Version Management
with
CVS

for CVS 1.11.18

Per Cederqvist et al

1 Overview

This chapter is for people who have never used CVS, and perhaps have never used version control software before.

If you are already familiar with CVS and are just trying to learn a particular feature or remember a certain command, you can probably skip everything here.

1.1 What is CVS?

CVS is a version control system. Using it, you can record the history of your source files.

For example, bugs sometimes creep in when software is modified, and you might not detect the bug until a long time after you make the modification. With CVS, you can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help.

You could of course save every version of every file you have ever created. This would however waste an enormous amount of disk space. CVS stores all the versions of a file in a single file in a clever way that only stores the differences between versions.

CVS also helps you if you are part of a group of people working on the same project. It is all too easy to overwrite each others' changes unless you are extremely careful. Some editors, like GNU Emacs, try to make sure that the same file is never modified by two people at the same time. Unfortunately, if someone is using another editor, that safeguard will not work. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done.

CVS started out as a bunch of shell scripts written by Dick Grune, posted to the newsgroup `comp.sources.unix` in the volume 6 release of July, 1986. While no actual code from these shell scripts is present in the current version of CVS much of the CVS conflict resolution algorithms come from them.

In April, 1989, Brian Berliner designed and coded CVS. Jeff Polk later helped Brian with the design of the CVS module and vendor branch support.

You can get CVS in a variety of ways, including free download from the Internet. For more information on downloading CVS and other CVS topics, see:

```
http://www.cvshome.org/
http://www.loria.fr/~molli/cvs-index.html
```

There is a mailing list, known as `info-cvs@gnu.org`, devoted to CVS. To subscribe or unsubscribe write to `info-cvs-request@gnu.org`. If you prefer a Usenet group, there is a one-way mirror (posts to the email list are usually sent to the news group, but not visa versa) of `info-cvs@gnu.org` at `news:gnu.cvs.help`. The right Usenet group for posts is `news:comp.software.config-mgmt` which is for CVS discussions (along with other configuration management systems). In the future, it might be possible to create a `comp.software.config-mgmt.cvs`, but probably only if there is sufficient CVS traffic on `news:comp.software.config-mgmt`.

You can also subscribe to the `bug-cvs@gnu.org` mailing list, described in more detail in Appendix H [BUGS], page 165. To subscribe send mail to `bug-cvs-request@gnu.org`. There is a two-way Usenet mirror (posts to the Usenet group are usually sent to the email list and visa versa) of `bug-cvs@gnu.org` named `news:gnu.cvs.bug`.

1.2 What is CVS not?

CVS can do a lot of things for you, but it does not try to be everything for everyone.

CVS is not a build system.

Though the structure of your repository and modules file interact with your build system (e.g. ‘`Makefile`’s), they are essentially independent.

CVS does not dictate how you build anything. It merely stores files for retrieval in a tree structure you devise.

CVS does not dictate how to use disk space in the checked out working directories. If you write your ‘`Makefile`’s or scripts in every directory so they have to know the relative positions of everything else, you wind up requiring the entire repository to be checked out.

If you modularize your work, and construct a build system that will share files (via links, mounts, `VPATH` in ‘`Makefile`’s, etc.), you can arrange your disk usage however you like.

But you have to remember that *any* such system is a lot of work to construct and maintain. CVS does not address the issues involved.

Of course, you should place the tools created to support such a build system (scripts, ‘`Makefile`’s, etc) under CVS.

Figuring out what files need to be rebuilt when something changes is, again, something to be handled outside the scope of CVS. One traditional approach is to use `make` for building, and use some automated tool for generating the dependencies which `make` uses.

See Chapter 14 [Builds], page 83, for more information on doing builds in conjunction with CVS.

CVS is not a substitute for management.

Your managers and project leaders are expected to talk to you frequently enough to make certain you are aware of schedules, merge points, branch names and release dates. If they don’t, CVS can’t help.

