Personal Version Control

By Jerry Peek

If you've worked with a group of programmers, you've probably used *version control* software like CVS. However, version control is also useful for non-programmers — for instance, for a group of technical writers producing product documentation.

As we'll see this month, version control can be very handy for a personal project, like tracking changes to a report you're writing or maintaining a history of revisions to your personal *crontab* file. We'll look at handy scripts and tips to make version control easier, and we'll also look at some handy shell tips — like running commands before each prompt — that are useful in a lot of situations, not just with version control.

If you haven't used version control before, read on to get an overview and some pointers! Version control is incredibly helpful for all kinds of computer files... not just programs or documents.

What is Version Control?

Version control (VC) lets you keep a "history" of a file or a group of files. It lets you see, sometime later, who changed what when.

For instance, if a bug suddenly appears in a program, version control lets you inspect the program's source code to discover which change caused the error. You can undo changes — temporarily or permanently — by having the VC software recreate a previous version of some file or files.

When you're working in a group, version control also lets you manage conflicting changes. Those happen when two people make overlapping changes — for instance, deleting part of a file that someone else wants to keep, or editing the same lines of the same file. A VC system can tell you about these problems and, in many cases, help you resolve them.

If you haven't used VC software before, you've probably done something similar by saving backup copies of, say, a report you're writing. You might tell your word-processing software to "Save As" with a filename that includes the date and time of your most recent modifications, such as *report_2004-03-15_1042_bak*.

A more advanced VC system can help you, though, even if you're working on a personal project like that report. VC systems commonly maintain *logs* to record specifics such as the exact date and time each version was saved and a message that describes the changes, such as "Edited the 'Conclusion' to work around concerns about revenue projections." You also don't have to invent new filenames for backups because the VC system keeps the same filename.

This month, that's what we'll focus on: how to use version control for personal projects.

Versions of Version Control

There are lots of version control systems. One of the most-established version control systems is *RCS*, an acronym for *Revision Control System*. It's good for individual file, and also has minimal overhead. For each file you track with RCS, there's a corresponding RCS file that keeps track of all changes, log messages, and so on.

RCS doesn't handle groups of files well, though. It isn't always simple to "take a snapshot" of files at a particular time and then, sometime later, restore all files in a project to the way they were when you took the snapshot. RCS was also designed before everyone had networked computers. For collaborative projects with programmers scattered around a company (or around the world), the lack of network support in RCS can be a pain.

The most popular alternative to RCS is probably *CVS*, the *Concurrent Versions System*. CVS uses RCS as a back-end to manage the actual file storage, but it uses other data files and programs to maintain overall information. It also understands networks: you can have a central repository where all contributors get and save copies of project files. CVS is widely used for Open Source projects. It's too complex, though, to cover thoroughly in a three-page magazine column. For simple projects, RCS is often enough.

CVS also has its limitations. For instance, when you "commit" a group of changes to a CVS repository, it's possible that some archived files will be updated but others won't. (CVS commits aren't always atomic.) This leaves a mess that has to be sorted out by hand.

Subversion is a CVS replacement that's intended to have most CVS features without the CVS limitations. The home page http://subversion.tigris.org gives a nice summary of dif-

HOW VERSIONS ARE STORED

RCS and other VC systems typically don't store complete copies of each version of a file. Instead, they store *the differences* between the previous version and the current one. For plain-text files with widely-spaced changes, this scheme saves a lot of disk space.

Modern VC systems can archive any file, plain-text or not. For instance, you can archive versions of a JPEG photograph or other binary (nontext) file. Be aware, though, that nontext files can take a lot more storage space in the archive. That's because, in many cases, there aren't easy-to-calculate differences between the versions, so the entire file must be archived each time.

ferences between Subversion and CVS. [Look for a feature on Subversion in next month's issue of *Linux Magazine*.]

In this short column, we'll focus on RCS because it's arguably the simplest VC system. If you'd like to use another system, though, it will probably work if it's a well-behaved Linux utility (if it accepts command-line arguments and options, if it reads text from its standard input, and so on).

A Quick Introduction to RCS

Here's a brief overview of using RCS for an individual project (one you *aren't* sharing with other people). For more detail, see your online manual pages or the RCS home page at http://www.cs.purdue.edu/homes/trinkle/RCS.

Each file you want to maintain needs its own RCS archive file. If the current directory has a subdirectory named *RCS* (in uppercase), RCS will put the archive files there. Although you can create a new archive in one step, we recommend these two steps:

1. First, create the RCS archive by typing `rcs -i -U filename`. In that command line, `-i` means "initialize," `-U` means "non-strict locking" (which you should use only on a personal project), and *filename* is the name of the file (called the *working file*) that you'll want RCS to archive. That working file doesn't have to exist yet. When you initialize a file, RCS prompts you for a description of the file. This can be the file's overall purpose or any other "global" information about the file.

