

Linux CVS Tutorial

1. Introduction

Tutorial layout

This tutorial has two parts. The first shows you how to use CVS as a non-developer, i.e. how to get sources from CVS and keep them updated. The second part introduces you to using CVS as a developer, showing you how to modify, add and remove files on CVS and perform other developer-related tasks. If you are new to CVS, it's recommended that you begin in the first section and proceed to the second section; if you have some basic CVS experience but are going to be using CVS as a full-fledged developer for the first time, you should find everything you need in the second section, but you may want to go through the first section as a review.

What is CVS and what does it do?

CVS is a client/server system allowing developers to store their projects in a central location, called a repository. Using the cvs client tools, developers can make changes to the contents of the repository. In turn, the cvs repository tracks every change made to every file, creating a complete history of the evolution of the development project. Developers can request older versions of a particular source file, view a log of changes, and perform other useful tasks as needed.

The role of CVS

A lot of open software projects have their own CVS servers, which are used by the project developers as a central repository for all their work. Developers often make improvements to the sources in the CVS repository on a daily basis; and often, these developers are scattered around the world, yet CVS provides the necessary mechanism to unite their project into a centralized, cohesive whole. CVS creates the "organizational glue" that allows these developers to make improvements to the code without stepping on each other's toes, losing important data or missing each other's critical updates to particular source files.

CVS -- the latest developer sources

When the developers are ready, they'll roll some their current work on CVS into a .tar.gz file and release it as a new official version of their software package. However, the latest official release sometimes isn't recent enough, for a variety of possible reasons. In the first section of this tutorial, I'll show you how to use CVS for this purpose -- acquiring the latest and greatest developer version of the sources for your own personal use.

CVS -- do you have it?

Before you can actually use CVS, you need to get it installed on your system. The easiest way to test to see if it's installed is to type:

Code Listing 1.1: Starting CVS

```
# cvs
```

If a cvs command is found, then you've got it! Otherwise, you'll need to either track down a binary package for your particular distribution, or install it from sources. Installing CVS from sources is actually quite simple, and I'll show you how in the next panel.

Installing CVS from sources

Installing CVS from sources is easy. First, grab the `cvs-1.11.tar.gz` tarball from `ftp://ftp.cvshome.org/pub/cvs-1.11/cvs-1.11.tar.gz` (if there's a newer version listed here, you might as well grab the new one instead.) Then perform the following steps (command output has been omitted for brevity):

Code Listing 1.2: Installing CVS using a tarball

```
# tar xzvf cvs-1.11.tar.gz
# cd cvs-1.11
# ./configure
# make
# make install
```

Now you should be ready to go.

Installing CVS using a package management system

Many distributions provide an easy method to install software. For instance, the Gentoo distribution provides the `emerge` command. To install `cvs`, just type in `emerge cvs`:

Code Listing 1.3: Installing CVS using emerge

```
# emerge cvs
```

The CVSROOT

Before we begin, there are a few CVS fundamentals that you need to know. The first is that in order to connect to a CVS repository, you first need to know a path called the "CVSROOT". The CVSROOT is a string, like a URL, that tells the `cvs` command where the remote repository is and how we'd like to connect to it. Just to make things interesting, CVS has a number of CVSROOT formats, depending on whether the CVS repository is local or remote and what method you're going to use to connect to it. Here are some example CVSROOTs, along with explanations...

A local CVSROOT

Code Listing 1.4: Setting CVSROOT

```
CVSROOT=/var/cvsroot
```

This is an example of a local CVSROOT path; you'd use a CVSROOT like this if you wanted to connect to a local repository that exists at `/var/cvsroot`; or maybe you have a repository mounted via NFS at `/var/cvsroot`.

A remote password server CVSROOT

Code Listing 1.5: Setting CVSROOT with authentication

```
CVSROOT=:pserver:cvs@foo.bar.com:/var/cvsroot
```

Here's an example of a CVSROOT for a remote repository that exists on the `foo.bar.com` host and lives in the `/var/cvsroot` directory on that machine. The leading `:pserver:` part tells our client to connect to this remote machine using the CVS password server protocol, a protocol that's built-in to CVS. Typically, public CVS repositories use the password server protocol to allow access to anonymous users.

