CSE 374
Programming Concepts & Tools

Hal Perkins

Winter 2011

Lecture 16 – Version control and svn

Where we are

- Learning tools and concepts relevant to multi-file, multi-person, multi-platform, multi-month projects.
- Today: Managing source code
  - Reliable backup of hard-to-replace information (i.e., sources)
  - Tools for managing concurrent and potentially conflicting changes from multiple people
  - Ability to retrieve previous versions
- Note: None of this has anything to do with code. Like make, version-control systems are typically not language-specific.
  - Many people use version control systems for everything they do (code, papers, slides, letters, drawings, pictures, . . .)
    - Traditional systems are best at text files (comparing differences, etc.); newer ones are better with others too.

Version-control systems

- There are plenty: scss (historical), rcs (mostly historical), cvs (built on top of rcs), subversion, git (much more distributed), mercurial, sourcesafe, …
- The terminology and commands aren't particularly standard, but once you know one, the others aren't difficult – the basic concepts are the same
- cvs had the biggest mind-share for about a decade (particularly in the open-source community)
- svn improves on several cvs shortcomings and is widely used – we'll learn basic svn
- git and mercurial are the hot new thing – distributed version control – but core ideas are the same

The setup

- There is a svn *repository*, where files (and past versions) are reliably stored.
  - Hopefully the repository files are backed up, but that's not svn's problem.
- You do not edit files in the repository directly. Instead:
  - You check-out a *working copy* and edit it.
  - You commit changes back to the repository.
- You use the svn program to perform any operations that need the repository.
- One repository may hold many projects. A subversion repository is just a database of projects and files.
  - Looks like a filesystem tree of project directories

Tasks

Learn the common cases; look up the uncommon ones.

In a production shop…

- Create
  - a repository (rare – every few years)
  - a new project (infrequent – once or twice a year)
  - a working copy of a project (every few weeks or months?)
- Working with files
  - Get updates, add or remove files, commit changes to repository (daily)
  - Check version history, differences (as needed)
- Branches, locks, watches, others (every now and then)

Basic command structure is the same for all

svn *svn-options* cmd *cmd-options* *files…*

Repository access

A repository can be:

- Local: specify repository directory root via a regular file path name url
- Remote: specify user-id and machine
  - Must have svn and ssh installed locally
  - Need authentication (ssh password or other)
- Suggestion: experiment on a local machine
- Next homework project will use remote access to a server

Getting started

- Set up a repository (your choice of name, location; we'll do this for you on hw6)

  svnadmin create path/svnrepos
- Put initial version of project directory in repository

  svn import projdir svn://path/svnrepos/proj -m *msg*
  - Commands that update a repository require a message (msg) that should briefly document the change
  - Once a project is imported, ***never*** use the original directory again (never! We really mean that!)
  - Path depends on kind of access (local/remote)
- Check out a copy of the project to a *working directory*

  cd working-directory

  svn checkout svn://path/svnrepos/proj proj
  - Working directory remembers repository location for future checkin, update, etc.
- HW6: path to repository server is different – see writeup

File manipulation

- Add files with svn add
- Bring local working copy up to date with svn update (get changed files from repository)
- Commit local changes with svn commit
  - Any number of files including subdirectories recursively if no filename specified
  - Files not actually added to repository until here
- Commit messages are mandatory
  - -m “short message”
  - -F filename-containing-message
  - Else pop up editor if EDITOR or VISUAL environment variable is set
  - Else complain

Some examples

- Update local working directory to match repository

  svn update
- Make changes (use svn instead of local file commands so repository will also change on commit)

  svn add file.c

  svn move oldfile.c newfile.c

  svn delete obsoletefile
- Commit changes

  svn commit -m ‘‘this is much better’’
- Examine your changes

  svn status

  svn diff file.c

  svn revert file.c

Conflicts

- This all works great if there is one working-copy. With multiple working-copies there can be conflicts:
  1. Your working-copy checks out version 17 of foo
  2. You edit foo
  3. Somebody else commits a new version (18) of foo
- Subversion tries to merge changes automatically; if it can't you must resolve the conflict. If svn commit fails:
  - Do svn update to get repository version and attempt merge
    - "G" means the automatic merge succeeded
    - "C" means you have to resolve the conflict
  - Merging is line-based, which is why svn is better for text files
  - Conflicts indicated in the working-copy file (search for <<<<<<)
  - Newer versions of svn handle more of this automatically or interactively

svn gotchas

- Do not forget to add files or your group members will be very unhappy.
- Keep in the repository *exactly* (and *only*) what you need to build the application!
  - Yes: foo.c foo.h Makefile
  - No: foo.o a.out
  - You don’t want versions of .o files:
    - Replaceable things have no value
    - They change a lot when .c files change a little
    - Developers on other machines can’t use them

Summary

- Another tool for letting the computer do what it's good at:
  - Much better than manually emailing files, adding dates to filenames, etc.
  - Managing versions, storing the differences
  - Keeping source-code safe.
  - Preventing concurrent access, detecting conflicts.