

Lecture 4

Version Control

It is usually handy to have old versions of the source code stored somewhere. It becomes more important when there are multiple developers.

We discuss the basics of version control, and a particular implementation called CVS (Concurrent Versions System).

References:

- Info pages of (**cvs**).
In the "Overview" node you can find a "sample session" which summarizes the most frequent commands, which you might find helpful.

PMOOP(0396A)

PMOOP(0396A)-4.1

Why old versions

- Suppose, after you write a program, you find something you want to improve on, and thus updated the code.
- Is the old code now becoming useless? Probably not.
- Problem is, **your modification probably create some bug**.
- An **old version** can usually **assist the debug process**.
If you have two versions one having a bug and another don't, you just need to check those parts changed by between them.
- So when should we discard our old version? It is difficult to decide.
A bug might be there for a long time before being discovered.
- If old version is good to keep, **why don't just keep a lot of copies?**
- This **waste a lot of space**.

Multiple file complications

- Thus the basic idea of version control is to **take snapshot of source file** during the development process.
- Once your project consists of multiple files, you **don't really need to keep all the files for each version**.
This will save a real lot of space. Remember that space saving means the possibility to keep longer and more frequent history.
- One possibility: **take more snapshots** for files that changes **frequently**, and take less snapshots for files that are mostly stable.
- Then **each file has independent history**: we don't take snapshot for the whole project, but instead just for each of the files.
- Now we have another problem to solve: **how do we make up the whole picture from the snapshots of these files?**
That is, how to pick the right snapshot for each file to make up the whole project?

PMOOP(0396A)-4.2

Multiple developer complications

- The version control problem gets **more serious** when our program is written by **multiple programmers**.
- Q1: should all programmers work on the **same set of files?**
- A1: **No!** When two programmers are simultaneously working on two different files, there is virtually *no time when either can compile!*
- Now we have yet another problem: **how the two developers should combine their changes?**
- Old snapshot helps by letting us know how which files are **modified by each developer**.
- There is one more problem: **what should happen if a file is modified by both developers?** (Should be rare, but it does happen.)
Now you should understand why we need some automation here. It is simply too time consuming to deal with all these complications manually.

PMOOP(0396A)-4.3

Version control system

We solve all these problems by using a **version control system**, which:

- **Automates "taking a snapshot"** of the source files of a program.
So after you started working with a version control system, you just need to give a simple command to "keep a version".
- **Stores all our snapshots** in a way that **preserve memory**.
i.e., store only the changes made, not all the versions.
- **Determine the difference** between a file and any previous snapshot.
- **Find the snapshots of each file** correspond to a particular time.
- **Combine the changes made by other programmers** into your own working copy of the program source.

All version control systems work similarly. We will learn one called **CVS**.
In free software community, CVS takes basically all the market share.

PMOOP(0396A)-4.4

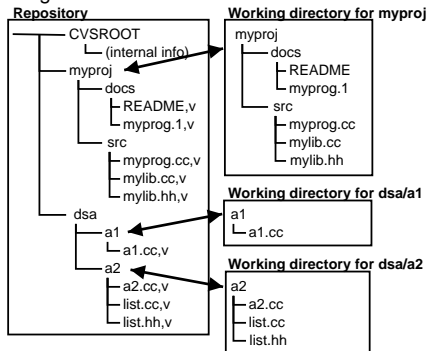
Basic concept 1: Repositories

- A repository is a **place where CVS stores all snapshots of all files of all projects it manages**.
- It acts like a **mini-filesystem**. That is, the repository is a **directory containing other sub-directories and files**.
- Usually, **each programming project take one sub-directory** and all its descendents of the repository.
So one repository usually serves many projects.
- **Each file** of the repository **records all snapshots** of a file of some project, in a space-preserving way.
It stores the latest snapshot, and "diffs"—instructions to modify the file to each previous version. This works only for text, so don't store non-text file into it.
To remove all records of a file, just delete the corresponding file in repository.
- The **directory structure** within the sub-directory of a project **mirrors that of the project**.

