Module 17: Source Code Control System in UNIX

A very large programming project evolves over time. It is also possible that many teams work concurrently on different parts of such a large system. In such a case each team is responsible for a certain part of the project. Their own segments evolve over time. These segments must dovetail with the rest of the project team efforts. So, during the development phase, each development team has to learn to manage changes as they happen. It is quite a common practice to proceed on the assumption of availability of some module at a future time. So, even while designing one's own mandated module, one needs to be able to account for some yet to be completed module(s). One may even have to integrate a newer version of a module with enhanced features. In some sense, a software may have several generations and versions of evolution. In industry parlance, these generations are sometimes called α or β releases. There may even be a version number to suit a certain specific configuration.

Unix supports these developmental possibilities, i.e. supporting creation and invocation of various versions by using a system support tool called SCCS. In this chapter we shall study how SCCS helps in versioning.

17.1 How Does Versioning Help

Essentially, a large software design is like an emerging collage in an art studio with many assistants assisting their master in developing a large mural. However, in the software design, the ability to *undo* has its charms. There are two major ways in which versioning helps. First, we should realize that a rollback may be required at any stage of development. Should it be the case then, it should then be possible for us to get back to an acceptable (and perhaps an operating) version of a file. From this version of file one may fork out to a newer file which is a better version. The forking may be needed to remove some errors which may have manifested or because it was felt necessary to add newer features.

Secondly, software should cater to users of various levels of abilities and workplace environments. For instance, a home version and office version of XP cater to different workplace environments even while offering truly compatible software suites. In other words, sometimes customers need to be provided with options to choose from and tune the software for an optimal usage which may entail making choices for features, adding

some or leaving a few others to leverage the advantage of a configuration with as little overhead as possible.

17.2 The SCCS

During the period of development, there will be modules in various stages of progression. As more modules become *stable*, the system comes along in small increments. Each stable set yields a fairly stable working version of the system. So the support one seeks from an operating system is to be able to *lock* each such version for limited access. In addition, one should be able to *chain* various stages of locked versions in a *hierarchy* to reflect particular software's evolution. Unix obtains this capability by using a version tree support mechanism. The version tree hierarchy is automatically generated. The version number identifies the evolution of different nodes in the version tree over time. A typical version tree numbering scheme is shown in Figure 17.1. The higher the number, the more recent is the version. The numbers appear as an extension to show the evolution. In the next few sections we see what is provided and how it is achieved under Unix.

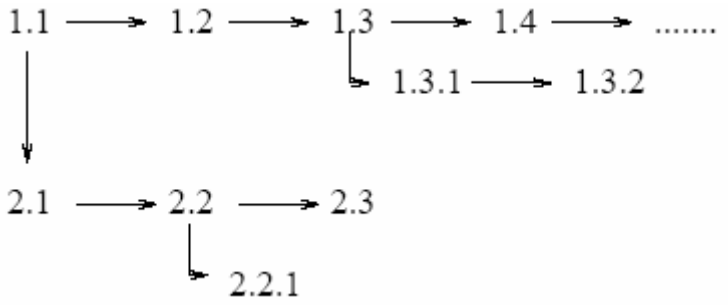

Figure 17.1: A typical version tree under SCCS.

17.2.1 How This Is Achieved

Unix obtains this capability by maintaining a version tree with the following support mechanisms.

- Version tree hierarchy is generated automatically and follows a numbering scheme as shown in figure 17.1. The numbering helps users to identify the evolution of software from the version tree. Internally, it helps Unix in maintaining differences between related versions of files.
- The SCCS creates files in a special format.
- The compilation is handled in a special way.
- Edits to the files are handled in a special way. In fact one can edit files at any level in the version tree.

- A suite of SCCS commands are provided to manage versions and create links in the version trees.

