

CME 212: Unix tools

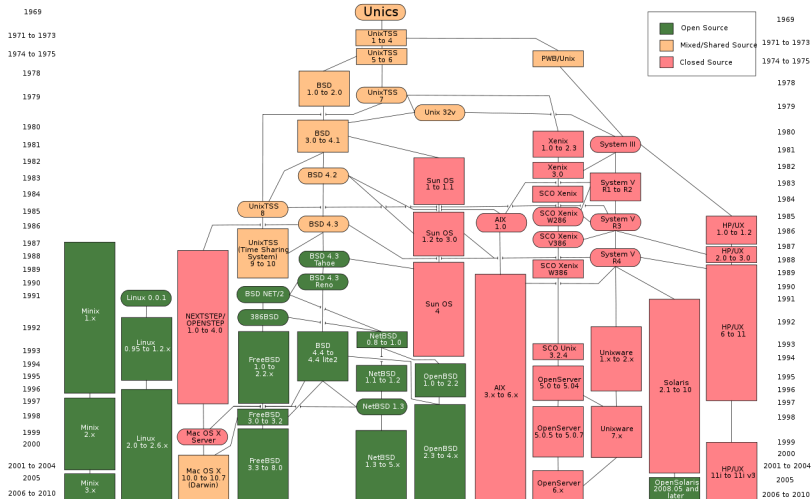
Lecture 6

January 27, 2012

UNIX

- Started in 1969 by a group AT&T (Bell Labs) employees: Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, and Joe Ossanna.
- First written in assembly language. All tools in assembly language!
- By 1973 almost entirely ported to C, which facilitated porting to other hardware.
- Because of antitrust settlements, AT&T was not allowed to enter the computer industry. Bell labs had to give it to anybody who asked.
- Over the decades there have been many derivatives and many lawsuits.
- Today we typically use a variance called GNU/Linux.

UNIX History



UNIX Philosophy

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Douglas McIlroy, the inventor of Unix pipes

POSIX

- Means: Portable Operating System Interface
- Standard specified by IEEE for interoperability between operating systems
- Includes: C, C standard library, process creation and control, error handling, pipes, I/O...
- One of the things that makes most Unix-like operating systems similar
 - For example, many programs written for GNU/Linux will work on Mac OS X

GNU/Linux

- The GNU project, founded and lead by Richard Stallman, gives us most of the tools we use including `gcc`, `gdb`, `bash`
- The Linux project, founded and lead by Linus Torvalds, provides the operating system kernel
- Combined, it's known as GNU/Linux. There are many distributions: Debian, Redhat, Ubuntu, Arch, Gentoo. . .
- The name "GNU" is a recursive acronym for "GNU's Not Unix!". "GNU" is pronounced g'noo, as one syllable, like saying "grew" but replacing the r with n.
- GNU and Linux don't use copyrighted UNIX code. There have been many lawsuits over this.

System properties: `uname -a`

My laptop:

```
Darwin nwh-mbair.local 10.8.0 Darwin Kernel Version
10.8.0: Tue Jun 7 16:33:36 PDT 2011;
root:xnu-1504.15.3~1/RELEASE_I386 i386
```

My workstation:

```
Linux icme-nwh 2.6.35-31-generic #63-Ubuntu
SMP Mon Nov 28 19:29:10 UTC 2011 x86_64 GNU/Linux
```

corn.stanford.edu:

```
Linux corn19.stanford.edu 2.6.38-10-server
#46-Ubuntu SMP Tue Jun 28 16:31:00
UTC 2011 x86_64 x86_64 x86_64 GNU/Linux
```

Using the system: the shell

- A shell is program that reads, interprets, and executes commands from the user
- Most shell programs today have all the features of full fledged programming languages
- `bash` is default on most GNU/Linux systems. The `corn` admins make `tcsh` default for users. You can always start `bash` by just typing `bash`.
- Today we cover `bash`. Things like piping and redirection are the same. You must use `setenv` to set environment variables in `tcsh`. Also, the configuration scripts are different.

Basic bash usage

Enter commands at the command line:

```
$ command arg1 arg2 arg3
```

There are two types of commands: **executables** and **builtins**. **builtin** commands are executed directly by the shell. There need not be an executable file. For example, `cd` for change directory is a builtin. If you do `$ man cd` you will be taken to the documentation for **builtin** commands.

You can also **alias** commands and create `bash` **functions**.