CVS is an instrument for making sources dance to your tune. But you are the piper and the composer. No instrument plays itself or writes its own music.

CVS is not a substitute for developer communication.

When faced with conflicts within a single file, most developers manage to resolve them without too much effort. But a more general definition of “conflict” includes problems too difficult to solve without communication between developers.

CVS cannot determine when simultaneous changes within a single file, or across a whole collection of files, will logically conflict with one another. Its concept of a *conflict* is purely textual, arising when two changes to the same base file are near enough to spook the merge (i.e. `diff3`) command.

CVS does not claim to help at all in figuring out non-textual or distributed conflicts in program logic.

For example: Say you change the arguments to function `X` defined in file '`A`'. At the same time, someone edits file '`B`', adding new calls to function `X` using the old arguments. You are outside the realm of CVS's competence.

Acquire the habit of reading specs and talking to your peers.

CVS does not have change control

Change control refers to a number of things. First of all it can mean *bug-tracking*, that is being able to keep a database of reported bugs and the status of each one (is it fixed? in what release? has the bug submitter agreed that it is fixed?). For interfacing CVS to an external bug-tracking system, see the '`rcsinfo`' and '`verifymsg`' files (see Appendix C [Administrative files], page 135).

Another aspect of change control is keeping track of the fact that changes to several files were in fact changed together as one logical change. If you check in several files in a single `cvs commit` operation, CVS then forgets that those files were checked in together, and the fact that they have the same log message is the only thing tying them together. Keeping a GNU style '`ChangeLog`' can help somewhat.

Another aspect of change control, in some systems, is the ability to keep track of the status of each change. Some changes have been written by a developer, others have been reviewed by a second developer, and so on. Generally, the way to do this with CVS is to generate a diff (using `cvs diff` or `diff`) and email it to someone who can then apply it using the `patch` utility. This is very flexible, but depends on mechanisms outside CVS to make sure nothing falls through the cracks.

CVS is not an automated testing program

It should be possible to enforce mandatory use of a test suite using the `commitinfo` file. I haven't heard a lot about projects trying to do that or whether there are subtle gotchas, however.

CVS does not have a built-in process model

Some systems provide ways to ensure that changes or releases go through various steps, with various approvals as needed. Generally, one can accomplish this with CVS but it might be a little more work. In some cases you'll want to use the '`commitinfo`', '`loginfo`', '`rcsinfo`', or '`verifymsg`' files, to require that certain steps be performed before cvs will allow a checkin. Also consider whether features such as branches and tags can be used to perform tasks such as doing work in a development tree and then merging certain changes over to a stable tree only once they have been proven.

1.3 A sample session

As a way of introducing CVS, we'll go through a typical work-session using CVS. The first thing to understand is that CVS stores all files in a centralized *repository* (see Chapter 2 [Repository], page 7); this section assumes that a repository is set up.

Suppose you are working on a simple compiler. The source consists of a handful of C files and a '`Makefile`'. The compiler is called '`tc`' (Trivial Compiler), and the repository is set up so that there is a module called '`tc`'.

1.3.1 Getting the source

The first thing you must do is to get your own working copy of the source for '`tc`'. For this, you use the `checkout` command:

```
$ cvs checkout tc
```

This will create a new directory called '`tc`' and populate it with the source files.

```
$ cd tc
$ ls
CVS         Makefile    backend.c   driver.c    frontend.c  parser.c
```

The '`CVS`' directory is used internally by CVS. Normally, you should not modify or remove any of the files in it.

You start your favorite editor, hack away at '`backend.c`', and a couple of hours later you have added an optimization pass to the compiler. A note to RCS and SCCS users: There is no need to lock the files that you want to edit. See Chapter 10 [Multiple developers], page 63, for an explanation.