A remote rsh/ssh CVSROOT

Code Listing 1.6: RSH/SSH CVSROOT

```
CVSROOT=drobbins@foo.bar.com:/data/cvs
```

Here's an example of a CVSROOT that uses the RSH or SSH protocol; in this example, the CVS server will attempt to access the repository on foo.bar.com using the drobbins account. If the CVS_RSH environment variable is set to "ssh", then our cvs client will attempt to use ssh to connect; otherwise rsh will be used. The ssh access method is popular with those who are concerned about security; however, neither the RSH or SSH method provides a way for anonymous users to get the sources. In order to use this method, you must have a login account at foo.bar.com.

A few more things...

In addition to the CVSROOT, you'll also need to know the name of the module (collection of sources) that you'd like to check out, as well as an anonymous password that you'll need to log in to the CVS password server. Unlike anonymous ftp, there is no "standard" format for the anonymous password, so you'll need to get the specific password from the developer web site or the developers themselves. Once you have all this info, you're ready to begin.

Interacting with CVS, part 1

Grabbing the sources is a two-stage process. First, we log in to the password server. Then, we grab the sources with a checkout command. Here's an example set of commands that can be used to check out the latest Samba sources, a popular UNIX/Windows integration project:

Code Listing 1.7: Setting up CVSROOT

```
# export CVSROOT=:pserver:cvs@pserver.samba.org:/cvsroot
```

This first command sets the CVSROOT environment variable. If you don't set this variable, the following two commands will require an additional -d :pserver:cvs@pserver.samba.org:/cvsroot following the cvs command. Exporting the CVSROOT saves a us bit of typing.

Interacting with CVS, part 2

Here are the commands needed to get a current copy of the developer sources. You may want to jump forward to the next panel to read the explanation of these commands, and then jump back here:

Code Listing 1.8: Checking out sources

```
# cvs login          (Logging in to cvs@pserver.samba.org)
CVS password: (enter password here)
# cvs -z5 co samba
U samba/COPYING
U samba/Manifest
U samba/README
U samba/Read-Manifest-Now
U samba/Roadmap
U samba/WHATSNEW.txt
(this is just a snippet of the complete cvs co output)
```

Interacting with CVS -- the explanation

The first cvs command above logs us in to the pserver, and the second tells our CVS client to check out ("co") the samba module using a gzip compression level of 5 ("-z5") to speed up the transfer over a slow link. For every new file that is created locally, cvs prints out a "U [path]" indicating that this particular file has been updated on disk.

Checkout complete

Once the checkout command completes, you'll see a "samba" directory in your current working directory that contains the latest sources. You'll also notice that all the directories have a "CVS" directory inside them -- CVS stores accounting information inside these directories, and they can safely be ignored. From this point forward, we don't need to worry about having the CVSROOT environment variable set nor do we need to specify it on the command line because it's now cached inside all those extra "CVS" directories. Remember -- you only need to have the CVSROOT set for the initial login and checkout.

Updating the sources

Well, there you are -- fresh sources! Now that you have the sources, you can go ahead and compile and install them, inspect them, or do whatever you like with them.

Every now and then, you may want to bring your checked-out source directory in-sync with the current version on CVS. To do this, you don't need to log in to the pserver again; your authentication info is also cached by cvs inside those "CVS" accounting directories. First, enter the main checked-out directory (in this case "samba"), and type:

Code Listing 1.9: Updating your sources

```
# cvs update -dP
```

Looking at "cvs update", part 1

If there are any new files, cvs will output "U [path]" lines for each one as it updates them. Also, if you compiled the sources, you will probably see a lot of "? [path]" lines; these are object files that cvs notices are not from the remote repository.

Looking at "cvs update", part 2

Also, notice the two command-line options we used for "cvs update". "-d" tells cvs to create any new directories that may have been added to the repository (this doesn't happen by default), and "-P" tells cvs to remove any empty directories from your locally checked-out copy of the sources. "-P" is a good idea, because cvs has a tendency to collect a lot of empty (once used, but now abandoned) directory trees over time.

