Practical software engineering: Revision control & make

15-441 Spring 2010, Recitation #2

Overview

- Revision control systems
  - Motivation
  - Features
  - Subversion primer
- Make
  - Simple gcc
  - Make basics and variables
  - Testing
- Useful Unix commands

Dealing with large codebases

- Complications
  - Code synchronization
  - Backups
  - Concurrent modifications
- Solution: Revision Control System (RCS)
  - Store all of your code in a repository on a server
  - Metadata to track changes, revisions, etc…
  - Also useful for writing books, papers, etc…

Basic RCS features (1/2)

- Infinite undo
  - go back to previous revisions
- Automatic backups
  - all your code ever, forever saved
- Changes tracking
  - see differences between revisions

Basic RCS features (2/2)

- Concurrency
  - Multiple people working at the same time
- Snapshots
  - Save stable versions on the side (i.e. handin)
- Branching
  - Create diverging code paths for experimentation

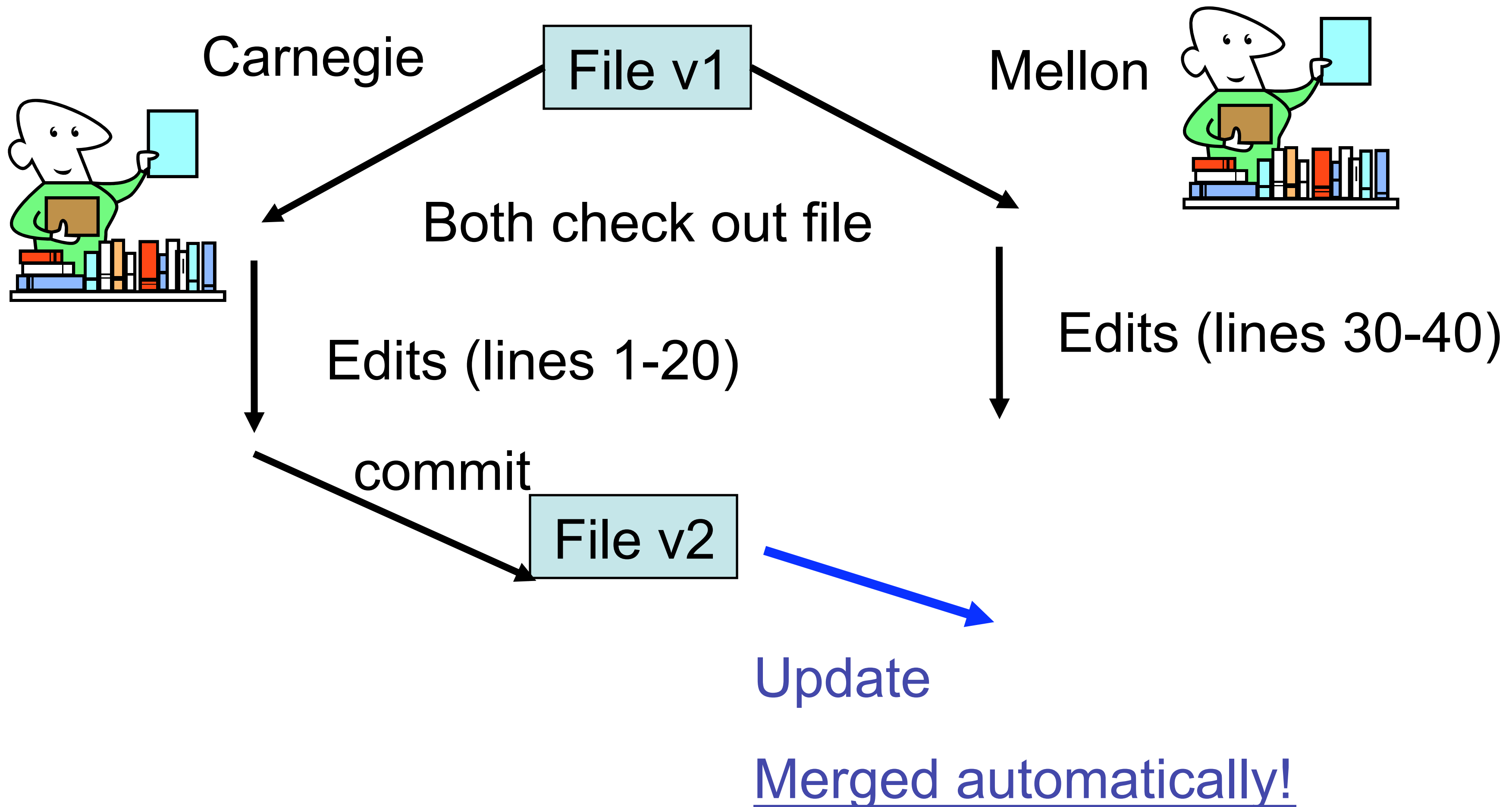
Typical RCS workflow

1. Create repository on RCS server
2. Checkout the repository to your local machine
3. Work locally: create, delete and modify files
4. Update the repository to check for changes other people might have made
5. Resolve any existing conflicts
6. Commit your changes to the repository