1.3.2 Committing your changes

When you have checked that the compiler is still compilable you decide to make a new version of '`backend.c`'. This will store your new '`backend.c`' in the repository and make it available to anyone else who is using that same repository.

```
$ cvs commit backend.c
```

CVS starts an editor, to allow you to enter a log message. You type in "Added an optimization pass.", save the temporary file, and exit the editor.

The environment variable `$CVSEDITOR` determines which editor is started. If `$CVSEDITOR` is not set, then if the environment variable `$EDITOR` is set, it will be used. If both `$CVSEDITOR` and `$EDITOR` are not set then there is a default which will vary with your operating system, for example `vi` for unix or `notepad` for Windows NT/95.

In addition, CVS checks the `$VISUAL` environment variable. Opinions vary on whether this behavior is desirable and whether future releases of CVS should check `$VISUAL` or ignore it. You will be OK either way if you make sure that `$VISUAL` is either unset or set to the same thing as `$EDITOR`.

When CVS starts the editor, it includes a list of files which are modified. For the CVS client, this list is based on comparing the modification time of the file against the modification time that the file had when it was last gotten or updated. Therefore, if a file's modification time has changed but its contents have not, it will show up as modified. The simplest way to handle this is simply not to worry about it—if you proceed with the commit CVS will detect that the contents are not modified and treat it as an unmodified file. The next `update` will clue CVS in to the fact that the file is unmodified, and it will reset its stored timestamp so that the file will not show up in future editor sessions.

If you want to avoid starting an editor you can specify the log message on the command line using the '`-m`' flag instead, like this:

```
$ cvs commit -m "Added an optimization pass" backend.c
```

1.3.3 Cleaning up

Before you turn to other tasks you decide to remove your working copy of tc. One acceptable way to do that is of course

```
$ cd ..
$ rm -r tc
```

but a better way is to use the `release` command (see Section A.16 [release], page 119):

```
$ cd ..
$ cvs release -d tc
M driver.c
? tc
You have [1] altered files in this repository.
Are you sure you want to release (and delete) directory ‘tc’: n
** ‘release’ aborted by user choice.
```

The `release` command checks that all your modifications have been committed. If history logging is enabled it also makes a note in the history file. See Section C.8 [history file], page 147.

When you use the ‘`-d`’ flag with `release`, it also removes your working copy.

In the example above, the `release` command wrote a couple of lines of output. ‘`? tc`’ means that the file ‘`tc`’ is unknown to CVS. That is nothing to worry about: ‘`tc`’ is the executable compiler, and it should not be stored in the repository. See Section C.6 [cvsignore], page 145, for information about how to make that warning go away. See Section A.16.2 [release output], page 119, for a complete explanation of all possible output from `release`.

‘`M driver.c`’ is more serious. It means that the file ‘`driver.c`’ has been modified since it was checked out.

The `release` command always finishes by telling you how many modified files you have in your working copy of the sources, and then asks you for confirmation before deleting any files or making any note in the history file.

You decide to play it safe and answer *`n`* ⟨RET⟩ when `release` asks for confirmation.

1.3.4 Viewing differences

You do not remember modifying ‘`driver.c`’, so you want to see what has happened to that file.

```
$ cd tc
$ cvs diff driver.c
```

This command runs `diff` to compare the version of ‘`driver.c`’ that you checked out with your working copy. When you see the output you remember that you added a command line option that enabled the optimization pass. You check it in, and release the module.

```
$ cvs commit -m "Added an optimization pass" driver.c
Checking in driver.c;
/usr/local/cvsroot/tc/driver.c,v  <--  driver.c
new revision: 1.2; previous revision: 1.1
```

```
done
$ cd ..
$ cvs release -d tc
? tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory ‘tc’: y
```

2 The Repository

The CVS *repository* stores a complete copy of all the files and directories which are under version control.

Normally, you never access any of the files in the repository directly. Instead, you use CVS commands to get your own copy of the files into a *working directory*, and then work on that copy. When you've finished a set of changes, you check (or *commit*) them back into the repository. The repository then contains the changes which you have made, as well as recording exactly what you changed, when you changed it, and other such information. Note that the repository is not a subdirectory of the working directory, or vice versa; they should be in separate locations.