PMOOP(0396A)-4.5

Files inside and outside a repository

A possible arrangement:



Projects not in the top-most directory of a repository should have a "module" defined. Don't bother about it until you are familiar with CVS.

PMOOP(0396A)-4.6

Creating a repository

- The next question is: **where is the repository?**
- The repository **can be placed at any directory** that you can write to. Likewise, **you can name it however you like.**
- To create a CVS directory, use the `cvs init` command.
- To tell CVS where you want to create the directory, use the `-d` flag. You may also use an environment variable called `CVSROOT` discussed later.
- Example: `cvs -d ~/cvsroot init`
Of course, you won't need to create the same repository twice, so this needs to be done only once.
- The repository will then contain a directory called `CVSROOT` to **contain options and logging files for CVS.**
But at this point, you can safely ignore the configuration files there until you really need to deal with them.

PMOOP(0396A)-4.7

Starting a new project using the repository

- After you `init` a new repository, it contains nothing except configuration files. So the next affair is to create sub-directories there to hold new projects.
- This is in fact really simple: you just need to **go to the repository and create whatever directory structure** you want there.
- E.g., we can `cd ~/cvsroot` and `mkdir myproj`.
- **Question:** will creating directory interfere with the history records of CVS in the repository?
- **Answer:** no, because CVS never remember the history of directories. It only remember the history (i.e., snapshots) of files.
- So you can directly create directories in repositories, **but never directly create files in the repository!**

PMOOP(0396A)-4.8

Creating Working Directories

- If we cannot touch the files in the repository, **how can we create and modify files there?**
- The answer is, we create, or **check-out, working directories** from the repository, and work on files in the working directory. So "check-out" really means "create a working directory".
- The CVS command `co` let us do that. You'll need to tell CVS **where the repository is**, and within it **which sub-directory** you want to check out.
- Example: `cvs -d ~/cvsroot co myproj`
Order is important: `-d` must come before `co`.
- This creates a new directory "myproj", containing a subdirectory `CVS`.
- That directory contains the **current status** of all files in this directory, the location of the repository and the sub-directory. I.e., the parameters when we check-out. We don't need to specify them again.

PMOOP(0396A)-4.9

Adding new files

- You can do whatever edit you want in the working directory: **modify** files, **create** files, **remove** files, or even **create** subdirectories.
From now on we assume you're in the working directory.
- The edits will go into the repository when you **check-in** (next slide).
- Normally **CVS** will **ignore** all the new files you create, and **re-create** all the old files disappearing.
- This is to make sure that file will not easily get lost, and those generated or testing files will not easily get into the repository.
You need snapshots only for sources, not for generated objects or executables.
- So after you create a file or directory, **use `cvs add` to prepare for adding** it to the repository. After you remove a file, **use `cvs remove` to prepare for removal.**
"Prepare for" adding and removal, since you still need to perform a check-in before the changes actually take place.

PMOOP(0396A)-4.10

Checking in

- Once you are ready to make your modifications back to the repository, you can **check-in your changes** using the command `cvs ci`.
But check-in will not delete the repository, so it is not the exact opposite of check-out. Later we will see that check-in is the direct opposite of update.
- This is the **most frequent command** that you should be using, since you have to do this whenever you modify something.
That's why the command is made so short. Note that we don't need to tell which repository to update: it is remembered in the CVS directory.
- Once you type the command, a **default editor** will be invoked, asking you to type in some "log-string".
If you write a good log string, it will be very useful if you need to find a correct version of your file out of 100 snapshots.
- After you save it, the check-in proceeds to take a snapshot of **all modified files in the directory.**
That is, unchanged files will have no snapshot taken, saving space. In fact, you can ask CVS to just checkin one or two files instead of the whole directory.

PMOOP(0396A)-4.11

Importing an existing project to the repository

What if we already **have a project before using CVS**? Instead of adding all files into it manually using `cvs add`, we can **import** it to CVS.