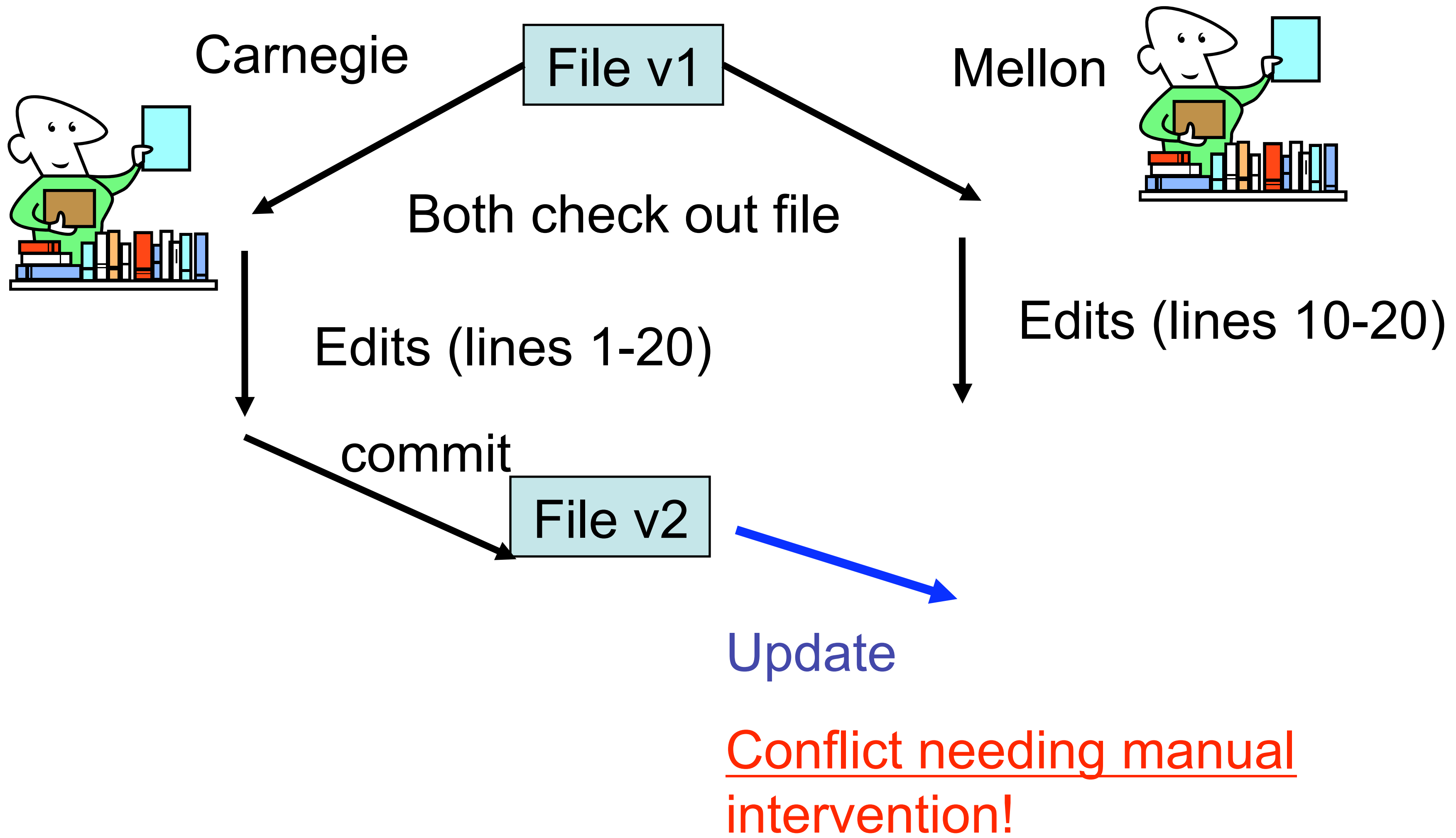
RCS implementations

- Revision Control System (RCS)
  - Early system, no concurrency or conflict resolution
- Concurrent Versions System (CVS)
  - Concurrency, versioning of single files
- Subversion (SVN)
  - File and directory moving and renaming, atomic commits

Concurrent edits (1/2)



Concurrent edits (2/2)



Resolving conflicts

- When changes can't be merged automatically
  - SVN gives you 3 files:
    - file.mine : your file
    - file.rx : the remote file
    - file : original file with marked conflicts
- You can
  - Discard your changes
  - Discard others' changes
  - Go over each conflict and arbitrate as appropriate

Interacting with SVN

- Command line
  - Readily available in andrew machines
- Graphical tools
  - Easier to use
  - Subclipse for Eclipse gives IDE/SVN integration

Command line SVN example (1/2)

```
$ svn co https://moo.cmcl.cs.cmu.edu/441-s10/svn/Project1Team63
A    Project1Team63/trunk
A    Project1Team63/branches
A    Project1Team63/tags
Checked out revision 1.
$ cd Project1Team63/trunk/

$ echo -e ”hello world" > sircd.c
$ svn add sircd.c

$ vim Makefile
$ svn add Makefile

$ svn commit -m 'adding Makefile and sircd.c!'

$ cd ../
$ svn cp trunk tags/checkpoint1
$ svn commit -m 'making a tag of the trunk for checkpoint1!'