CVS can access a repository by a variety of means. It might be on the local computer, or it might be on a computer across the room or across the world. To distinguish various ways to access a repository, the repository name can start with an *access method*. For example, the access method `:local:` means to access a repository directory, so the repository `:local:/usr/local/cvsroot` means that the repository is in '`/usr/local/cvsroot`' on the computer running CVS. For information on other access methods, see Section 2.9 [Remote repositories], page 19.

If the access method is omitted, then if the repository starts with '`/`', then `:local:` is assumed. If it does not start with '`/`' then either `:ext:` or `:server:` is assumed. For example, if you have a local repository in '`/usr/local/cvsroot`', you can use `/usr/local/cvsroot` instead of `:local:/usr/local/cvsroot`. But if (under Windows NT, for example) your local repository is '`c:\src\cvsroot`', then you must specify the access method, as in `:local:c:/src/cvsroot`.

The repository is split in two parts. '`$CVSROOT/CVSROOT`' contains administrative files for CVS. The other directories contain the actual user-defined modules.

2.1 Telling CVS where your repository is

There are several ways to tell CVS where to find the repository. You can name the repository on the command line explicitly, with the `-d` (for "directory") option:

```
cvs -d /usr/local/cvsroot checkout yoyodyne/tc
```

Or you can set the `$CVSROOT` environment variable to an absolute path to the root of the repository, '`/usr/local/cvsroot`' in this example. To set `$CVSROOT`, `csh` and `tcsh` users should have this line in their '`.cshrc`' or '`.tcshrc`' files:

```
setenv CVSROOT /usr/local/cvsroot
```

`sh` and `bash` users should instead have these lines in their '`.profile`' or '`.bashrc`':

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
```

A repository specified with `-d` will override the `$CVSROOT` environment variable. Once you've checked a working copy out from the repository, it will remember where its repository is (the information is recorded in the '`CVS/Root`' file in the working copy).

The `-d` option and the '`CVS/Root`' file both override the `$CVSROOT` environment variable. If `-d` option differs from '`CVS/Root`', the former is used. Of course, for proper operation they should be two ways of referring to the same repository.

2.2 How data is stored in the repository

For most purposes it isn't important *how* CVS stores information in the repository. In fact, the format has changed in the past, and is likely to change in the future. Since in almost all cases one accesses the repository via CVS commands, such changes need not be disruptive.

However, in some cases it may be necessary to understand how CVS stores data in the repository, for example you might need to track down CVS locks (see Section 10.5 [Concurrency], page 67) or you might need to deal with the file permissions appropriate for the repository.

2.2.1 Where files are stored within the repository

The overall structure of the repository is a directory tree corresponding to the directories in the working directory. For example, supposing the repository is in

```
/usr/local/cvsroot
```

here is a possible directory tree (showing only the directories):

```
/usr
 |
 +--local
 |   |
 |   +--cvsroot
 |   |    |
 |   |    +--CVSROOT
          |      (administrative files)
          |
          +--gnu
          |   |
          |   +--diff
          |   |   (source code to GNU diff)
          |   |
          |   +--rcs
          |   |   (source code to RCS)
          |   |
          |   +--cvs
          |       (source code to CVS)
          |
          +--yoyodyne
              |
              +--tc
              |    |
              |    +--man
              |    |
              |    +--testing
              |
              +--(other Yoyodyne software)
```

With the directories are *history files* for each file under version control. The name of the history file is the name of the corresponding file with '`,v`' appended to the end. Here is what the repository for the '`yoyodyne/tc`' directory might look like:

```
$CVSROOT
  |
  +--yoyodyne
  |   |
  |   +--tc
  |   |   |
          +--Makefile,v
          +--backend.c,v
          +--driver.c,v
          +--frontend.c,v
          +--parser.c,v
          +--man
          |    |
          |    +--tc.1,v
          |
          +--testing
               |
               +--testpgm.t,v
               +--test2.t,v
```

The history files contain, among other things, enough information to recreate any revision of the file, a log of all commit messages and the user-name of the person who committed the revision. The history files are known as *RCS files*, because the first program to store files in that format was a version control system known as RCS. For a full description of the file format, see the `man` page *rcsfile(5)*, distributed with RCS, or the file '`doc/RCSFILES`' in the CVS source distribution. This file format has become very common—many systems other than CVS or RCS can at least import history files in this format.

The RCS files used in CVS differ in a few ways from the standard format. The biggest difference is magic branches; for more information see Section 5.5 [Magic branch numbers], page 44. Also in CVS the valid tag names are a subset of what RCS accepts; for CVS's rules see Section 4.4 [Tags], page 34.

2.2.2 File permissions

All '`,v`' files are created read-only, and you should not change the permission of those files. The directories inside the repository should be writable by the persons that have permission to modify the files in each directory. This normally means that you must create a UNIX group (see group(5)) consisting of the persons that are to edit the files in a project, and set up the repository so that it is that group that owns the directory. (On some systems, you also need to set the set-group-ID-on-execution bit on the repository directories (see chmod(1)) so that newly-created files and directories get the group-ID of the parent directory rather than that of the current process.)

This means that you can only control access to files on a per-directory basis.

Note that users must also have write access to check out files, because CVS needs to create lock files (see Section 10.5 [Concurrency], page 67). You can use LockDir in CVS-

ROOT/config to put the lock files somewhere other than in the repository if you want to allow read-only access to some directories (see Section C.10 [config], page 148).

Also note that users must have write access to the '`CVSROOT/val-tags`' file. CVS uses it to keep track of what tags are valid tag names (it is sometimes updated when tags are used, as well as when they are created).

Each RCS file will be owned by the user who last checked it in. This has little significance; what really matters is who owns the directories.

CVS tries to set up reasonable file permissions for new directories that are added inside the tree, but you must fix the permissions manually when a new directory should have different permissions than its parent directory. If you set the `CVSUMASK` environment variable that will control the file permissions which CVS uses in creating directories and/or files in the repository. `CVSUMASK` does not affect the file permissions in the working directory; such files have the permissions which are typical for newly created files, except that sometimes CVS creates them read-only (see the sections on watches, Section 10.6.1 [Setting a watch], page 68; -r, Section A.4 [Global options], page 88; or `CVSREAD`, Appendix D [Environment variables], page 151).

Note that using the client/server CVS (see Section 2.9 [Remote repositories], page 19), there is no good way to set `CVSUMASK`; the setting on the client machine has no effect. If you are connecting with `rsh`, you can set `CVSUMASK` in '`.bashrc`' or '`.cshrc`', as described in the documentation for your operating system. This behavior might change in future versions of CVS; do not rely on the setting of `CVSUMASK` on the client having no effect.

Using pserver, you will generally need stricter permissions on the CVSROOT directory and directories above it in the tree; see Section 2.9.3.3 [Password authentication security], page 24.

Some operating systems have features which allow a particular program to run with the ability to perform operations which the caller of the program could not. For example, the set user ID (setuid) or set group ID (setgid) features of unix or the installed image feature of VMS. CVS was not written to use such features and therefore attempting to install CVS in this fashion will provide protection against only accidental lapses; anyone who is trying to circumvent the measure will be able to do so, and depending on how you have set it up may gain access to more than just CVS. You may wish to instead consider pserver. It shares some of the same attributes, in terms of possibly providing a false sense of security or opening security holes wider than the ones you are trying to fix, so read the documentation on pserver security carefully if you are considering this option (Section 2.9.3.3 [Password authentication security], page 24).

2.2.3 File Permission issues specific to Windows

Some file permission issues are specific to Windows operating systems (Windows 95, Windows NT, and presumably future operating systems in this family. Some of the following might apply to OS/2 but I'm not sure).

If you are using local CVS and the repository is on a networked file system which is served by the Samba SMB server, some people have reported problems with permissions. Enabling WRITE=YES in the samba configuration is said to fix/workaround it. Disclaimer: I haven't investigated enough to know the implications of enabling that option, nor do I know whether there is something which CVS could be doing differently in order to avoid the problem. If you find something out, please let us know as described in Appendix H [BUGS], page 165.

2.2.4 The attic

You will notice that sometimes CVS stores an RCS file in the `Attic`. For example, if the CVSROOT is '`/usr/local/cvsroot`' and we are talking about the file '`backend.c`' in the directory '`yoyodyne/tc`', then the file normally would be in

