((c = getopt (argc, argv, \"*:\")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'a':
                break;
            case 'i':
                // Do something with '-i'
                break;
            default:
                break;
        }
    }
local = 0;
break;
case 'a':
    a omitted = 0;
    if (strcmp (optarg, "edit") == 0)
        the_args.edit = 1;
    else if (strcmp (optarg, "unedit") == 0)
        the_args.unedit = 1;
    else if (strcmp (optarg, "commit") == 0)
        the_args.commit = 1;
    else if (strcmp (optarg, "all") == 0)
        {
            the_args.edit = 1;
            the_args.unedit = 1;
            the_args.commit = 1;
        }
    else if (strcmp (optarg, "none") == 0)
        {
            the_args.edit = 0;
            the_args.unedit = 0;
            the_args.commit = 0;
        }
    else
        usage (watch_usage);
    break;
case '?':
default:
    usage (watch_usage);
    break;
}
 argc -= optind;
 argv += optind;

if (a omitted)
    {
        the_args.edit = 1;
        the_args.unedit = 1;
        the_args.commit = 1;
    }

#ifdef CLIENT_SUPPORT
if (client_active)
    {
        start_server ();
        ign_setup ();
        if (local)
            send_arg ("-1");
            /* FIXME: copes poorly with "all" if server is extended to have
            new watch types and client is still running an old version. */
        if (the_args.edit)
            {
                send_arg ("-a");
                send_arg ("edit");
            }
        if (the_args.unedit)
            {
                send_arg ("-a");
                send_arg ("unedit");
            }
        if (the_args.commit)
            {
                send_arg ("-a");
                send_arg ("commit");
            }
        if (the_args.edit && the_args.unedit && the_args.commit)
            {
                send_arg ("-a");
                send_arg ("none");
            }
        sendfiles (argv, SEND_EXPAND_WILD);
        sendfiles (argv, SEND_NO_CONTENTS);
        send_to_server (the_args.adding ?
            "watch-add\012" : "watch-remove\012",
            0);
        return gen_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

sendif /* CLIENT_SUPPORT */

the_args.setting_default = (argc <= 0);

lock_tree_for_write (argv, argv, local, 0);

err = start_recursion (addremove_fileproc, addremove_thesendoneproc,
                (DIREFIXPROC) NULL, (DIREFIXPROC) NULL, NULL,
                argv, argv, local, LOCAL, 0, 0, (char *)NULL,
                1);

Lock Cleanup ();
return err;
}

int
watch_add (argc, argv)
int argc;
char *argv;
{
    the_args.adding = 1;
    return watch_addremove (argc, argv);
}

int
watch_remove (argc, argv)
int argc;
char *argv;
{
    the_args.adding = 0;
    return watch_addremove (argc, argv);
}

int
watch (argc, argv)
int argc;
char *argv;
{
    if (argc <= 1)
        usage [watch_usage];
    if (strcmp (argv[1], "on") == 0)
    {
        --argc;
        ++argv;
        return watch_on (argc, argv);
    } else if (strcmp (argv[1], "off") == 0)
    {
        --argc;
        ++argv;
        return watch_off (argc, argv);
    } else if (strcmp (argv[1], "add") == 0)
    {
        --argc;
        ++argv;
        return watch_add (argc, argv);
    } else if (strcmp (argv[1], "remove") == 0)
    {
        --argc;
        ++argv;
        return watch_remove (argc, argv);
    } else
        usage [watch_usage];
    return 0;
}

static const char *const watchers_usage[] =
{
    "Usage: %s %s [-lR] [files...]
     	-l	Process this directory only (not recursive).
     	-R	Process directories recursively.
     	(Specify the --help global option for a list of other help options)
     	NULL
};

static int watchers_fileproc PROTO ((void *callerdat,
                                    struct file_info *finfo));

static int
watchers_fileproc (callerdat, finfo)

void *callerdat;
struct file_info *finfo;
{
    char *them, *p;
    them = fileattrget0 (finfo->file, "_watchers");
    if (them == NULL)
        return 0;

    puts (finfo->fullname, stdout);
    p = them;
    while (1)
    {
        ptc ("\t", stdout);
        while (p != \n & & ++p != \n  )
            ptc (*p++, stdout);
        if (p == \n
  )
450     {  /* Only happens if attribute is misformed. */
        putc ("\n", stdout);
        break;
    }  
    ++p;
    putc ("\t", stdout);
    while (1)  
    {  
        while ( (sp = ' ' & & sp != ' ' & & sp != '\0')
        {  
            putc (sp++, stdout);
        }  
        if (sp == '\0')  
        {  
            putc ("\n", stdout);
            goto out;
        }  
        if (sp == ',')  
        {  
            ++p;
            break;
        }  
        ++p;
        putc ("\t", stdout);
    }  
    putc ("\n", stdout);
    }  
    out:;  
    return 0;
}

480 int watchers (argc, argv)
    
    int argc,
    char **argv;
    
    {  
        int local = 0;
        int c;
        
        if (argc == -1)  
            usage (watchers_usage);
        
        optind = 0;
        while ((c = getopt (argc, argv, "+lR")) != -1)  
        {  
            switch (c)  
            {  
                case 'l':  
                    local = 1;
                    break;
                case 'R':  
                    local = 0;
                    break;
                case '?':  
                    default:  
                    usage (watchers_usage);
                    break;
            }  
        }  
        argc -= optind;
        argv += optind;
    }