```

Command line SVN example (2/2)

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
```

> svn diff -r 1:2 file

```
Index: file
===================================================================
--- file (revision 1)
+++ file (revision 2)
@@-1,2+1,3@@
This isatestfile
-It startedwithtwolines
+It nolongerhastwolines
+it hasthree
```

General SVN tips

- Update, make, test, only *then* commit
- Merge often
- Comment commits
- Avoid commit races
- Modular design avoids conflicts

Know more

- Chapter 2 of Dave Andersen's notes "SE for Systems Hackers" (link on course website)
- subversion.tigris.org for SVN software & info
- svnbook.red-bean.com for SVN book

Make

- Utility for executable building automation
- Saves you time and frustration
- Helps you test more and better

Simple gcc

If we have files:

- prog.c: The main program file
- lib.c: Library .c file
- lib.h: Library header file

```
% gcc -c prog.c -o prog.o
% gcc -c lib.c -o lib.o
% gcc lib.o prog.o -o binary
```

gcc flags

- Useful flags
  1. -g: debugging hook
  2. -Wall: show all warnings
  3. -Werror: treat warning as errors
  4. -O0, -O1, -O2, -O3: optimization level
  5. -DDEBUG: macro for DEBUG (#define DEBUG)
- Avoid using dangerous optimizations that could affect correctness

More gcc

