REVISION CONTROL WITH RCS

Once a programming project gets really large, we can expect different people to be working on different parts of it simultaneously. What happens if two people decide to work on the same class implementation (.cxx) file? Or if one person modifies the class definition (.h) file without telling someone else who is working on the class implementation (.cxx) file? Even if each file is modified correctly, differences may introduce compilation errors or misbehaviors throughout the entire system. Disastrous problems can result from failing to coordinate changes to the project in some organized way.

Version control systems manage access to files to reduce problems with multiple users who work on a common set of files. In this lab, we'll be playing with the unix rcs (revision control system) commands.

Begin by creating a lab directory in your 2270 directory:

```
mkdir ~/csci2270/lab10
cd ~/csci2270/lab10
```

Copy my node1.h file to this directory.

```
cp ~/ekwhite/2270SP04/lab5/node1.h .
```

Create a directory in your working directory for the RCS information:

```
mkdir RCS
```

Set the permissions for the files as loosely as possible:

```
chmod a+rw node1.h
ls –lta node1.h          // note the permissions: -rwxrw-rw-
chmod a+rw node1.cxx
ls –lta node1.cxx
```

The ls command output should show that anyone can write to this file. But now let's check it into the rcs system, and unlock it with the –l switch so anyone can take it out:

```
ci –u node1.h
```

You'll be asked to write a short comment; terminate the comment by typing a single '.' and hitting Enter. The system now assigns version 1.1 to this file. See what's happened to your file permissions:

```
ls –lta node1.h          // -r-xr--r--
```

If you check it out again, the permissions loosen to allow you to write to the file:

```
co –l node1.h
ls –lta node1.h          // -rwxr--r--
```

Open the node1.h file in emacs and add a comment; save the changes. Now check it back in, adding another comment, and then note the different permissions:

```
ci –u node1.h
ls –lta node1.h          // -r-xr--r--
```

So what happens if you don't pay attention to this? Try opening the file node1.h without checking it out first. What happens if you try to use emacs to change it?

The rcs command lets you override the locking mechanism without using ci or co:

```
rcs –l node1.h            // locks the file
```

This will work unless someone else has checked it out and locked it in the meantime; if that happens, you'll need to negotiate with that person.

If one user checks out and locks the file, then other users can still check the locked file out to read or compile the code, but they can't make changes.

If you check a file out and back in without modifying it, RCS will detect this and not update its version. You can also check in a version without giving up the lock on it; for example, if you want to keep editing the file:

```
ci –l nodel.h
```

To permit different people, like the people in your groups, to use RCS to manage code files, you'll probably need to disable "strict checking" in the RCS system for the files you want to share in a group.

```
rcs –U node1.h
```

If you look inside the RCS subdirectory, you'll find a version file (extension is ",v")for every file you ever checked into the RCS system:

```
emacs RCS/node1.h,v &
```

You can also get this information from the rlog command:

```
rlog node1.h
```

As an experiment, pair up with a friend and see if you can check their files out and back in. You'll probably need to finesse the permissions to make sure that the files permit others to have access to them.

<u>The underlying principle of RCS</u>

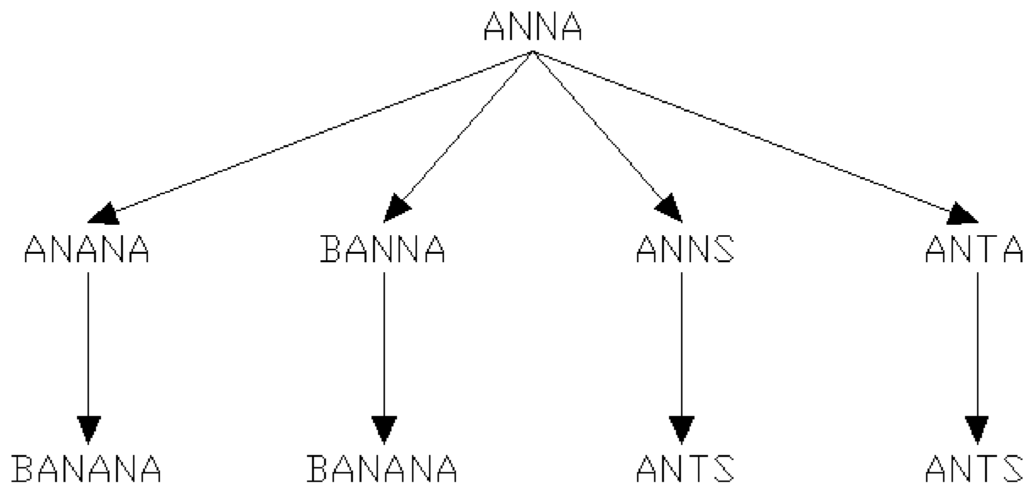
In the RCS system, changes to different versions are represented as a tree, which is useful for conserving space. We can see the basic idea of this with a string example.

The string "ANNA" can be altered to "BANANA" in two insertion steps, an insertion of A in the middle of the string and an insertion of B at the start of the string (and these can happen in either order).
ANNA → ANANA → BANANA
ANNA → BANNA → BANANA

The string "ANNA" can also be altered to "ANTS" in two substitution steps, changing the third letter from N to T and changing the fourth letter from A to S.
ANNA → ANTA → ANTS
ANNA → ANNT → ANTS

Other substitutions, insertions, and deletions are also possible, and can generate an infinite number of strings.

If we map these transformations of ANNA into a tree, we can trace down from the original string at the root, through a series of changes, until we find a final string at one of the leaves. Notice that representing the changes this way (as a tree, not a graph) means that the two final strings ANTS are distinct "versions", and so are the two final strings BANANA.




The tree representation works on the assumption that most of the time, changes you make to the file are small relative to the whole size of the file. If you add a new B-tree routine to a B-tree class, the RCS system can store the original class and record the new txt you've added as a change to the original file; it doesn't need to store parts of the file that haven't changed. This lets the RCS system keep track of your changes without wasting much space on redundant data. It's as if we kept track of "BANANA" by saying "take the original string ANNA, and add an A in the middle and a B at the beginning." For really long strings, like files, this approach actually makes some sense.