500 #ifdef CLIENT_SUPPORT  
    if (client_active)  
    {  
        start_server ();  
        ignore_setup ();  
        
        if (local)  
            send_arg ("-1");  
            send_names (argc, argv, SEND_EXPAND_WILD);
            send_files (argc, argv, local, SEND_NO_CONTENTS);
            send_server ("watchers\012", 0);
            return get_responses (c, ch);  
        }  
    }  
    #endif  
    /∗ CLIENT_SUPPORT ∗/  
    return start_recursion (watchers_fileproc, (FILESDONEPROC) NULL,  
        (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,  
        argc, argv, local, WLOCAL, 0, 1, (char *)NULL,  
        1);  
    }
A.66  watch.h

/* Interface to "cvs watch add", "cvs watchers", and related features */

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. */

extern const char *const watch_usage[];

/* Flags to pass between the various functions making up the add/remove code. All in a single structure in case there is some need to make the code reentrant some day. */

struct addremove_args {
  /* A flag for each watcher type. */
  int edit;
  int unedit;
  int commit;
  /* Are we adding or removing (non-temporary) edit,unedit,and/or commit watches? */
  int adding;
  /* Should we add a temporary edit watch? */
  int add_tedit;
  /* Should we add a temporary unedit watch? */
  int add_tunedit;
  /* Should we add a temporary commit watch? */
  int add_tcommit;
  /* Should we remove all temporary watches? */
  int remove_temp;
  /* Should we set the default? This is here for passing among various routines in watch.c (a good place for it if there is ever any reason to make the stuff reentrant), not for watch_modify watchers. */
  int setting_default;
};

/* Modify the watchers for FILE. WHAT tells what to do to them. If FILE is NULL, modify default args (WHAT->SETTING_DEFAULT is not used). */

extern void watch_modify_watchers_PROTO ((char *file, struct addremove_args *what));

extern int watch_add_PROTO ((int argc, char **argv));
extern int watch_remove_PROTO ((int argc, char **argv));
wrapper.c

/* This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
y any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details. */

#include "cvs.h"
#include "getline.h"

Original Author: athanmorgan <Andrew C. Athan> 2/1/94
Modified By: vdemarco@shl.com

This package was written to support the NEXTSTEP concept of
"wrappers." These are essentially directories that are to be
treated as "files." This package allows such wrappers to be
"processed" on the way in and out of CVS. The intended use is to
wrap up a wrapper into a single tar, such that that tar can be
treated as a single binary file in CVS. To solve the problem
effectively, it was also necessary to be able to prevent rcsmerge
application at appropriate times.

Format of wrapper file (SCVSROOT/CVSROOT/cswrappers or .cvswrappers):

wildcard [option value][option value]...

where option is one of
-f from cvs filter value: path to filter
-t to cvs filter value: path to filter
-m update methodology value: MERGE or COPY
-k default -k rcs option to use on import or add

and value is a single-quote delimited value.

E.g:
+.nib -f 'gunzipuntar' -t 'targzip' -m 'COPY'

*/

typedef struct {
char *wildCard;
char *tocvsFilter;
char *fromcvsFilter;
char *rcsOption;
WrapMergeMethod mergeMethod;
} WrapperEntry;

static WrapperEntry **wrap_list=NULL;
static WrapperEntry **wrap_saved_list=NULL;
static int wrap_count=0;
static int wrap_tempcount=0;

#define WRAPPER_GROW 8

void wrap_add_entry PROTO((WrapperEntry **e,int temp));
void wrap_kill PROTO((void));
void wrap_kill_temp PROTO((void));
void wrap_free_entry PROTO((WrapperEntry **e));
void wrap_free_entry_internal PROTO((WrapperEntry **e));
void wrap_restore_waved PROTO((void));

void wrap_setup()
{
    /* FIXME: reentrancy: if we do a multithreaded server, will need to
    move this to a per-connection data structure, or better yet
    */
}
think about a cleaner solution. */
static int wrap_setup_already_done = 0;
char *homedir;
if (wrap_setup_already_done != 0)
    return;
else
    wrap_setup_already_done = 1;

#define CLIENT SUPPORT
if (client_active)
#endif
send
{
    char *file;
    file = xmalloc(strlen(CVSroot_directory)
                   + sizeof(CVSROOTADM) + 10);
    /* Then add entries found in repository, if it exists. */
    (void) sprintf(file, "%s/%s", CVSroot_directory, CVSROOTADM,
                   CVSDOTWRAPPER);
    if (isfile (file))
        { 
            wrap_add_file(file,0);
        }
    free (file);
}
/* / Then add entries found in home dir, (if user has one) and file
exists. */
home_dir = get_home_dir();
if (home_dir != NULL)
    {
        char *file;
        file = xmalloc(strlen(home_dir) + sizeof(CVSDOTWRAPPER) + 10);
        (void) sprintf(file, "%s/%s", home_dir, CVSDOTWRAPPER);
        if (isfile (file))
            { 
                wrap_add_file(file,0);
            }
        free (file);
    }
/* * FIXME: calling wrap_add() below implies that the CVSWRAPPER
* environment variable contains exactly one "wrapper" - a line
* of the form
* FILENAME_PATERN FLAG OPTS [ FLAG OPTS ] ...
*/

