Recitation 1: Intro & Revision Control

Dave Andersen
CMU Computer Science
Fall 2006

Systems Software

- Low-level (projects in C)
- Designed to run forever
  - Handle every possible error condition
  - Manage resources carefully
- Must be secure!
  - The Internet is not a friendly place
- Based on documented protocols

Quite different from 213

- Project size: 1000s of lines vs 100s.
- Project duration: 5 weeks vs 1-2 weeks
- Partners vs. solo developer
- Result:
  - You can't keep the "state" for 441 projects in your head at all times. It's too big!
  - Requires one step more care in development.

Software Engineering For Systems Hackers

- Goals and foundations:
  - 1-5 developers
  - Context: Systems code, but not too much is systems specific
  - Very low overhead and up-front cost
    - Like to see benefits w/in a single project

Our Philosophy

- Your time is valuable and limited
- Some things are fun:
  - Design, initial coding, a working project
- Some things are less fun:
  - Agonizing debugging, bad project grades, spending 10x longer than you thought
- Use techniques that minimize time and maximize fun vs less fun.

Partly-free lunch

- Techniques take a bit of time to learn
  - E.g., revision control software (today)
  - But they *will* pay off!
- Some techniques take a bit more up-front time
  - E.g., writing good log messages, thinking about design, good debugging capabilities
  - But they make the rest of the project *more predictable* and reduce the uncertainty of failing in the last day.
  - (And they save debugging time).

Your job

- Ask yourself: "Could I be doing this in a more efficient way?"
  - Typing "gcc –g –Wall foo.c bar.c baz.c" vs typing "make"
- Debugging: "Have I seen this bug before? What caused it? How could I avoid it?"
  - Be reflective; strive to learn & improve.

In Practice: Algorithms

- Most systems programs need:
  - Hashes, linked lists
  - Searching and sorting
- For many, that's it.
  - (Databases are different)
- Given this,
  - What would a good, lazy programmer do?

Don't write it twice

- Hashes/lists: Have a nice implementation that you reuse.
  - We suggest either the ones from
    - "The Practice of Programming"
    - Or rip them out of the BSD kernel
    - This is perfectly acceptable in 441
- Sorting: Don't write at all!
  - C library "qsort" (heap, merge…)

Don't prematurely optimize

- If it ain't slow, don't break it
- Keep your programs *simple*
  - Easier to write
  - Easier to debug
- But make it easy to change implementation details
  - Modularity! (Later lecture)

Optimizing your time

- Sorting 3 numbers: Do it by hand
- Sorting 3000 numbers: Do it in ruby
- Sorting 300,000,000,000 numbers: Write some serious code
- Mental calculation
  - Time spent doing task
  - Time spent automating/optimizing
  - Will you have to do this again?

Overview

- Today: Intro & Revision Control
  - Managing your source code wisely
- Makefiles and automation 1
  - Automate the boring stuff!
- Design: Modularity and Testability
  - Managing 1000 LoC != 100 LoC
- Debugging: Techniques & Tools
- Automation 2: Scripting

Resources

- Some great books:
  - The Pragmatic Programmer
  - The Practice of Programming
  - Writing Solid Code
- Recitation notes:
  - http://www.cs.cmu.edu/~dga/systems-se.pdf
  - Please don't redistribute: They're very preliminary!

Recitation Mechanics

- 1) These are *your* recitations.
  - We've got a schedule. It's flexible.
  - Ask questions, make comments, …
  - 1 part lecture, 1 part "public office hours" (homework questions? Sure! Project questions? Great!)
- 2) These aren't the final answers
  - Recitations culled from our experience, other faculty, friends in industry, books, etc.
  - We're always looking for better ideas/tools/practices/etc. If you have some, please share.

Revision Control

- Before you write a line of code…
- Use subversion/CVS/etc.
- Provides access to all old versions of your code
  - No more "cp file.c file.c.2006-01-01-1059am-oh-god-please-let-this-work"

What is revision control?

- A repository that stores each version
- You explicitly "check out" and "check in" code and changes.
- ```
  597 bark:~/tmp> svn checkout
  https://moo.cmcl.cs.cmu.edu/svn/systems-se
  A    systems-se/related.tex
  A    systems-se/acks.tex
  A    systems-se/tinylang.tex
  A    systems-se/emacs.tex
  ….
  ```

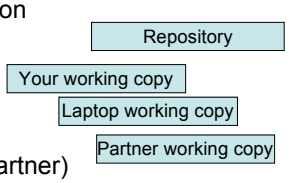
Why do I want it?