When it comes to simply grabbing the latest sources, that's about all you need to know. Now, we take a look at how to interact with CVS as a developer.

2. CVS for developers

Modifying files

As a developer, you'll need to modify files on CVS. To do this, simply make the appropriate changes to your local copy of the repository. The changes you make to the sources are not applied to the remote repository until you explicitly tell cvs to "commit" your changes. When you've tested all your modifications to ensure that they work properly and you're ready to apply your changes to the repository, follow this two-step process. First, update your sources by typing the following command in your main source directory:

Code Listing 2.1: Updating sources and directories

```
# cvs update -dP
```

CVS merges others' changes

As we've seen earlier, "cvs update" will bring your sources up-to-date with the current version in the repository -- but what happens to the changes you've made? Don't worry, they aren't thrown away. If another developer made changes to a file that you haven't touched, your local file will be updated so that it's in-sync with the version on the repository.

And, if you modified lines 1-10 of a local file, and another developer deleted lines 40-50, added 12 new lines at the end of the file, modified lines 30-40 and then committed their changes to the repository before you, cvs will intelligently merge these changes into your locally modified copy so that none of your changes are lost. This allows two or more developers to work on different parts of the same file at the same time.

Merging isn't perfect

However, if two or more developers have made changes to the same region of the same file, then things get a bit more complicated. If that happens, then cvs will tell you that there's been a conflict. No work will be lost, but a bit of manual intervention will be required, since cvs now requires your input on how to merge the conflicting changes.

The commit

We'll look at exactly how conflicts can be resolved in just a little bit, but for now, let's assume that there are no conflicts after you typed "cvs update -dP" -- there usually aren't. With no conflicts, your local sources are up-to-date, and you're ready to commit your changes to the repository by typing the following command in your main source directory:

Code Listing 2.2: Committing changes

```
# cvs commit
```

What commit does

"cvs commit" doesn't just apply your changes to the repository. Before actually committing your changes to the remote repository, cvs will fire up your default editor so that you can type in a description of your modifications. Once you've entered your comments, saved the file and exited the editor, your changes (and comments) will be applied to the remote repository and will be available to the other developers in your team.

Viewing the log

It's really easy to view the complete history of a particular file, along with any comments that the developers (including you) may have made when committing. To view this information, type:

Code Listing 2.3: View log information

```
# cvs log myfile.c
```

The "cvs log" command is recursive, so if you want to see the complete log for an entire directory tree, just enter the directory and type:

Code Listing 2.4: View log information with a pager

```
# cvs log | less
```

Commit options

You may want to use another editor than the one cvs starts by default when you type "cvs commit". If so, simply set the EDITOR environment variable to the name of the editor you want to use. Putting a setting such as this one in your ~/.bashrc would be a good idea:

Code Listing 2.5: Setting your editor

```
export EDITOR=jpico
```

Alternatively, you can also specify a log message as a command line option so that cvs doesn't need to load up an editor in the first place:

Code Listing 2.6: Committing changes with a small log information

```
# cvs commit -m 'I fixed a few silly bugs in portage.py'
```

The .cvsrc file

Before we continue looking at more cvs commands, I recommend setting up a ~/.cvsrc file. By creating a .cvsrc file in your home directory, you can tell cvs to use preferred command-line options by default so that you don't have to remember to type them in each time. Here's a recommended default .cvsrc file:

Code Listing 2.7: Recommended defaults

```
cvs -q
diff -u -b -B
checkout -P
update -d -P
```

The .cvsrc file, continued

In addition to setting useful options for a bunch of cvs commands, the first line of the .cvsrc puts cvs into quiet mode, which has the primary benefit of making the cvs update output more concise and readable. Also, once you have this .cvsrc in place, you can type cvs update instead of typing cvs update -dP.