- First, change to the directory storing your project.
- Now you can import all files to the repository:

```
cvs -d ~/cvsroot import myproj imported initial
```

"imported" and "initial" are called the vendor and release tag respectively. Any two names will do. You probably won't need to know the exact meaning.
- After CVS asks for log messages with `vi`, all files are imported to the `myproj` directory of the repository.
- At this time, the directory is **not** managed by CVS, so you should then immediately perform a checkout.

Actually, you can follow these **even if you are creating a new project**. Just create the empty directory structure and import it to the repository.

PMOOP(0396A)-4.12

Environment variables

- You probably start to get annoyed: everytime we check-out or import, we have to add `-d ~/cvsroot` to tell CVS where is the repository. Unless you use many different repository, which makes this necessary.
- Solution: you can set an **environment variable** called `CVSROOT` to tell it where it is. In Unix, the environment variables are passed as the third argument to the main function of all invoked programs as an array of `char *`.
- How to do this depends on what shell you're using. If you use `bash`, add `export CVSROOT=~/cvsroot` to the `.bashrc` file. This export has nothing to do with the CVS command `import`!
- If you use `tcsh`, add `setenv CVSROOT ~/cvsroot` to the `.cshrc` file.
- There is another environment variable that you might want to set: **CVSEDITOR**. This set the editor used to edit the log message. E.g.:

```
export CVSEDITOR=pico
```

PMOOP(0396A)-4.13

Viewing history

- Now comes the big question: **what's the use of the snapshots**?
- One possibility: **read the log messages** you've made in a file.

```
cvs log myproj.cc
```
- Another possibility: **read the source code annotated with the revision number** when the line first appear in the file:

```
cvs annotate myproj.cc
```

You probably want to do this in a wide terminal so that the lines still "lines-up".
- But what are those "revision numbers"? Each time you check-in a file and create a snapshot for it, a new revision number is assigned to it.
- It is automatic, so we don't need to bother about it. If suddenly you want to bump up the number, e.g. to 2.0, you can do it like this:

```
cvs ci -r 2.0
```

PMOOP(0396A)-4.14

Going back to an old version

- Another thing that we usually want to do: **get the project state at a previous date and time**.
- How to do this? **If you know the revision number** of a file, you can go back to it using the CVS **update** command.

```
cvs update myprog.cc -r 1.3
```
- What if I want the previous state of the whole project, not just a file?
- Problem is, **revision number would be different for different files**. Remember that if a file is not modified, a snapshot is not taken, so a new revision number will not be assigned to the file.
- If you know the date and time, you can something like this:

```
cvs update -D "2001-12-30 20:05"
```
- But **how I can remember all the important dates?**!

PMOOP(0396A)-4.15

Tagging

- Solution: we **assign symbolic tags to important snapshots** of files. E.g., when a new version of the project is released, or just before you modify a big data-structure, etc.
- We can assign tag to individual files, but usually we **tag all files of a project** at the same time (so the tag is the symbolic name of the time).
- E.g., we might **make a tag like "rel-2-0"** this when we release version 2.0 of our project. To do this, we issue the following after check-in:

```
cvs tag rel-2-0
```

A tag can contain alpha-numeric characters, hyphens and underscores.
- To **get back to a particular release**, you can use `update` with `-r`. E.g.,

```
cvs update -r rel-2-0
```
- You can also **create a new working directory** to contain the release:

```
cvs co -r rel-2-0 myproj
```

PMOOP(0396A)-4.16

Branching

Usually, after a release is made, and after new code has been checked in, the **old release need modifications**, e.g., to fix a serious bug.

The relationship between snapshots is then **tree-like**, not list-like:



It is not possible to insert a snapshot between two: it would be changing history.

The main development line is called "main trunk", and other lines are called "branches". E.g., revisions 1.2.2.1–1.2.2.3 form branch 1.2.2.

Again, revision numbers of branches are different for different files. So we **usually use tags instead of the revision numbers of branches**.

Actually, the vendor tag is a branch tag.

PMOOP(0396A)-4.17

Using branches

Example: suppose, after we release versions 1.0 with tag rel-1-0 and continued the development towards 2.0, we want to fix a bug in 1.0.