This may disagree with the documentation, which states:
* * *CVSWRAPPER*
* A whitespace-separated list of file name patterns that CVS
* should treat as wrappers. *Note Wrappers::*
* * Does this mean the environment variable can hold multiple
* wrappers lines? If so, a single call to wrap_add() is
* insufficient.
* */
/* * Then add entries found in CVSWRAPPER environment variable. */
wrap_add (getenv (WRAPPER_ENV), 0);
}
#endif

/* * Send -W arguments to the wrappers to the server. The command must
* be one that accepts them (e.g. update, import). */
void
wrap_send ()
{
    int i;
    for (i = 0; i < wrap_count + wrap_tempcount; ++i)
    {
        if (wrap_list[i]->tovcsFilter != NULL
            || wrap_list[i]->fromcvsFilter != NULL)
            /* For greater studliness we would print the offending option
               and (more importantly) where we found it. */
            error (0, 0, "t and -f wrapper options are not supported remotely; ignored");
        if (wrap_list[i]->mergeMethod == WRAP_COPY)
            /* For greater studliness we would print the offending option
               and (more importantly) where we found it. */
            error (0, 0, "m wrapper option is not supported remotely; ignored");
        if (wrap_list[i]->rcsOption != NULL)
            { 
                send_to_server ("Argument -W\012Argument ", 0);
            }
sendlist(server (wraplist[i] ->wildCard, 0));
sendlist(server (*k**, 0);
sendlist(server (wraplist[i] ->rcsOption, 0);)
sendlist(server (**"\012", 0);)
}
}
sendit /* CLIENT_SUPPORT */

#if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)
/* Output wrapper entries in the format of cvswrappers lines.
 * This is useful when one side of a client/server connection wants to
 * send its wrappers to the other; since the receiving side would like
 * to use wrapadd() to incorporate the wrapper, it's best if the
 * entry arrives in this format.
 * The entries are stored in 'line', which is allocated here. Caller
 * can free() it. */
/* If first_callp is nonzero, then start afresh. */

200 void
wrap_unparse_rcs_options (line, first_callp)
char **line;
int first_callp;
{ /* FIXME-reentrancy: we should design a reentrant interface, like
     a callback which gets handed each wrapper (a multithreaded
     server being the most concrete reason for this, but the
     non-reentrant interface is fairly unnecessary/ugly). */
static int i;

210 if (first_callp)
  i = 0;
for (; i < wrap_count + wrap_tempcount; ++i)
{ if (wraplist[i] ->rcsOption != NULL)
  { *line = xmalloc (strlen (wraplist[i] ->wildCard)
    + strlen (**"\012")
    + strlen (**"- k' ")
    + strlen (wraplist[i] ->rcsOption)
    + strlen (**")
    + 1); /* leave room for \012 * /
     strcpy (*line, wraplist[i] ->wildCard);
     strcat (*line, **"- k' ");
     strcat (*line, wraplist[i] ->rcsOption);
     strcat (*line, **");

230 /* We're going to miss the increment because we return, so
     do it by hand. */
  ++i;
  return;
  }
  *line = NULL;
  return;
}
}
#endif /* SERVER_SUPPORT || CLIENT_SUPPORT */

250 void
wrap_add_file (file, temp)
const char *file;
int temp;
{ FILE *fp;
  char *line = NULL;
  size_t line_allocated = 0;
  wrap_restore_saved ();
  wrap_kill_temp ();

260 /* Load the file. */
  fp = CVS_FOPEN (file, *r*);
  if (fp == NULL)
    { if (!existence_error (errno))
        error (0, errno, *cannot open \"%, file\*);
        return;
    }
    while (getline (&line, &line_allocated, fp) >= 0)
void wrap_add(char *line, int isTemp)
{
    char *temp;
    char ctemp;
    WrapperEntry e;
    char opt;
    if (!line || line[0] == '#') return;
    memset(&e, 0, sizeof(WrapperEntry));

    /* Search for the wild card */
    while(isspace(*line)) ++line;
    for(temp=line; *line && isspace(*line); ++line)
    {
        if(isTemp) ctemp = *line;
        *line = *temp;
        if (isspace(*line)) ctemp = *line;
    }
}

void wrap_kill()
{
    WrapperEntry *temp;
    while(wrap_count)
    {
        if (isspace(*wrap_list[--wrap_count]))
            continue;
        if (isspace(*wrap_tempcount))
            continue;
        *temp = *wrap_tempcount = wrap_free_entry(wrap_list);
    }
}

void wrap_free_entry(WrapperEntry *e)
{
    free(e);
}

void wrap_free_entry_internal(WrapperEntry *e)
{
    if (e->wildCard) free(e->wildCard);
    if (e->fromcvsFilter) free(e->fromcvsFilter);
    if (e->fromrcsOption) free(e->fromrcsOption);
    if (e->tocvsFilter) free(e->tocvsFilter);
    if (e->tocrcsOption) free(e->tocrcsOption);
}

void wrap_restore_saved()
{
    if(wrap_saved_list) return;
    wrap_kill();
    free(wrap_list);
    wrap_list = wrap_saved_list;
    wrap_count = wrap_saved_count;
    wrap_tempcount = wrap_saved_tempcount;
    wrap_saved_list = NULL;
    wrap_saved_count = 0;
    wrap_saved_tempcount = 0;
}

void wrap_add(char *line, int isTemp)
char *line;
int isTemp;
{
    char *temp;
    char ctemp;
    WrapperEntry e;
    char opt;
    if (!line || line[0] == '#') return;
    memset(&e, 0, sizeof(WrapperEntry));