Basic bash usage

Table: Some basic commands

command	function
cd	change directory
ls	list files
pushd	push directory onto stack
popd	pop directory from stack
pwd	print working directory
cd -	change to previous directory
cd ~	change to HOME directory
cat	print file to stdout
less	basic text reader (pager)
file	inspect file type
which	inspect command
top	system activity

Where does `bash` find commands

The basic rules:

- 1 If the command name contains slashes, the shell attempts to call the executable.
- 2 If the command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, that function is invoked as described in Shell Functions.
- 3 If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked.
- 4 If the name is neither a shell function nor a builtin, and contains no slashes, Bash searches each element of `$PATH` for a directory containing an executable file by that name.

Practical examples

Execute command in current directory

```
$ ./hello_world
```

Locate and execute program

```
$ which top
```

```
/usr/bin/top
```

```
$ top
```

```
$ /usr/bin/top # does same thing
```

Alias commands

- On my laptop

```
$ ls
DOS Games Downloads Movies Public matlab
Desktop Dropbox Music Sites tmp
Documents Library Pictures bin
```

- I can alias `ls` to something I like better

```
$ alias ls="ls -l"
$ ls
total 0
drwxr-xr-x@ 10 nwh staff 340 Nov 28 11:39 DOS Games
drwx-----+ 4 nwh staff 136 Jan 26 08:24 Desktop
drwx-----+ 8 nwh staff 272 Dec 27 10:02 Documents
drwx-----+ 5 nwh staff 170 Jan 25 20:53 Downloads
drwx-----@ 28 nwh staff 952 Jan 22 22:16 Dropbox
drwx-----+ 40 nwh staff 1360 Jan 20 21:34 Library
drwx-----+ 3 nwh staff 102 Aug 29 16:23 Movies
drwx-----+ 12 nwh staff 408 Sep 3 22:44 Music
drwx-----+ 10 nwh staff 340 Dec 22 13:06 Pictures
drwxr-xr-x+ 5 nwh staff 170 Aug 29 16:23 Public
drwxr-xr-x+ 5 nwh staff 170 Aug 29 16:23 Sites
drwxr-xr-x 4 nwh staff 136 Sep 11 15:46 bin
drwxr-xr-x 2 nwh staff 68 Sep 4 15:46 matlab
drwxr-xr-x 5 nwh staff 170 Jan 8 13:07 tmp
```

Determine if a command is aliased

Just use the alias command:

```
$ alias ls  
alias ls='ls -l'
```

If the command is not aliased:

```
$ alias cat  
-bash: alias: cat: not found
```

The PATH environment variables

- Inspect PATH:

```
$ echo $PATH
./bin:/Users/nwh/bin:/Users/nwh/.cabal/bin:/Applications/MATLAB_R2011a_Student.app/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin:/usr/X11/bin
```

- Directories separated by “:” and listed in decreasing precedence.
- Add something to PATH:

```
$ export PATH=/path/to/dir:$PATH
```

Configuring bash

- People often like to configure `bash` with their own aliases and variables
- This is done with `bash` configuration scripts
- A basic `.bashrc` might look like

```
# add matlab to the path
export PATH=/opt/matlab/bin:$PATH
```

```
# alias ls
alias ls='ls -l'
```

Bash startup procedure

- When started as an interactive login shell:
 - Bash reads and executes the `/etc/profile` (if it exists).
 - After reading that file, it looks for `~/.bash_profile`, `~/.bash_login`, and `~/.profile` in that order, and reads and executes the first one (that exists and is readable).
- When a login shell exits:
 - Bash reads and executes `~/.bash_logout` (if it exists).
- When started as an interactive shell (but not a login shell):
 - Bash reads and executes `~/.bashrc` (if it exists). This may be inhibited by using the `--norc` option. The `--rcfile` file option will force Bash to read and execute commands from file instead of `~/.bashrc`.

Bash definitions

- **interactive shell**: the shell is connected to a terminal for input and output. A non-interactive shell is normally running a shell script.
- **login shell**: this is simply a flag to determine the start up sequence.

Practical bash setup

- Have `~/.bash_profile` call `~/.bashrc` and do nothing else:

```
# ~/.bash_profile
# executed by bash for login shells.
if [ -e ~/.bashrc ] ; then
    . ~/.bashrc
fi
```

- Put everything in `~/.bashrc`
- Now all interactive shells should work
- Warning: some operating systems mess this all up. Mac OS X and Ubuntu seem to do the right thing.

bash setup examples

- Ryan McGeary

`https://github.com/rmm5t/dotfiles`

- NWH, derived from `rmm5t`

`https://github.com/nwh/config`

Another item of advice

- Bash and other shell programs are also complete programming languages. They have arrays, loops, conditionals, functions, and more. However, even simple shell programs are hard to read and debug.
- Use shell scripts for simple sequences of commands
- Use python for anything more complicated. i.e. if you need a function, loop, or conditional.