```
/usr/local/cvsroot/yoyodyne/tc/backend.c,v
```

but if it goes in the attic, it would be in

```
/usr/local/cvsroot/yoyodyne/tc/Attic/backend.c,v
```

instead. It should not matter from a user point of view whether a file is in the attic; CVS keeps track of this and looks in the attic when it needs to. But in case you want to know, the rule is that the RCS file is stored in the attic if and only if the head revision on the trunk has state `dead`. A `dead` state means that file has been removed, or never added, for that revision. For example, if you add a file on a branch, it will have a trunk revision in `dead` state, and a branch revision in a non-`dead` state.

2.2.5 The CVS directory in the repository

The '`CVS`' directory in each repository directory contains information such as file attributes (in a file called '`CVS/fileattr`'. In the future additional files may be added to this directory, so implementations should silently ignore additional files.

This behavior is implemented only by CVS 1.7 and later; for details see Section 10.6.5 [Watches Compatibility], page 71.

The format of the '`fileattr`' file is a series of entries of the following form (where '`{`' and '`}`' means the text between the braces can be repeated zero or more times):

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. CVS 1.10 and later will preserve them.

Note that the order of the lines is not significant; a program writing the fileattr file may rearrange them at its convenience.

There is currently no way of quoting tabs or line feeds in the filename, '`=`' in *attrname*, '`;`' in *attrval*, etc. Note: some implementations also don't handle a NUL character in any of the fields, but implementations are encouraged to allow it.

By convention, *attrname* starting with '`_`' is for an attribute given special meaning by CVS; other *attrname*s are for user-defined attributes (or will be, once implementations start supporting 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 zero or more of edit,unedit,commit separated by '`+`' (that is, 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 (or equivalent) happened, and *hostname* and *pathname* are for the working directory.

Example:

```
Ffile1 _watched=;_watchers=joe>edit,mary>commit
Ffile2 _watched=;_editors=sue>8 Jan 1975+workstn1+/home/sue/cvs
D _watched=
```

means that the file '`file1`' should be checked out read-only. Furthermore, joe is watching for edits and mary is watching for commits. The file '`file2`' should be checked out read-only; sue started editing it on 8 Jan 1975 in the directory '`/home/sue/cvs`' on the machine `workstn1`. Future files which are added should be checked out read-only. To represent this example here, we have shown a space after '`D`', '`Ffile1`', and '`Ffile2`', but in fact there must be a single tab character there and no spaces.

2.2.6 CVS locks in the repository

For an introduction to CVS locks focusing on user-visible behavior, see Section 10.5 [Concurrency], page 67. The following section is aimed at people who are writing tools which want to access a CVS repository without interfering with other tools accessing the same repository. If you find yourself confused by concepts described here, like *read lock*, *write lock*, and *deadlock*, you might consult the literature on operating systems or databases.

Any file in the repository with a name starting with '`#cvs.rfl.`' is a read lock. Any file in the repository with a name starting with '`#cvs.wfl`' is a write lock. Old versions of CVS (before CVS 1.5) also created files with names starting with '`#cvs.tfl`', but they are not discussed here. The directory '`#cvs.lock`' serves as a master lock. That is, one must obtain this lock first before creating any of the other locks.

To obtain a read lock, first create the '`#cvs.lock`' directory. This operation must be atomic (which should be true for creating a directory under most operating systems). If it fails because the directory already existed, wait for a while and try again. After obtaining the '`#cvs.lock`' lock, create a file whose name is '`#cvs.rfl.`' followed by information of your choice (for example, hostname and process identification number). Then remove the '`#cvs.lock`' directory to release the master lock. Then proceed with reading the repository. When you are done, remove the '`#cvs.rfl`' file to release the read lock.

To obtain a write lock, first create the '`#cvs.lock`' directory, as with read locks. Then check that there are no files whose names start with '`#cvs.rfl.`'. If there are, remove '`#cvs.lock`', wait for a while, and try again. If there are no readers, then create a file whose name is '`#cvs.wfl`' followed by information of your choice (for example, hostname and process identification number). Hang on to the '`#cvs.lock`' lock. Proceed with writing the repository. When you are done, first remove the '`#cvs.wfl`' file and then the '`#cvs.lock`' directory. Note that unlike the '`#cvs.rfl`' file, the '`#cvs.wfl`' file is just informational;

it has no effect on the locking operation beyond what is provided by holding on to the '`#cvs.lock`' lock itself.

Note that each lock (write lock or read lock) only locks a single directory in the repository, including '`Attic`' and '`CVS`' but not including subdirectories which represent other directories under version control. To lock an entire tree, you need to lock each directory (note that if you fail to obtain any lock you need, you must release the whole tree before waiting and trying again, to avoid deadlocks).

Note also that CVS expects write locks to control access to individual '`foo,v`' files. RCS has a scheme where the '`,foo,`' file serves as a lock, but CVS does not implement it and so taking out a CVS write lock is recommended. See the comments at rcs_internal_lockfile in the CVS source code for further discussion/rationale.

2.2.7 How files are stored in the CVSROOT directory

The '`$CVSROOT/CVSROOT`' directory contains the various administrative files. In some ways this directory is just like any other directory in the repository; it contains RCS files whose names end in '`,v`', and many of the CVS commands operate on it the same way. However, there are a few differences.

For each administrative file, in addition to the RCS file, there is also a checked out copy of the file. For example, there is an RCS file '`loginfo,v`' and a file '`loginfo`' which contains the latest revision contained in '`loginfo,v`'. When you check in an administrative file, CVS should print

```
cvs commit: Rebuilding administrative file database
```

and update the checked out copy in '`$CVSROOT/CVSROOT`'. If it does not, there is something wrong (see Appendix H [BUGS], page 165). To add your own files to the files to be updated in this fashion, you can add them to the '`checkoutlist`' administrative file (see Section C.7 [checkoutlist], page 146).

By default, the '`modules`' file behaves as described above. If the modules file is very large, storing it as a flat text file may make looking up modules slow (I'm not sure whether this is as much of a concern now as when CVS first evolved this feature; I haven't seen benchmarks). Therefore, by making appropriate edits to the CVS source code one can store the modules file in a database which implements the `ndbm` interface, such as Berkeley db or GDBM. If this option is in use, then the modules database will be stored in the files '`modules.db`', '`modules.pag`', and/or '`modules.dir`'.

For information on the meaning of the various administrative files, see Appendix C [Administrative files], page 135.

2.3 How data is stored in the working directory

While we are discussing CVS internals which may become visible from time to time, we might as well talk about what CVS puts in the '`CVS`' directories in the working directories. As with the repository, CVS handles this information and one can usually access it via CVS commands. But in some cases it may be useful to look at it, and other programs, such as the `jCVS` graphical user interface or the `VC` package for emacs, may need to look at it. Such programs should follow the recommendations in this section if they hope to be able to work with other programs which use those files, including future versions of the programs just mentioned and the command-line CVS client.

The ‘`CVS`’ directory contains several