    /* Search for the wild card */
    while(isspace(*line)) ++line;
    for(temp=line; *line && isspace(*line); ++line)
    {
        if(isTemp) ctemp = *line;
        *line = *temp;
        if (isspace(*line)) ctemp = *line;
    }
    ctemp = *line;
wrapper.c – 757

*e.wildCard=xstrdup(temp);
+line=ctemp;

while(*line){
  /* Search for the option */
  while(*line && *line!='-')
    ++line;
  if(*line)
    break;
  ++line;
  if(*line)
    break;
  opt=line;

  /* Search for the filter commandline */
  for(++line,*line=*line=='\';++line);
  if(*line)
    break;

  for(temp=++line,*line=*line=='\' || *line[-1]=='\';++line)
    ;

  /* This used to "break;" (ignore the option) if there was a
   * single character between the single quotes (I'm guessing
   * that was accidental). Now it "break;"s if there are no
   * characters. I'm not sure either behavior is particularly
   * necessary-the current options might not require "
   * arguments, but surely some future option legitimately
   * might. Also I'm not sure that ignoring the option is a
   * swift way to handle syntax errors in general. */
  if(line==temp)
    break;

  ctemp=*line;
  *line='\0';
  switch(opt){
    case 'f':
      /* Before this is reenabled, need to address the problem in
       * commit.c (see http://www.cyclic.com/cvs/dev-wrap.txt). */
      error (1, 0,
        "-t/-f wrappers not supported by this version of CVS");
      if(e.fromcvsFilter)
        free(e.fromcvsFilter);
      /* FIXME: error message should say where the bad value
       * came from. */
      e.fromcvsFilter=xexpand_path (temp, "<wrapper>", 0);
      if (e.fromcvsFilter)
        error (1, 0, "Correct above errors first");
      break;
    case 't':
      /* Before this is reenabled, need to address the problem in
       * commit.c (see http://www.cyclic.com/cvs/dev-wrap.txt). */
      error (1, 0,
        "-t/-f wrappers not supported by this version of CVS");
      if(e.tocvsFilter)
        free(e.tocvsFilter);
      /* FIXME: error message should say where the bad value
       * came from. */
      e.tocvsFilter=xexpand_path (temp, "<wrapper>", 0);
      if (e.tocvsFilter)
        error (1, 0, "Correct above errors first");
      break;
    case 'm':
      /* FIXME: look into whether this option is still relevant given
       * the 24 Jun 96 change to merge/tc. */
      if(*temp=='-' || *temp=='c')
        e.mergeMethod=WRAP_COPY;
    else
      e.mergeMethod=WRAP_MERGE;
    break;
    case 'k':
      if (e.rcsOption)
        free (e.rcsOption);
      e.rcsOption = xstrdup (temp);
      default:
        break;
  }
  *line=ctemp;
  if(*line)break;
  ++line;
}
wrap_add_entry(&e, isTemp);
void wrap_add_entry(char *, temp)

{ int temp;

  WrapperEntry *e;

  if (temp && wrap->tempcount >= wrap->size)
  { wrap->size += WRAPPER_SIZE;
    wrap->list = (WrapperEntry **) xrealloc((char *) wrap->list,
               wrap->size *
               sizeof (WrapperEntry *));
  }

  if(temp & wrap->tempcount)
  { for(x=wrap->count+wrap->tempcount-1;x>=wrap->count;--x)
    wrap_list[x+1]=wrap_list[x];
  }

  x=(temp ? wrap->count+wrap->tempcount+1: (wrap->count+1));

  wrap_list[x]=WrapMergeHas;
  wrap_list[x]->wildCard=wrap->wildCard;

  wrap_list[x]->fromcvsFilter=wrap->fromcvsFilter;
  wrap_list[x]->tocvsFilter=wrap->tocvsFilter;
  wrap_list[x]->mergeMethod=wrap->mergeMethod;

  return wrap_list[x]->rcsOption;
}

/* Return 1 if the given filename is a wrapper filename */

int wrap_name_has(const char *name, has)

{ int x, count=wrap->count+wrap->tempcount;

  const char *name;

  for(x=0;x<count;++x)
  if (CVS_FNMATCH (wrap_list[x]->wildCard, name, 0) == 0)
  { switch(has)
    case WRAP_TOCVS:
      temp=wrap_list[x]->tocvsFilter;
      break;
    case WRAP_FROMCVS:
      temp=wrap_list[x]->fromcvsFilter;
      break;
    case WRAP_RCSOPTION:
      temp=wrap_list[x]->rcsOption;
      break;
    default:
      abort (1);
  }

  if(temp==NULL)
    return (0);
  else
    return (1);

  return (0);
}

static WrapperEntry *wrap_matching_entry PROTO ((const char *));

static WrapperEntry *
wrap_matching_entry (name)