Example bash script

- `upload.sh`:

```
#!/bin/bash

# the first line indicates the interpreter that should be used for the file

cd WWW
make
cd ..
rsync -e ssh -avz WWW cardinal.stanford.edu:/afs/ir.stanford.edu/class/cme212
```

- Make file executable and run

```
$ chmod +x upload.sh
$ ./upload.sh
```

Basic redirection and piping

- Bash and other shells allow us to direct the output of a command to a file or to another command.
- Example count the number of files in a directory:

```
$ ls -l | wc -l  
14
```

Here I tell `ls` to give me a list of all files in a directory with one line per file. The text output is “piped” to `wc` with the `-l` flag, which will count lines.

- We can direct output to a file

```
$ ls -l > my_files.txt
```

- These sorts of combinations make unix shells very powerful.

Basic piping

- The basic form is:

```
$ command1 [| or |&] command2
```

- The “|” connects **stdout** from `command1` to **stdin** of `command2`
- A “|&” connects **stderr** from `command1` to **stdin** of `command2`
- You can chain commands:

```
ls -l | wc -l | less
```

Basic redirection

- General form:

```
$ command1 symbol file
```

- Example:

```
$ find . -name "*.m" > dot_m_files.txt
```

Table: Basic redirection symbols

symbol	function
>	stdio to file
<	file to stdin
>>	append stdio to file
&>	stdio and stderr to file
&>>	append stdio and stderr to file

Version control

Version control systems attempt to solve many problems that come up in the development of code:

- accident recovery: “Oops, I just deleted `important.c!`”
- experimentation: “I want to try X, but don’t want to break Y”
- collaboration: “How long ago did you email me that code?”
- record of history: “Our only customer wants the code from 2004”

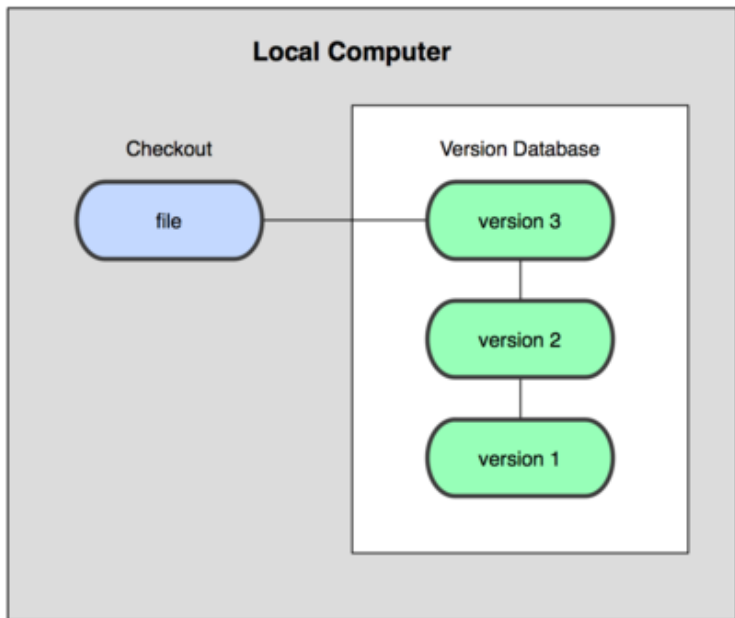
Some history and terms

- **RCS**: “revision control system”, old, local, file system based (1982)
- **CVS**: “concurrent versioning system”, adaptation of RCS to client-server model (1990)
- **SVN**: “subversion”, a newer, better implementation of CVS (2000)
- **Git, Bzr, Mercurial**: “distributed” version control systems
- There are a few more
- We will use **git**