In addition, Unix provides the following facilities:

- Unix permits locking out of modules under development. Some modules which are stable may be permitted full access. However, those versions of modules that are still evolving may not be accessible to all the users. The access to these modules may be restricted to this module's developers only. General users may have access to an older stable version of the module. It is possible to also add gid for access to a new version being released. In Unix the access is permitted for all (or none) in the group. So adding gid for a release adds all the users in that group.
- For management of hierarchy, it is possible to create a tree structure amongst versions. At any stage of development it is possible to create siblings in a tree. Such a sibling can have its own child versions evolving through the sub-trees under it. As shown in Figure 17.1, the version tree grows organically as new versions emerge.
- Unix supports precedence amongst the versions of a module. So if a module has a newer version (usually with some bugs removed or with extensions), then this module shall be the one that shall have precedence during the loading of modules. Often this is true of files where a file may be updated frequently.

Various Unix versions from BSD, System V or Linux provide SCCS-like tools under a variety of utility names. These are CSSC, CVS, RCS, linCVS, etc. We will briefly discuss CVS in Linux environment in Section 17.4. Obviously, more recent versions clearly have more bells and whistles. Now that we have explained the underlying concepts, we shall next explore the command structure in SCCS that makes it all happen.

17.3 SCCS Command Structure

We can appreciate the commands under SCCS better if we first try to understand the design considerations. Below we enumerate some of the points which were borne in mind by the designers when SCCS was created initially.

- Between two revisions a file under SCCS is considered to be idle.
- Between two adjacent versions, both idles, there is only a small change in the content. This change is regarded as *delta*. Suppose for some file *F* there is a delta

change *d*, then one needs to maintain both *F* and d to be able to do the version management. This is because SCCS supports versioning at every node in a version tree.

- Version numbers can be generated automatically. The delta changes finally result in creating a hierarchy. In fact, version numbers determine the traversal required on the version tree to spot a particular version.
- We need a command *use* to indicate which file we retrieve and a command *delta* to indicate the change that needs to be incorporated. That then explains the command structure for SCCS.

Figure 17.2 illustrates the manner in which an SCCS encoded file may be generated.

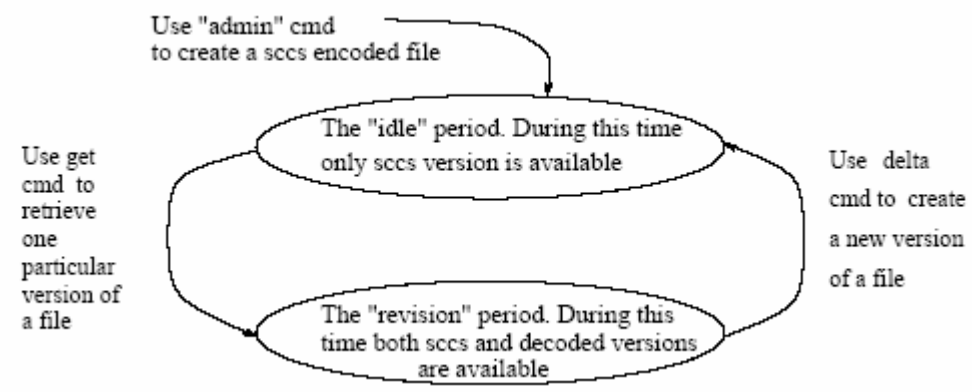

Figure 17.2: The “idle" and “revision" periods in SCCS.

Such files may subsequently go through an evolution with several idle periods in between. When a file is retrieved, it is decoded for normal viewing and presented. Usually, one uses the delta command to create a new version. Based on the version taken up for revision, a new version number is generated automatically.

17.3.1 An Example Session

In this example session we have used some command options. In Table 17.1 we give explanation for options available with various SCCS commands.

To begin with we need to create a new file. We may do so as follows:

admin -n s.testfile

Option	Effect
-n	creates a new SCCS history file
-a	used to add users to a gid, this user can now check deltas
-e	used to erase (when used with admin) a user

Table 17.1: SCCS command options.

This command creates an SCCS formatted file with an empty body. The options used with admin command have interpretations given in table 17.1.