{ int x, count=wrap->count+wrap->tempcount;

  for(x=0;x<count;++x)
  if (CVS_FNMATCH (wrap_list[x]->wildCard, name, 0) == 0)
    return wrap_list[x];

  return (WrapperEntry *)NULL;

} /* Return the RCS options for FILENAME in a newly malloc’d string. If
   ASFLAG, then include "-k" at the beginning (e.g. "-kb"), otherwise
   just give the option itself (e.g. "k"). */

char *
wrap_rcsoption (filename, asflag)

{ const char *filename;

  int asflag;

  { WrapperEntry *e = wrap_matching_entry (filename);

    char *buf;

    if (e == NULL || e->rcsOption == NULL || (e->rcsOption == ^O))
      return NULL;

    buf = xmalloc(strlen (e->rcsOption) + 3);
    if (asflag)
    {
strcpy (buf, "-k");
}
else
{
  strcpy (buf, e->rcsOption);
}
return buf;
}

char *
wrap_tocvs_process_file(const char *fileName)
{
  WrapperEntry *e=wrap_matching_entry(fileName);
  static char *buf = NULL;
  char *args;
  if (e==NULL || e->tocvsFilter==NULL)
    return NULL;
  if (buf != NULL)
    free (buf);
  buf = cvs_temp_name();
  args = xmalloc (strlen (e->tocvsFilter)
    + strlen (fileName)
    + strlen (buf));
  /* FIXME: sprintf will blow up if the format string contains items other
   * than %s, or contains too many %s's. We should instead be parsing
   * e->tocvsFilter ourselves and giving a real error. */
  sprintf (args, e->tocvsFilter,
    fileName);
  run_setup (args);
  run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORM_AL|RUN_REAL); free (args);
  return buf;
}

int
wrap_merge_copy (fileName)
{
  WrapperEntry *e=wrap_matching_entry(fileName);
  if (e==NULL || e->mergeMethod==WRAP_MERGE)
    return 0;
  return 1;
}

void
wrap_fromcvs_process_file(const char *fileName)
{
  char *args;
  WrapperEntry *e=wrap_matching_entry(fileName);
  if (e==NULL || e->fromcvsFilter==NULL)
    return;
  args = xmalloc (strlen (e->fromcvsFilter)
    + strlen (fileName));
  /* FIXME: sprintf will blow up if the format string contains items other
   * than %s, or contains too many %s's. We should instead be parsing
   * e->fromcvsFilter ourselves and giving a real error. */
  sprintf (args, e->fromcvsFilter, fileName);
  run_setup (args);
  run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORM);
  free (args);
  return;
A.68  zlib.c

/* zlib.c — interface to the zlib compression library
Ian Lance Taylor <ian@lance.com>

This file is part of GNU CVS.

GNU CVS is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2, or (at your option) any
later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
Licence for more details.

The routines in this file are the interface between the CVS
client/server support and the zlib compression library. */

#include <assert.h>
#include "zlib.h"
#include "buffer.h"

#define SERVER_SUPPORT || defined (CLIENT_SUPPORT)
#include "zlib.h"

#if OS/2 doesn't have EIO. FIXME: this whole notion of turning
a different error into EIO strikes me as pretty dubious. */
#define defined (EIO)
#endif

#define EIO EBADPOS

/* The compression interface is built upon the buffer data structure.
We provide a buffer type which compresses or decompresses the data
which passes through it. An input buffer decompresses the data
read from an underlying buffer, and an output buffer compresses the
data before writing it to an underlying buffer. */

/* This structure is the closure field of the buffer. */

struct compress_buffer
{
    /* The underlying buffer. */
    struct buffer *buf;
    /* The compression information. */
    zstream *zstr;
};

static void compress_error PROTO((int, int, z_stream *, const char *));

static int compress_buffer_input PROTO((void *, char *, int, int, int *));
static int compress_buffer_output PROTO((void *, const char *, int, int *));
static int compress_buffer_flush PROTO((void *));
static int compress_buffer_block PROTO((void *, int));
static int compress_buffer_shutdown_input PROTO((void *));
static int compress_buffer_shutdown_output PROTO((void *));

/* Report an error from one of the zlib functions. */

static void
compress_error (status, zstatus, zstr, msg)
{
    int status;
    int zstatus;
    z_stream *zstr;
    const char *msg;

    int hold_errno;
    const char *zmsg;
    char buf[100];

    hold_errno = errno;
    zmsg = zstr->msg;
    if (zmsg == NULL)
    {
        sprintf (buf, "%s: %s", zstatus, hold_errno ? hold_errno : 0,
                "\x", 3, msg, zmsg);
    } else
    { /* Create a compression buffer. */
        struct buffer *
            compress_buffer_initialize (buf, input, level, memory);
        struct buffer *buf;
        }
int input;
int level;

void (*memory) PROTO([struct buffer *]);

{ struct compress_buffer *n;
  int zstatus;

  n = (struct compress_buffer *) xmalloc (sizeof *n);
  memset (n, 0, sizeof *n);

  n->buf = buf;

  if [input]
    zstatus = inflateInit (&n->zstr);
  else
    zstatus = deflateInit (&n->zstr, level);

  if [zstatus != Z_OK]
    compress_error (1, zstatus, &n->zstr, "compression initialization");