```
% gcc -g -Wall -Werror -c prog.c -o prog.o
% gcc -g -Wall -Werror -c lib.c -o lib.o
% gcc -g -Wall -Werror lib.o prog.o -o binary
```

This gets boring, fast!

Makefile basics

- Build targets

```
target: dependency1 dependency2 ...
        unix command (start line with TAB)
        unix command
```

Makefile example

```
binary: lib.o prog.o
    gcc -g -Wall lib.o prog.o -o binary

lib.o: lib.c
    gcc -g -Wall -c lib.c -o lib.o

prog.o: prog.c
    gcc -g -Wall -c prog.c -o prog.o

clean:
    rm *.o binary
```

Makefile variables (1/7)

- Variables

```
CC = gcc
CFLAGS = -g -Wall -Werror
OUTPUT = binary
```

Makefile variables (2/7)

```
binary: lib.o prog.o
    gcc -g -Wall lib.o prog.o -o binary

lib.o: lib.c
    gcc -g -Wall -c lib.c -o lib.o

prog.o: prog.c
    gcc -g -Wall -c prog.c -o prog.o

clean:
    rm *.o binary
```

Makefile variables (3/7)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary

$(OUTPUT): lib.o prog.o
    $(CC) $(CFLAGS) lib.o prog.o -o binary

lib.o: lib.c
    $(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
    $(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
    rm *.o $(OUTPUT)
```

Makefile variables (4/7)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary

$(OUTPUT): lib.o prog.o
	$(CC) $(CFLAGS) lib.o prog.o -o binary

lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
	rm *.o $(OUTPUT)
```

Makefile variables (5/7)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
	rm *.o $(OUTPUT)
```

Makefile variables (6/7)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
    $(CC) $(CFLAGS) $(OBJFILES) -o binary

lib.o: lib.c
    $(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
    $(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
    rm *.o $(OUTPUT)
```

Makefile variables (7/7)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
    $(CC) $(CFLAGS) $(OBJFILES) -o binary

%.o: %.c
    # $<: dependency (%.c)
    # $@: target (%.o)
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm *.o $(OUTPUT)
```

Simple Test Script (1/2)

```
% ./server 6667 &
% cat testfile.01 | ./testscript.py
% cat testfile.02 | ./testscript.py
% killall -9 server
```

Simple Test Script (2/2)

```
#/bin/sh

echo “Starting server on port 6667.”
./server 6667 &
SERVERPID = $!

echo “Running test files.”
cat testfile.01 | ./testscript.py
cat testfile.02 | ./testscript.py

echo “Killing server process.”
kill $(SERVERPID)
```

Augmenting the Makefile for testing (1/2)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

all: $(OUTPUT)

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

%.o: %.c
	# $<: dependencies (%.c)
	# $@: target (%.o)
	$(CC) $(CFLAGS) -c $< -o $@


clean:
	rm *.o $(OUTPUT)
```

Augmenting the Makefile for testing (2/2)

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

all: $(OUTPUT)  test

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

%.o: %.c
	# $<: dependencies (%.c)
	# $@: target (%.o)
	$(CC) $(CFLAGS) -c $< -o $@

test: $(OUTPUT)
	sh ./testscript.sh

clean:
	rm *.o $(OUTPUT)
```

Using the Makefile

```
% make
% make test
% make clean
```

- Know more

  Google

  – “makefile example”

  – “makefile template”

  – “make tutorial”

  – Chapter 3 of Dave Andersen’s notes

Extra: useful Unix commands (1/2)

- find “func_name” in files

```
% grep -r func_name .
```

- replace “bad_func_name” to “good_func_name”

```
% sed -e “s/bad_func_name/good_func_name/g”\
  prog.c > prog.c.new
```

Extra: useful Unix commands (2/2)

- find a file named “prog.c”

```
% find -name prog.c
```

- download files from the Internet

```
% wget http://address/to/file.tar.gz
```

- untar and unzip a file

```
% tar xzvf file.tar.gz
```