- Super-undo: Go to arbitrary versions
  - rm –rf your source tree? No problem!
- Tracking changes /"why did this break?"
- Concurrent development
- Snapshots
  - Turning in the assignment: just make a snapshot when you want, and we'll grade that. You can keep developing afterwords.
  - Useful, e.g., for optimization contest, or for making sure you have *something* working.

You've sold me. What should I know about it?
  ```

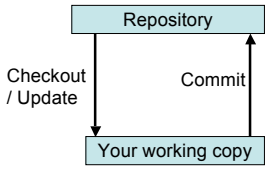
The repository

- Master copy of code is separate from what you work on
- You can have multiple working copies checked out. (So can your partner)

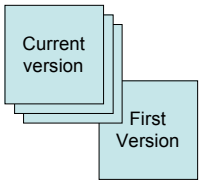


Check out and commit

- Explicitly synchronize with the repository



Every revision is available



And you can see what changed

Revision control lets you note (and then see) what you changed:

```
> svn log gtcd.cc
r986 | ntolia | 2006-08-01 17:13:38 -0400 (Tue, 01 Aug 2006) | 6 lines

This allows the sp to get rid of chunks early before a transfer is
   complete.
Useful when a file is requested in-order and the file size > mem cache
   size
```

And makes it easy to go back to other versions:

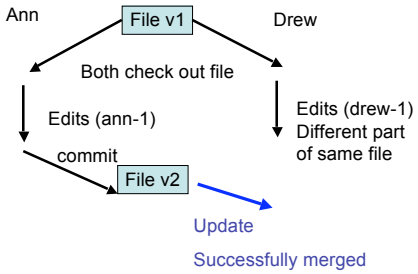
```
------------------------------------------------------------------------
r987 | ntolia | 2006-08-02 13:16:21 -0400 (Wed, 02 Aug 2006) | 1 line

After much thought, I am reverting the last patch. We will need to
   revisit the
issue when we think about DOT on storage-limited clients
```

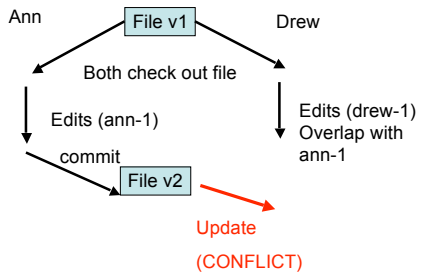
Concurrent Development

- Each person checks out a copy
- Both can work at the same time without (much) fear of clobbering the other
  - Changes only visible on commit/update
- What if both people edit the same file and commit it?

Concurrent edits



Concurrent edits



Resolving Conflicts

- Subversion will give you 3 files:
  - The original with conflict markers
  - The version you were editing
  - The latest version in the repository
- You can:
  - Keep your changes, discarding others
  - Toss your changes
  - Manually resolve

Branches

- Multiple paths of development, e.g.
  - Release 1.0 only gets security patches
  - "Development" branch gets everything
- "tags" or "snapshots"
  - Save a good known state. E.g., for handing in.
- Issue of merging (read on your own)

Subversion (see handout)

- `svn checkout https://moo.cmcl.cs.cmu.edu/441/..`
- `svn commit`
- `svn update`

- `svn add`
- `svn mkdir`

- `svn copy: create a branch or snapshot`
- `svn diff: See difference between versions by default: between what you started on and where you are now`

Brief walkthrough

```
> svn checkout https://moo.cmcl.cs.cmu.edu/svn/systems-
  se
    A    systems-se/acks.tex
    …
> cd systems-se
> echo "new file" >> test.txt
> svn add test.txt
  A         test.txt
> svn commit
[svn will open an editor for log message]
Adding          test.txt
Transmitting file data ..
Committed revision 21.
```

Thoughts on Revision Control

- Update, make, test, *then* commit
- Update out of habit before you start editing
- Merge often
- Commit format changes separately
- Check *svn diff* before committing
- Try not to break the checked in copy
  - Invasive changes? Maybe a branch
- Don't use svn lock
- Avoid conflicts by good decomposition (modularity) and out-of-band coordination

Go forth and revise!

- Revision control will save you untold pain
  - Most people I know have accidentally nuked files or entire directories
  - Logs and diffs very useful for finding bugs
  - Much better way to coordinate with partners (but useful on your own! I use it for almost everything)
- Very small investment to learn
- Try it on your own!
- Read the SVN book online for more info