  /* There may already be data buffered on BUF. For an output
   * buffer, this is OK, because these routines will just use the
   * buffer routines to append data to the (uncompressed) data
   * already on BUF. An input buffer expects to handle a single
   * buffer of data of buffered input to be uncompressed, so that is OK
   * provided there is only one buffer. At present that is all
   * there ever will be; if this changes, compress_buffer_input
   * must be modified to handle multiple input buffers. */
  assert (!input || buf->data == NULL || buf->data->next == NULL);

  return buf->initialize (input ? compress_buffer_input : NULL,
    input ? NULL : compress_buffer_output,
    input ? NULL : compress_buffer_flush,
    compress_buffer_block,
    (input
    ? compress_buffer_shutdown_input :
    compress_buffer_shutdown_output),
    memory,
    n);
}

/* Input data from a compression buffer. */

static int
compress_buffer_input (cb, data, need, size, got)
  void *cb;
  char *data;
  int need;
  int size;
  int *got;

  { struct compress_buffer *cb = (struct compress_buffer *) cb;

    struct buffer_data *bd = NULL;
    if (cb->buf->input == NULL)
      abort ();

    /* We use a single buffer structure to buffer up data which
     * the stream structure won't use yet. We can safely store
     * this on the buffer routine, because we never call the buffer routines on
     * the buffer input routine, since that
     * gives us the semantics we want. As noted in
     * compress_buffer_input, the buffer_data structure may
     * already exist, and hold data which was already read and
     * buffered before the decompression began. */
    bd = cb->buf->data;
    if (bd == NULL)
      { bd = (struct buffer_data *) malloc (BUFFER_DATA_SIZE);
        if (bd == NULL)
          return -2;
        bd->text = (char *) malloc (BUFFER_DATA_SIZE);
        if (bd->text == NULL)
          { free (bd);
            return -2;
          }
        bd->bufp = bd->text;
        bd->size = 0;
        cb->buf->data = bd;
      }

    cb->zstr.avail_out = size;
    cb->zstr.next_out = (Bytef *) bd;

    while (1)
      { int zstatus, sofar, status, nread;

        /* First try to inflate any data we already have buffered up.
         * This is useful even if we don't have any buffered data, 
         */
because there may be data buffered inside the stream structure.

```c
    cb->zstr.avail_in = bd->size;
    cb->zstr.next_in = (Byte *) bd->bufp;
    do
    {
        zstatus = inflate (&cb->zstr, Z_NO_FLUSH);
        if (zstatus == Z_STREAM_END)
            break;
        if (zstatus != Z_OK && zstatus != Z_BUF_ERROR)
            { compress_error (0, zstatus, &cb->zstr, "inflate");
              return EIO;
            }
    } while (cb->zstr.avail_in > 0
    && cb->zstr.avail_out > 0);
    bd->size = cb->zstr.avail_in;
    bd->bufp = (char *) cb->zstr.next_in;

if (zstatus == Z_STREAM_END)
    return -1;

/* If we have obtained NEED bytes, then return, unless NEED is
zero and we haven’t obtained anything at all. If NEED is
zero, we will keep reading from the underlying buffer until
we either can’t read anything, or we have managed to
inflate at least one byte. */
solar = size - cb->zstr.avail_out;

if (solar > 0 && solar >= need)
    break;

/* All our buffered data should have been processed at this
point. */
assert (bd->size = 0);

/* This will work well in the server, because this call will
do an unblocked read and fetch all the available data. In
the client, this will read a single byte from thestdio
stream, which will cause us to call inflate once per byte.
It would be more efficient if we could make a call which
would fetch all the available bytes, and at least one byte. */
status = (cb->buf->input) (cb->buf->closure, bd->text, 
need > 0 ? 1 : 0, BUFFER_DATA_SIZE, &nread);
if (status != 0)
    return status;

/* If we didn’t read anything, then presumably the buffer is
in nonblocking mode, and we should just get out now with
whatever we’ve inflated. */
if (nread == 0)
    { assert (need == 0);
      break;
    }
    bd->bufp = bd->text;
    bd->size = nread;
    }
  *got = size - cb->zstr.avail_out;
  return 0;
}

/* Output data to a compression buffer. */

static int
compress_buffer_output (closure, data, have, wrote)
  void *closure;
  const char *data;
  int have;
  int *wrote;
{
  struct compress_buffer *cb = (struct compress_buffer *) closure;
  cb->zstr.avail_in = have;
  cb->zstr.next_in = (unsigned char *) data;
  while (cb->zstr.avail_in > 0)
    { char buffer[BUFFER_DATA_SIZE];
      int zstatus;
      cb->zstr.avail_out = BUFFER_DATA_SIZE;
      cb->zstr.next_out = (unsigned char *) buffer;
```
zstatus = deflate (&cb->zstr, Z_NO_FLUSH);
if (zstatus != Z_OK)
{
    compress_error (0, zstatus, &cb->zstr, "deflate");
    return EIO;
}
if ((cb->zstr.availout != BUFFER_DATA_SIZE)
    && (cb->zstr.block))
{
    Buf output (cb->buf, buffer, BUFFER_DATA_SIZE = cb->zstr.availout);
}

