CISC 372: Parallel Computing

Unix

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

How you can play along with today's lecture

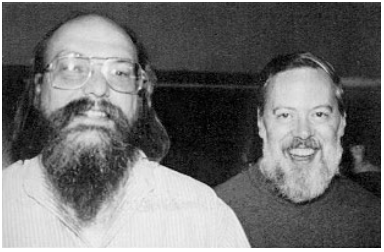
Hopefully one of the following will work for you...

- if you have a Mac, open up a Terminal window
- if you have a Linux VM, start it up, open up a Terminal window
- Windows users: install WSL2
- `ssh cisc372.cis.udel.edu` (you must first be on UD VPN)

UNIX Historical Overview

- AT&T, circa 1970
- started as a reaction to Multics
  - revolutionary OS under development at MIT, AT&T, GE
  - Multics thought to be too complicated but having good ideas
  - Multics = Multiplexed Information and Computing Service
  - Unics = UNiplexed Information and Computing Service
- Ken Thompson, Dennis Ritchie, M. D. McIlroy, J. F. Ossanna, ...
- C language invented in order to develop Unix in a portable way
  - previously used assembly language
- extremely influential and popular, leading to many variants
  - BSD Unix (Berkeley Standard Distribution)
    - Free-BSD, OpenBSD, DragonFly BSD $\cdots \rightarrow$ Darwin (OS X)
    - Solaris (Sun $\rightarrow$ Oracle)
  - GNU/Linux
  - AIX
  - Xenix

Ken Thompson and Dennis Ritchie



Unix: Main ideas

- multitasking, multi-user
- set of relatively independent tools built around a single small kernel
- modularity, re-usable software components
- everything is a file
  - the filesystem is the main means of communication
  - printer, terminal, drives, ... are all files
  - most commands are files: files can be made executable and thus become commands
- shell scripting + command language to combine different tools
- hierarchical file system with no limit on hierarchy
- command interpreter is just another program (shell)
  - makes it very easy to create new commands and thereby extend the language
- all files: newline-delimited ASCII
- use of regular expressions
- self-documenting (man)

UNIX: File system

- the file system is a rooted tree
- the root is `/`
- `a/b/c`
  - `c` is a child of `b` is a child of `a`
  - `a` and `b` are directories (a kind of file)
  - `c` may or may not be a directory (but is a file)
- `.` is the current (working) directory
- `..` is the parent directory of the current directory
- each file has certain metadata associated to it by the OS
  - owner: ID number of the user who "owns" this file
  - group: ID number of the group for this file
  - permissions: who can read/write/execute this file
    - permissions for the owner
    - permissions for the group
    - permissions for everyone else

Using the bash shell

- interactive: prompt, you type a command, get a response, repeat
- an any time, you are "in" a directory (point in the hierarchy)
  - the working directory
- `ls` : list the contents (children) of the working directory
  - `ls -l` : long format: show the user, group, file size, permissions, etc.
  - `ls -a`: show all files including the hidden ones (start with `.`)
- `pwd` : print working directory
- `cd` : change directory to another directory
  - argument to `cd` is a path to a directory
  - argument can be absolute or relative
  - absolute: starts with `/` (root)
  - relative: starts with anything else, is interpreted to be a direction starting from working directory
- `mv a b`: move a file from `a` to `b`
  - this can be used to rename a file
  - or it can be used to move a file into another directory (change the hierarchy)
  - remember: "file" above can be a directory

bash shell commands, cont.

- `cp a b` : copy a file to another directory (or file)
- `rm a` : remove the file named `a`
- `rmdir a` : remove the empty directory named `a`
- `man cmd` : show the manual page for the command cmd
- `chmod` : change the permissions on a file
  - `chmod ugo+rx foo`
    - give everyone read and execute permission on file `foo`
  - `chmod go-w foo`
    - take away write permission from group and others on `foo`
- `chown` : change the owner of the file
- `chgrp` : change the group of the file
- `touch filename` : creates an empty file with that name
- `cat filename` : print the file to the terminal
- `more filename` : page through the file one screen at a time
- `bash` : start (another) bash shell

Exercise 1

1. Create a directory in your home directory called `A`.
2. Create two sub-directories of `A` called `B` and `C`.
3. Adjust the permissions of `C` so that only you can read, write or change into it.
4. Create a file called `foo.txt` in `C`.
5. Copy `foo.txt` to `B`. Check that both copies are really there.
6. Delete both copies of `foo.txt`.
7. Delete `B` and `C` (command: `rmdir`).
8. Delete `A`.

Execution, environment

- many different kinds of executable files can be created
  - shell (e.g., bash) scripts
  - files created by compiling a C program, etc.
- the file then becomes a command
  - just type the full path to the file (and any command line arguments)
  - the user doing this must have execute permission on the file
- to avoid typing the full path, put the file "in your PATH"
- `PATH` is an environment variable
  - a variable used by the shell
- `PATH` is a colon-separated list of directories
- when you type a command in the shell, it looks in the directories in your path for a file with that name (in order)
  - if and when it finds the file, it executes it

Execution, environment, cont.