For example, let's set up the RCS archive for the file named *food*:

```
$ rcs -i -U food
RCS file: RCS/food,v
enter description, terminated with
single '.' or end-of-file:
>> List of my favorite restaurants.>
>> .
done
```

2. After you've put some content in the working file (*food*), you can *check it in* to the archive file. Checking in makes a "snapshot" of the (current state of the) file and assigns a *revision number*. By default, the first revision gets number `1.1`. A check-in automatically logs the date and time, your username, and more.

To check-in a file, use the RCS command *ci*.

```
$ ci -u food
RCS/food,v <-- food
```

FIGURE ONE: The output of *rlog* shows file history

```
RCS file: food,v
Working file: food
head: 1.2
branch:
locks:
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
List of my favorite restaurants.
----------------------------
revision 1.2
date: 2004/02/13 22:21:06;  author: jpeek;
  state: Exp;  lines: +2 -0
Added downtown restaurants.
----------------------------
revision 1.1
date: 2004/02/13 22:17:32;  author: jpeek;
  state: Exp;
Initial revision
```

```
new revision: 1.2; previous revision: 1.1
enter log message, terminated with
single '.' or end of file:
>> Added downtown restaurants.
>> CTRL-D
done
```

After the initial check-in, *ci* prompts for an optional *log message* that describes the changes you've made since the previous checkin. The `-u` option leaves a working copy of the file after check-in; by default, *ci* deletes the working file.

We're using RCS from a terminal window (on a command line). But you don't have to create your working files from a terminal window. RCS can archive files made with graphical applications like *OpenOffice* or *GIMP*. (Be careful if you use RCS keywords like `$Id $` or `$Date $` in the file. That's because *ci* may update the file's contents — and confuse your other program that already has the file open for editing. We recommend not using keywords unless you test them and understand the effect they have.)

You can do a lot more with RCS, but we'll mention just two basics:

- Use *rlog* to read a file's log: what revisions were checked in, the messages you wrote, and more. For example, to see the changes to *food*, type `rlog food`. Sample output is shown in *Figure One*.

- Use the RCS utility *co* to replace the current working file with a different revision from the archive. (Be sure to check in your working file first, though, if you want to

archive your recent changes!) For example, `co -r1.1 food` would make the working file identical to what it was when you checked in revision `1.1`. You can use the output of *rlog* to identify what revision you want.

You can give *ci* multiple filenames. For instance, to make a snapshot of all the filenames ending with *.txt* in your current directory, type `ci -u *.txt`. RCS will prompt you for a log message for each file unless you use the `-m` option on the command line to specify the message. For example, `ci -u -m"Bug fixes" *.pl` checks in all Perl files with the same message, `Bug fixes`.

If your wildcard (such as `ci -u *`) matches your RCS subdirectory, *ci* will complain. Restricting the wildcard (for example, `ci *.txt` or `ci [a-z]*`) can avoid this.

Archive While You Edit

Many text editors can make automatic backups as you work. Typically, though, these backups only let you recover the last few versions of a file for a short period of time. If you discover that you really need that section you deleted last week, you may be out of luck — unless you used version control to make your backups.

Emacs typically has the command `vc-toggle-read-only` bound to `C-x C-q`. That opens a buffer where you can type a log message. Type `C-c C-c` to finish your input and check the file into RCS.

You also can make an RCS snapshot from *vi* without leaving the editor. First, write out your editor buffer using `:w` or simply set the *autowrite* option, using `:se aw`. Then use the shell escape `:!ci` to run *ci*.

For example, to check in the file you're editing, do:

```
:!ci -u %
RCS/getinfo.pl,v <— getinfo.pl
...ci runs...
done
```

FIGURE THREE: Setting *tcsh* cwdcmd and precmd aliases

```
tcsh% alias cwdcmd 'source ~/lib/tcsh/do_ci'
tcsh% cat ~/lib/tcsh/do_ci
switch ($cwd)
  case /proj/*:
    alias precmd 'ci -u -q -m"by do_ci" rpt*'
    breaksw
  case $HOME/src/perl:
    alias precmd 'ci -u -q *.pl'
    breaksw
  default:
    unalias precmd
endsw
```

FIGURE TWO: Checking differences while editing

```
:!rcsdiff -u %
 ...omitted...
@@ -238,7 +238,7 @@
 sub my_fixname {
 my $from = shift;
 my $to = shift;
- !$ = 0;
+ $! = 0;
 fixname($from, $to);
 return ! $!;
}

Press RETURN or enter command to continue
```

vi replaces the `%` (percent sign) with the name of the current file. When *ci* finishes, press RETURN and you're right back to editing. Even better, the next time you want to check in the file, you only need to type `:!!`, which repeats the previous shell-escape command.

This same technique is great for checking what edits you've made. The *rcsdiff* command, which runs the Linux *diff* utility, shows the changes since the last check-in. *Figure Two* shows (in the *diff* `-u` or *unified* format) that you've changed line 241 since the last check-in.

If the *rcsdiff* output could be long, you can pipe it through a pager program like *less*. Use `:!rcsdiff -u % | less`.