/* We will only be here because buf->endoutput was called on the
   compression buffer. That means that we should now call
   buf->endoutput on the underlying buffer. */
return buf->endoutput (cb->buf);

/* Flush a compression buffer. */

static int
compress_buffer_flush (closure)
    void *closure;
{
    struct compress_buffer *cb = (struct compress_buffer *) closure;
    cb->zstr.availin = 0;
    cb->zstr.nextin = NULL;

    while (1)
    {
        char buffer[BUF_SIZE];
        int zstatus;
        cb->zstr.availout = BUFFER_DATA_SIZE;
        cb->zstr.nextout = (unsigned char *) buffer;
        zstatus = deflate (&cb->zstr, Z_SYNC_FLUSH);

        /* The deflate function will return Z_BUF_ERROR if it can't do
           anything, which in this case means that all data has been
           flushed. */
        if (zstatus == Z_BUF_ERROR)
            break;
        if (zstatus != Z_OK)
        {
            compress_error (0, zstatus, &cb->zstr, "deflate flush");
            return EIO;
        }
        if ((cb->zstr.availout != BUFFER_DATA_SIZE)
            && (cb->zstr.block))
        {
            Buf output (cb->buf, buffer, BUFFER_DATA_SIZE = cb->zstr.availout);
        }
        /* If the deflate function did not fill the output buffer,
           then all data has been flushed. */
        if (cb->zstr.availout > 0)
            break;
    }
    /* Now flush the underlying buffer. Note that if the original
       call to buf->flush passed 1 for the BLOCK argument, then the
       buffer will already have been set into blocking mode, so we
       should always pass 0 here. */
    return buf->flush (cb->buf, 0);
}

/* The block routine for a compression buffer. */

static int
compress_buffer_block (closure, block)
    void *closure;
    int block;
{
    struct compress_buffer *cb = (struct compress_buffer *) closure;
    if (block)
        return set_block (cb->buf);
    else
        return set_nonblock (cb->buf);
}

/* Shut down an input buffer. */

static int
compress_buffer_shutdown_input (closure)
void *closure;

360  {
  struct compress_buffer *cb = (struct compress_buffer *) closure;
  int zstatus;
  /* Pick up any trailing data, such as the checksum. */
  while (1)
  {
    int status, nread;
    char buf[100];
    status = compress_buffer_input (cb, buf, 0, sizeof buf, &nread);
    if (status == -1) break;
    if (status != 0)
      return status;
  }
  zstatus = inflateEnd (&cb->zstr);
  if (zstatus != Z_OK)
  {
    compress_error (0, zstatus, &cb->zstr, "inflateEnd");
    return EIO;
  }
  return buf_shutdown (cb->buf);
}

/* Shut down an output buffer. */

static int
390  compress_buffer_shutdown_output (closure)
void *closure;
  {
    struct compress_buffer *cb = (struct compress_buffer *) closure;
    int zstatus, status;
    do
    {
      char buffer[BUF_SIZE];
      cb->zstr.avail_out = BUF_SIZE;
      cb->zstr.next_out = (unsigned char *) buffer;
      zstatus = deflate (&cb->zstr, Z_FINISH);
      if (zstatus != Z_OK && zstatus != Z_STREAM_END)
      {
        compress_error (0, zstatus, &cb->zstr, "deflate finish");
        return EIO;
      }
      if (cb->zstr.avail_out != BUF_SIZE)
        buf_output (cb->buf, buffer, BUF_SIZE - cb->zstr.avail_out);
    } while (zstatus != Z_STREAM_END);
    zstatus = deflateEnd (&cb->zstr);
    if (zstatus != Z_OK)
    {
      compress_error (0, zstatus, &cb->zstr, "deflateEnd");
      return EIO;
    }
    status = buf_flush (cb->buf, 1);
    if (status != 0)
      return status;
    return buf_shutdown (cb->buf);
}

430  /* Here is our librarified gzip implementation. It is very minimal
   but attempts to be RFC1952 compliant. */
  /* Note that currently only the client uses the gzip library. If we
   make the server use it too (which should be straightforward), then
   filter_program, filter_gzip, and gzip can go away. */
  /* BUFI should contain SIZE bytes of gzipped data (RFC1952). We
   are to uncompress the data and write the result to the file
   descriptor FD. If something goes wrong, give an error message,
   mentioning FULLNAME as the name of the file for FD (and make it a
   fatal error if we can't recover from it). */
void
gunzip_and_write (fd, fullname, buf, size)
  int fd;
  char *fullname;
  unsigned char *buf;

size_t size;
    
    size_t pos;
    size_t stream zstr;
    int zstatus;
    unsigned char outbuf[22768];
    unsigned long zstatus;

    if ((buf[0] != 31 || buf[1] != 139)
        error (1, 0, "gzipped data does not start with gzip identification");
    if (buf[2] != 8)
      error (1, 0, "only the deflate compression method is supported");