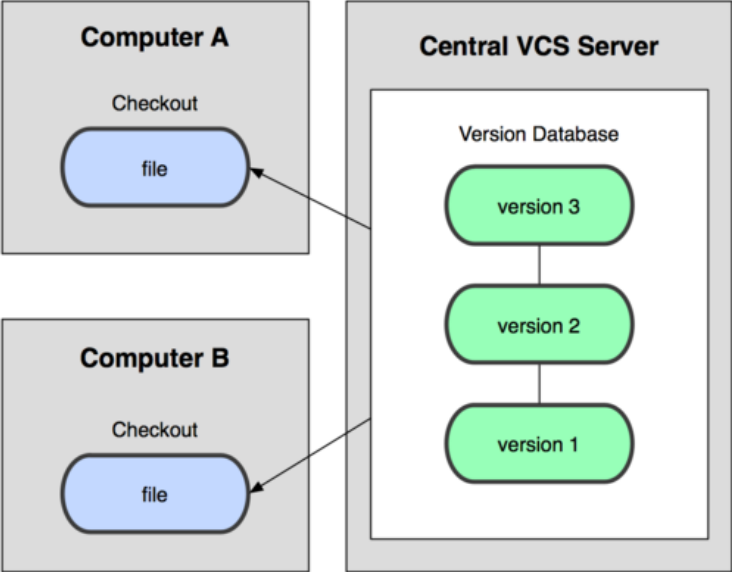
Pro Git

The following slides and discussion is pulled from an open source book by Scott Chacon called Pro Git. It's a great resource. Please see the links on the homepage.

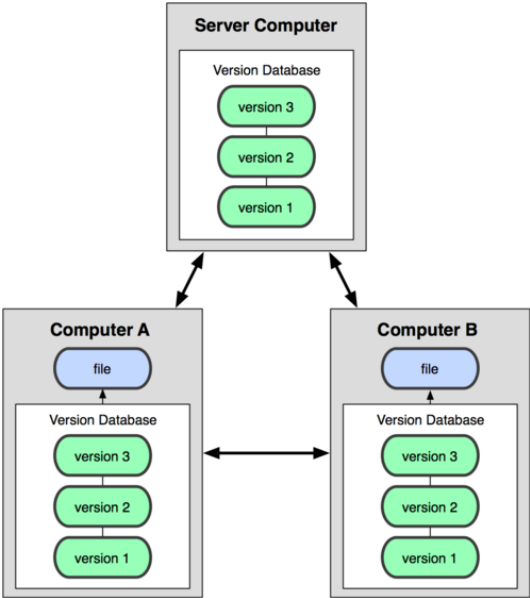
Local VCS



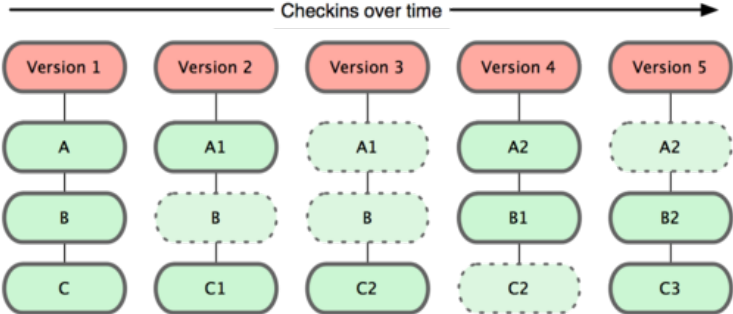
Local VCS



Distributed VCS



How it works



Obtain git

- For Mac and Windows see `git-scm.org`
- For GNU/Linux use the package manager
- Good tool for Mac: GitX, see `gitx.frim.nl`
- Good tool for Windows: GitExtensions, see `sourceforge.net/projects/gitextensions/`
- Good tools for GNU/Linux: `qgit`, `git-cola`

Use git

Initialize your local git repository with:

```
$ git init .
```

After you create some files, add and commit them to the repository:

```
$ git add foo.c bar.c  
$ git commit -m "added foo.c"
```

After you make a change to a file, add and commit the changes:

```
$ emacs bar.c  
$ git add bar.c  
$ git commit -m "fixed nasty bug in bar.c"
```

You can check the history of your work with:

```
$ git log
```

Staying connected with `screen`

There is a tool called `screen`, see

```
$ man screen
```

Most basic usage:

- connect to `corn.stanford.edu`
- start `screen`

```
$ screen
```

- Run your program, exit `screen` with `C-a d`
- Get coffee
- Reconnect to same `corn` server, and reconnect to `screen` with

```
$ screen -r
```

Some notes on screen

- To list running screen sessions: `$ screen -ls`
- If the terminal gets messed up: `$ reset`
- On `corn` you may lose the AFS token after a period of time. This would prevent you from accessing your home directory. I once heard there was a way to deal with this, but I can't remember it.