Adding a file to the repository

It's really easy to add a source file to CVS. First, create the file with your favorite text editor. Then, type the following:

Code Listing 2.8: Adding a file

```
# cvs add myfile.c
cvs server: use 'cvs commit' to add this file permanently
```

This will tell cvs to add this file to the repository the next time you do a cvs commit. Until then, other developers won't be able to see it.

Adding a directory to the repository

The process of adding a directory to CVS is similar:

Code Listing 2.9: Adding a directory

```
# mkdir foo
# cvs add foo
Directory /var/cvsroot/mycode/foo added to the repository
```

Unlike adding a file, when you add a directory it appears on the repository immediately; a cvs commit isn't required. Once you add a local directory to cvs, you'll notice that a "CVS" directory will be created inside it to serve as a container for cvs accounting data. Thus, you can easily tell if a particular directory has been added to cvs by looking inside it for a "CVS" directory.

"cvs add" notes

Oh, and as you might guess, before you add a file or directory to the repository, you must make sure that its parent directory has already been added to CVS. Otherwise, you'll get an error that looks like this:

Code Listing 2.10: Adding a file, but receive a failure

```
# cvs add myfile.c
cvs add: cannot open CVS/Entries for reading: No such file or directory
cvs [ add aborted]: no repository
```

Getting familiar with "cvs update", part 1

Before we take a look at how to resolve conflicts, let's get familiar with the output of the "cvs update" command. If you created a ~/.cvsrc file that contains a "cvs -q" line, you'll find "cvs update" output a lot easier to read. "cvs update" informs you of what it does and sees by printing out a single character, a space, and a filename; as an example:

Code Listing 2.11: Updating CVS

```
# cvs update -dP
? distfiles
? packages
? profiles
```

Getting familiar with "cvs update", part 2

"cvs update" uses the "?" character to tell you that it doesn't know anything about these particular files that it finds in the local copy of your repository. They're not officially part of the repository, nor have they been scheduled for addition. Here's a list of all the other single-character informational messages that CVS uses:

Code Listing 2.12: Informational message: U

```
U [path]
```

Used when a new file is created in your local repository, or an untouched (by you) file has been updated.

Code Listing 2.13: Informational message: A

A [path]

This file is scheduled for addition and will be officially added to the repository when you do a cvs commit.

Getting familiar with "cvs update", part 3

Code Listing 2.14: Informational message: R

R [path]

Like "A", an "R" lets you know that this file is scheduled for removal. The file will be removed from the repository as soon as you cvs commit.

Code Listing 2.15: Informational message: M

M [path]

This means that this file has been modified by you; additionally, it's possible that new changes from the repository were merged into this file successfully.

Code Listing 2.16: Informational message: C

C [path]

The "C" character indicates that this file has a conflict and requires manual fixing before you can "cvs commit" your changes.

Resolving conflicts intro

Now, let's take a look at how to resolve a conflict. I'm very involved in the Gentoo Linux project, and we have our own cvs server set up at cvs.gentoo.org. We developers spend most of our time hacking away at the sources inside the "gentoo-x86" module. Inside the gentoo-x86 module, we have a file called "ChangeLog" that houses (you guessed it) a description of the major changes we make to the files in the repository.

An example conflict

Because this file is modified nearly every time a developer makes a major change to CVS, it's a primary source of conflicts -- here's an example conflict. Let's say I add the following lines to the top of the ChangeLog:

Code Listing 2.17: ChangeLog entry

```
date 25 Feb 2001          This is the thing I added myself
```

However, let's say that before I'm able to commit these three new lines, another developer adds these lines to the top of the ChangeLog and commits their changes:

Code Listing 2.18: ChangeLog entry 2

```
date 25 Feb 2001      This is the part added by another developer
```

An example conflict, continued

Now, when I run `cvs update -dP` (as you should before every commit), cvs isn't able to merge the changes into my local copy of ChangeLog because we both have added lines to the exact same part of the file -- how is cvs to know which version to use? So, I get the following error from CVS:

Code Listing 2.19: CVS error

```
RCS file: /var/cvsroot/gentoo-x86/ChangeLog,v
retrieving revision 1.362
retrieving revision 1.363
Merging differences between 1.362 and 1.363 into ChangeLog
rcsmerge: warning: conflicts during merge
cvs server: conflicts found in ChangeLog
C ChangeLog
```

Conflict resolution, part 1

Argh -- a conflict! Fortunately, fixing conflicts is easy. If I fire up my favorite text editor, I see the following text at the top of the ChangeLog file:

Code Listing 2.20: ChangeLog conflict

```
>>>>>> ChangeLog
date 25 Feb 2001   This is the thing I added myself
=====
date 25 Feb 2001   This is the part added by another developer
>>>>>> 1.363
```

Conflict resolution, part 2

Instead of choosing one version over the other, cvs has added both versions to the ChangeLog file, and surrounded them with special separators to clearly mark the conflict in question. Now, it's up to me to replace this region with the text that should appear in ChangeLog; in this case, the replacement text is neither one or the other version but a combination of both:

Code Listing 2.21: ChangeLog entry

```
date 25 Feb 2001   This is the thing I added myself
This is the part added by another developer
```

Now that I've replaced the conflicting region of the file with the appropriate text (and removed the "=====", etc markers), I can now commit my changes to cvs without any problems.

Conflict resolution tips

Whenever you need to edit a file for conflicts, make sure that you scan the entire file so that you catch all of them; if you forget to address a particular conflict, cvs won't allow you to commit until it's resolved! It's also obviously very important to remove the special markers that cvs added to the conflicting file. Another tip -- if you make a mistake while fixing the conflict and then ("D'oh!") accidentally save your changes, you can find an original copy of your version in the file ".#filename.version".

Removing a file

Now, it's time to learn our final CVS skill -- removing files from the repository. Removing a file is a two-stage process. First, delete the file from your local copy of the sources, and then execute the appropriate cvs remove command:

Code Listing 2.22: Removing a file

```
# rm myoldfile.c
# cvs remove myoldfile.c
```

Removing a file, continued

The file will then be scheduled for removal from the repository the next time you do a commit. Once committed, the file will be officially deleted from the current version of the repository. However, cvs won't throw this file away, and will still keep a complete record of its contents and its history, just in case you need it back in the future. This is just one of the many ways that cvs protects your valuable source code.

cvs remove is recursive, which means that you can delete a bunch of files, and then run the cvs remove command with no other arguments from a parent directory. Doing this will cause all of the deleted files to be tagged for removal at the next commit.

Removing a directory

If you'd like to remove an entire directory, I recommend the following process. First, physically delete and "cvs remove" all files in the directory:

Code Listing 2.23: Removing a directory

```
# rm *.c
# cvs remove
```

Removing a directory, continued

Then, perform a commit:

Code Listing 2.24: Commit your changes

```
# cvs commit
```

Here comes the trick. Perform the following steps to delete the directory:

Code Listing 2.25: Removing a directory

```
# cd ..
# cvs remove mydir
# rm -rf mydir
```

Notice that removing the directory didn't require another commit -- directories are added to and removed from the repository in real-time.

Complete!

Your introduction to CVS is complete -- I hope that this tutorial has been helpful. There's much more to CVS than I've been able to cover in this introductory tutorial, but thankfully there are a bunch of great CVS resources you can use to further expand your CVS knowledge:

<http://www.cvshome.org> is the home of CVS development, and offers a bunch of documentation on CVS, including the official CVS documentation online

The CVS Version Control for Web Site Projects site has good info on how to use CVS for developing web sites

Karl Fogel has written a book called Open Source Development with CVS. A number of chapters are available for free from the website.

cvsweb is a really great CGI script that provides a web interface to your CVS repository; excellent for browsing.

The CVS Bubbles site has a bunch of good resources including a CVS FAQ-o-matic.

About this document

The original version of this article was first published on IBM developerWorks, and is property of Westtech Information Services. This document is an updated version of the original article, and contains various improvements made by the Gentoo Linux documentation team.

<http://www.gentoo.org/doc/en/cvs-tutorial.xml>