Automatic Archiving

You may not be able to archive a file when you should. For instance, you might want to log the progress of a factory process that runs all night to be able to go back and ask "Where was it at 2:50 a.m?"

If you're using a program or a script to automate a task, it can simply run a command like `ci -q -u filenames` to archive the specified files. The `-q` option tells *ci* not to complain or try to check-in the file if there's no change since the last revision. (On the other hand, the `-f` option forces a check-in; this can be handy when you want to add a message to the log to specifically show that there'd been no change to the working file.) Add the `-m` option (explained above) to specify a log message.

Another way to automate with the Linux *cron* system. Set your *crontab* configuration file to run *ci* periodically.

For example, to go to your *perl* directory at 15 minutes past each hour and archive any changes, try a simple entry like the next one.

```
15 * * * * cd perl && ci -q -u -m"cron
  checkin" *.pl
```

See Power, *pg. 60*

Power, *from pg. 36*

This *cron* job assumes that you run `rcs -i -U` before you add a new Perl script to the directory. If that's not the case, you might want to write a short script, which you run from *crontab*, to handle that.

Speaking of scripts, your *crontab* file can even archive itself! This is handy because it's easy to accidentally delete or overwrite that valuable file. Have a look at the *ci_crontab* script in the February 2003 column, which is online at http://www.linux-mag.com/2003-02/power_02.html.

One caveat: all of these simple systems can have trouble if the working file is being written at the moment *ci* tries to check it in. (The archived version could be incomplete or inconsistent.) You'll need to decide whether that could be a problem. If so, adding a lock file to keep *ci* from running while the working file is changing could do the trick. Often, though, a simple setup like the one we've shown is much more beneficial than the slight risk of problems.

Synchronous Archiving

If archiving your files asynchronously (while you might be modifying them) could cause problems, here's another idea. Have your shell run *ci* just before it prints a prompt. That way — unless something else (like a background program) could be modifying the files — you can be pretty sure that *ci* will run by itself.

In *bash*, for example, you can store the name of a function, or an entire command line in the `PROMPT_COMMAND` variable. In *tcsh*, use the `precmd` alias instead.

Let's see a simple example: we set `precmd` to run *ci* before each prompt, then edit a script with *vi*. Before *tcsh* prints the next prompt, it runs *ci*. *ci* notices the changed script and prompts for a log message:

```
tcsh% alias precmd 'ci -u -q *.pl'
tcsh% vi foo.pl
enter log message, terminated with
 single '.' or end of file:
>> Fixed the bar function.
>> .>
tcsh%
```

You can archive a file while you're editing it by suspending *vi* with CTRL-Z. Your shell will run `precmd` or `PROMPT_COMMAND` before giving you a prompt. Then type `fg` at the prompt to return to editing.

Of course, that setting persists even if you change to another directory. Unless you want to check in every Perl script on your system, you might want to add a little control. For instance, in *tcsh*, set the `cwdcmd` alias to *source* a little script that uses *switch* to set the `precmd` variable only in directories where you want to run *ci*. *Figure Three* (page 36) shows this.

Bash doesn't have a command that's run after *cd*, but you can have PROMPT_COMMAND run a shell function that checks the current directory. Here's a simple function `do_ci()` and the setting of PROMPT_COMMAND that runs it:

```
$ do_ci() {
  case "$PWD" in
>  /proj/*) ci -u -q -m"by do_ci" rpt* ;;
>  $HOME/src/perl) ci -u -q *.pl ;;
>  esac
>  }
$ PROMPT_COMMAND=do_ci
$
```

If the current directory is anything under */proj* (including several directory levels deeper, because this is a pattern match and not a wildcard!), the command `ci -u -q -m"by do_ci"` runs on all files in the current directory whose names start with *rpt*. In the directory *src/perl* under your home directory, it runs `ci -u -q` on all filenames that end with *.pl*.

If this approach bothers you, you can have `PROMPT_COMMAND` or `precmd` run a more intelligent shell function or script that only uses *ci* when it's really needed. Of course, as with any program or automated system, you should think carefully and test thoroughly before depending on a setup like this.

And So On...

Using techniques like the ones we've seen this month (along with a bit of scripting wizardry, maybe) can help you recover your work after a disaster — like accidentally removing files you didn't mean to. Once you start archiving your personal work, you'll probably find a lot of uses for the archives. If you make the archiving almost painless to do, the work you put into it will almost certainly pay off.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

POWER TIP: Quick Backup Copies

To make a copy of the file *report.txt* named (say) *report_bak_040315.txt*, you don't have to type the whole backup filename. Use the shell's curly-brace operators to build the name of the copy:

```
$ cp report{,_bak_040315}.txt
```

How does that work? The curly-brace operators build a series of space-separated strings. For instance, `h{i,ello}` expands into `hi hello`. We're using an empty first replacement string to make our copy, so the command line expands into `cp report.txt report_bak_040315.txt`.