We can now use the visual editor and put some text into it.

get -e s.testfile

The -e option is to invoke ”open for edit". This command prints the version number and number of lines in testfile. If we perform a delta, we can create a newer version of testfile. This can be done simply as follows:

delta s.testfile

We may just add comments or actually make some changes. This can be now checked by using a visual editor again.

Unix provides a facility to view any file with a given version number. For example, to view and run a certain previous version 1.2, we may use the command shown below.

get -p -r1.2 s.testfile

The -p option is to invoke a path and -r is to invoke a run.

Like Make, SCCS also supports some macros. Readers should consult their system's documentation to study the version management in all its facets. We shall next take a brief tour of CVS.

17.4 CVS : Concurrent Versioning System

Linux environment offers a concurrent versioning utility, called CVS which supports on-current development of a project. The team members in the project team may make concurrent updates as the project evolves. The primary idea in CVS revolves around maintaining a project repository which always reflects the current official status of the project. CVS allows project developers to work on parts of the project by making copies of the project in their own scratch pad areas. Updates on the work-in-progress scratch pads do not affect the repository. Individuals may seek to finalize their work on the scratch pad and write back to the repository to update a certain aspect of the project. Technically, it is quite possible for more than one individual to make copies and develop the same program. However, this can raise consistency problems. So writing back into the repository is done in a controlled way. If the updates are in different parts, these are carried out with no conflict. If the updates are in the same part of a file, a merge conflict may occur and these are reported. These conflicts need to be reconciled before a commit into the repository is performed.

Essentially, CVS is a command line utility in Linux (and also in some other flavors of Unix). There are network versions of CVS that allow access to repository over the network as well. For now we shall assume that we have a CVS utility available on a single machine with multiple users accessing the files in it. Let us examine some typical usage scenarios of CVS commands. For example, consider the command lines below.

export CVSROOT=/homes/iws/damu/cvs

cvs checkout project-5

The first command line is to look for the cvs command in user damu's home directory. The second line basically checks out what files are there in project-5. It also makes the copies of project-5 files available to the user. Typically, these may be some .c or .h files or some others like .tex files. As we stated earlier, CVS supports an update phase of operation. The update command and a typical response from it are shown next.

cvs update project-5

When one attempts to update, typically the system prompts to indicate if someone else also made any updates after you had copied the files. In other words, all the updates that occurred in sequence from different people are all shown in order. Each update obtains a distinct version number. Like SCCS, CVS also generates internally a numbering scheme which will give a versioning tree. In the case someone else's' update is at the same location in a file, then the messages would indicate if there is a merge conflict. A response to a typical update command is shown below:

$ cvs update

cvs update: Updating .

RCS file: /homes/iws/damu/cvs/project-5/main.c,v

retrieving revision 1.1

retrieving revision 1.2

Merging differences between 1.1 and 1.2 into proj.h

M proj.h

U main.c

In the example above we had no merge conflicts. In case of a merge conflict, we get messages to indicate that. Let us assume we have merge conflict in updating main.c, then the messages we may get would look like those shown below.

$ cvs update

cvs update: Updating .

RCS file: /homes/iws/damu/cvs/project-5/main.c,v

retrieving revision 1.2

retrieving revision 1.3

Merging differences between 1.2 and 1.3 into main.c

rcsmerge: warning: conflicts during merge

cvs update: conflicts found in main.c

C main.c

Usually, the conflicts are shown with some repeated character sequence to help identify where it occurred. One, therefore, needs to resolve the conflicts and then may be commit the correctly updated version to repository. A CVS *commit* generates a newer version in repository. A CVS commit command and its response is shown below.

$ cvs commit

cvs commit: Examining .

Checking in main.c;

/homes/iws/damu/cvs/project-5/main.c,v <-- main.c

new revision: 1.2; previous revision: 1.1

done

The steps shown above for checkout, update and commit are the basic steps. There are means available to *import* an entire project and effect an update to create a newer version of the project. For those details the reader is advised to refer to the man pages.