First check-out version 1.0:

```
cvs co -r rel-1-0
```

Next, cd into it, and create a branch from it:

```
cvs tag -b rel-1-0-patches
```

Then we can **switch to the newly created branch**:

```
cvs update -r rel-1-0-patches
```

Now fix the bug and check-in. The snapshot will be attached to the branch rel-1-0-patches. The update command records a "sticky tag" to make this possible. This is revealed by the status command:

```
cvs status
```

To switch back to the main trunk, do the following:

```
cvs update -A
```

PMOOP(0396A)-4.18

Files to ignore

- We usually have **some files in the working directory that we don't want to go into the repository**. For example, backup files and compiled files normally shouldn't be in the repository.
- They pose no problem, except that **cvs update** gives an warning.
- To avoid this, **list the patterns of such files in a file .cvsignore** in the working directory, and add that file to the repository. For example, .cvsignore might contain this:

```
*~  
*.o  
a.out
```

PMOOP(0396A)-4.19

Keywords

Sometimes we want the **revision information of files**, e.g., the revision number, to **appear directly in the file**.

CVS has a **keyword substitution mechanism** that automatically insert such information into the source files.

For example, we can add the following comment to the C++ source:

```
// $Id$
```

When the file is checked out, we get something like this:

```
// $Id: hello.cc,v 1.5 2001/2/19 14:57:32 ktko Exp $
```

This gives **easy access to the revision number** of the file, which can be extracted by the `ident` program, E.g., `ident myprog.cc`.

We can even embed the file into the compiled executable: read the manual.

PMOOP(0396A)-4.20

Multiple developers

What to do if there are **multiple people working on the code**?

- All developers must have **write access** to the directories of the repository, and **read access** to the files there.
It is probably a good idea to setting up a Unix group that contain just the developers for that project. All directories should be owned by that group.
- Each developer should **check-out his own working directory** and work there. This way temporary work of one developer will not affect others.
- The developer would check-in **only when he think that his work is okay for other developers to use**.
E.g., compile correctly, won't core-dump immediately, etc.
- Once he do that, he tell others (by other means, e.g., E-mail, IRC, ...). The other developers will **merge** the changes in the repository by

```
cvs update
```

That's why the command is called update.

PMOOP(0396A)-4.21

File locking

- Should **two developers** be **modifying the same file**?
- Usually not: it would be **difficult to merge changes** this way. If the source file is split according to division of labour, this shouldn't happen.
- Thus many version control system mandates that a file must be **locked before modified**, thus **forbidding conflicts**.
- But **sometimes concurrent edits are needed**, e.g., many developers might want to modify the Makefile at the same time.
- Then the system becomes very **counter-productive**:
 - A developer want to modify the Makefile (e.g., add a new source file), so he check-out the file and lock it. But he **can't check-in yet**, since the new source file is not ready for others to consume.
 - Another developer now want to do the same, but he can't check-out: **the file is locked by somebody else**. What he can do now?

PMOOP(0396A)-4.22

Unreserved checkout: doing without lock

- CVS use a scheme called **unreserved check-out**, i.e., no lock is needed.
- If two people are editing the same file, one will check-in the file too late.
- That one will **not be allowed to check-in** until he updates the source file to the version of the repository.
- If that happens, CVS will try its best to **merge the changes** made by in the repository to the working directory
Usually works, since changes are likely to be in completely different places.
- If that fails, **CVS marks the conflict in the source file**, and ask the developer to deal with the conflict manually.
- Once the update completes, he check-in his changes to the repository.

This normally works well, since the **developers should be working primarily on different files**, so conflicts are infrequent.

PMOOP(0396A)-4.23

What to read in CVS manual?

- Overview.
- Repository: Specifying, Storage (file permission), Creating.
- Starting a new project.
- Revision: revision numbers, versions, assigning, tags, tagging.
- Branching: motivation, creating, accessing, revisions.
- Recursive behaviour.
- Adding and removing
- History browsing: log, annotate.
- Multiple developers: Filestatus, updating, conflict, informing, concurrency, model.
- Revision management, Keyword substitution.

And anything in the manual that interests you.