    /* Skip over the fixed header, and then skip any of the variable-length fields. */
    pos = 16;
    if (buf[3] & 4)
        pos += buf[pos] + (buf[pos + 1] << 8) + 2;
    if (buf[3] & 8)
        pos += strlen (buf + pos) + 1;
    if (buf[3] & 16)
        pos += strlen (buf + pos) + 1;
    if (buf[3] & 2)
        pos += 2;

    memset (&zstr, 0, sizeof zstr);

    /* Passing a negative argument tells zlib not to look for a zlib (RFC1950) header. This is an undocumented feature; I suppose if we wanted to be anal we could synthesize a header instead, but why bother? */
    zstatus = inflateInit2 (&zstr, -15);

    if (zstatus != Z_OK)
      compress_error (1, zstatus, &zstr, fullname);

    /* I don't see why we should have to include the 8 byte trailer in avail. But I see that zlib/gzio.c does, and it seemed to fix a fairly rare bug in which we'd get a Z_BUF_ERROR for no obvious reason. */
    zstr.avail_in = size - pos;
    zstr.next_in = buf + pos;

    while (zstatus != Z_STREAM_END)
    {
        crc = crc32 (0, NULL, 0);

        do
        {
            zstr.avail_out = sizeof (outbuf);
            zstr.next_out = outbuf;
            zstatus = inflate (&zstr, Z_NO_FLUSH);
            if (zstatus != Z_OK && zstatus != Z_STREAM_END)
              compress_error (1, zstatus, &zstr, fullname);

            if (write (fd, outbuf, sizeof (outbuf) - zstr.avail_out) < 0)
              error (1, errno, "writing decompressed file " fullname);
            crc = crc32 (crc, outbuf, sizeof (outbuf) - zstr.avail_out);
        } while (zstatus != Z_STREAM_END);

        zstatus = inflateEnd (&zstr);
        if (zstatus != Z_OK)
          compress_error (0, zstatus, &zstr, fullname);

          error (1, 0, "CRC error uncompressing " fullname);

        if (zstatus != Z_OK)
          error (1, 0, "invalid length uncompressing " fullname);
    }

    /* Read all of FD and put the gripped data (RFC1952/RFC1951) into *BUF, replacing previous contents of *BUF. *BUF is malloc'd and *SIZE is its allocated size. Put the actual number of bytes of data in *LEN. If something goes wrong, give an error message mentioning FULLNAME as the name of the file for FD (and make it a fatal error if we can't recover from it). LEVEL is the compression level (1-9). */

    void read_and_gzip (fd, fullname, buf, size, len, level)
    
    int fd;
    char *fullname;
    unsigned char *buf;
    size_t size;
    size_t len;
    int level;

    { zstream zstr;
      int zstatus;

    
    }
unsigned char inbuf[8192];
int nread;
unsigned long crc;

if (*size < 1024)
{
    *size = 1024;
    *buf = (unsigned char *) realloc (*buf, *size);
}
*buf][8] = 31;
*buf][1] = 139;
*buf][2] = 8;
*buf][3] = 0;
/* Could set this based on level, but why bother? */
*buf][8] = 0;
*buf][9] = 295;
memset (&zstr, 0, sizeof zstr);
zstatus = deflateInit2 (&zstr, level, Z_DEFLATED, -15, 8,
Z_DEFAULT_STRATEGY);
crc = crc32 (0, NULL, 0);
if (zstatus != Z_OK)
    compress_error (1, zstatus, &zstr, fullname);
  zstr.avail_out = *size;
  zstr.next_out = *buf + 10;
while (1)
{
    int finish = 0;

    nread = read (fd, inbuf, sizeof inbuf);
    if (nread < 0)
        error (1, errno, "cannot read %s", fullname);
    else if (nread == 0)
        /* End of file. */
        finish = 1;
    crc = crc32 (crc, inbuf, nread);
    zstr.next_in = inbuf;
    zstr.avail_in = nread;

    do
    {
        size_t offset;

        /* I don’t see this documented anywhere, but deflate seems
tend to dump core sometimes if we pass it Z_FINISH and
a small (e.g. 2147 byte) avail_out. So we insist on at
least 4096 bytes (that is what zlib/gzip.c uses). */
        if (zstr.avail_out < 4096)
        {
            offset = zstr.next_out - *buf;
            *size = 2;
            *buf = realloc (*buf, *size);
            zstr.next_out = *buf + offset;
            zstr.avail_out = *size - offset;
        }

        zstatus = deflate (&zstr, finish ? Z_FINISH : 0);
        if (zstatus == Z_STREAM_END)
            goto done;
    }
    else if (zstatus != Z_OK)
        compress_error (0, zstatus, &zstr, fullname);
    while (zstr.avail_out == 0);
}
done:

    = (crc >> 8) & 0xff;
= (crc >> 16) & 0xff;
= (crc >> 24) & 0xff;

    = (crc >> 24) & 0xff;
= (crc >> 16) & 0xff;
= (crc >> 8) & 0xff;
= (crc >> 8) & 0xff;
= (crc >> 16) & 0xff;
= (crc >> 24) & 0xff;

    = zstr.total_out + 18;
    zstatus = deflateEnd (&zstr);
    if (zstatus != Z_OK)
        compress_error (0, zstatus, &zstr, fullname);
}
reendif /* defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT) */
Bibliography


   <http://www.maccvs.org/>,  
   <http://sourceforge.net/projects/maccvspro/>