- `X=foo` : set environment variable `X` to `foo`
- `$X` : expands to current value held by environment variable `X`
- `echo $X` : print the value of the environment variable `X`
- `export X=foo` : set `X` to `foo` and carry this over to all children shells
- `export PATH=/users/joe/bin:$PATH`
  - add `/users/joe/bin` to the front of the list of directories in the PATH
  - `set` : show the current environment

Package managers

- installing, configuring, organizing, and managing software is hard
  - even with `make`
- modern Unix distributions come with package managers
- maintain large databases of software "packages" and dependencies
  - e.g., Subversion 1.9.2 requires gcc 4.6.2 and ...
- these make it very easy to install, update, uninstall software and keep everything consistent
- Mac
  - MacPorts: `https://www.macports.org`
  - Homebrew: `https://brew.sh`
- Ubuntu, Debian
  - Advanced Packaging Tool (APT): `https://help.ubuntu.com/community/AptGet/Howto`

APT

Common commands:

1. `apt-get install` ⟨*package_name*⟩
   - install a package
2. `apt-get update`
   - update your local list of packages
3. `apt-get upgrade`
   - upgrade all your installed packages to the latest versions
4. `apt-cache search` ⟨*search_term*⟩
   - search for packages with names or descriptions matching the string
5. `apt-cache show` ⟨*package_name*⟩
   - show the description of the package and other information

Note: Most commands must be preceded by `sudo`.

See

- `https://help.ubuntu.com/community/AptGet/Howto`

for many more commands and deatils.

MacPorts

Common commands:

1. `port install` ⟨*package_name*⟩
   - install a package
2. `port selfupdate`
   - update your local list of packages
3. `port upgrade outdated`
   - upgrade all your installed packages to the latest versions
4. `port search` ⟨*search_term*⟩
   - search for packages with names or descriptions matching the string
5. `port info` ⟨*package_name*⟩
   - show the description of the package and other information

Note: Most commands must be preceded by `sudo`.

See

- `https://guide.macports.org/#using.port`

for many more commands and deatils.

Text editors

There are many, but some of the most popular are...

1. `pico`, `nano` (simple, easy-to-use, not powerful enough for most programming)
2. `vi` (universal, dating back to mid-late 1970s)
3. `emacs` (powerful, extensible, also mid 1970s)
   - recommended
   - get: use package manager!
   - learn: `https://www.gnu.org/software/emacs/tour/`
   - quick reference card:
     `https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf`

Choose your editor:

- in your home directory, find file `.bash_profile`
- or create new file with that name, if it is not there
- any bash commands you put here will be executed every time you log on
- add line: `export EDITOR=emacs`
- this will become your default editor for many different tasks

Emacs: thousands of commands with keyboard binding

- C-x C-f: open file
  - C=control, push and hold as you type the next character
- C-x C-s: save file
- C-x C-c: exit
- C-x C-w: write file ("save as...")
- C-a: move to beginning of line
- C-e: move to end of line
- C-n: move to next line
- C-p: move to previous line
- C-f: move forward one character
- C-b: move backward one character
- C-v: move forward one page
- M-v: move backward one page
  - M=meta key, usually ESC, maybe option
- C-spc: set the mark
- C-w: cut everything from the mark to current position (the "region"), copying it into the buffer
- M-w: copy the current region into the buffer without cutting
- C-y: yank from the buffer
- C-g: cancel whatever you're in the middle of
- C-s: search forward incrementally
- C-x u: undo (as many times as you want)

Exercise 2

1. Create a directory called `ex2` and change into it.
2. Create a new file named `hi.c` with these contents:

```
#include <stdio.h>
int main() {
  printf("Hi there\n");
}
```

3. Compile the program: `cc -o hi hi.c`
4. List the directory. You should see a file `hi`.
5. Change the permissions on `hi` so anyone can execute it.
6. Execute `hi`: `./hi`
7. Put the directory containing `hi` in your `PATH`.
8. Change into some other directories and type `hi`.

Congrats: you have extended the language of your OS.

make

make is a utility for automating builds and other tasks that have complex dependency graphs.

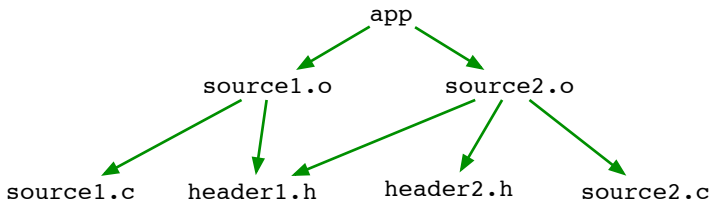
Example. You are developing a C program with source files:

1. `header1.h`
2. `header2.h`
3. `source1.c`, which includes `header1.h`, and
4. `source2.c`, which includes `header1.h` and `header2.h`

To build the binary `app` you issue the following command:

1. `cc -c source1.c` *[produces `source1.o`]*
2. `cc -c source2.c` *[produces `source2.o`]*
3. `cc -o app source1.o source2.o` *[produces `app`]*

make: dependency graph



- there is one build step for each non-leaf node in the graph
- suppose you modify `header1.h`
  - you need to repeat all 3 build steps
- suppose you modify `header2.h`
  - you only need to rebuild `source2.o` and `app`
- now imagine you have hundreds of nodes in a complex directed graph
- goal: when files are modified, figure out the minimal set of build steps to bring the system up-to-date

Makefile

Put the following in a file called "`Makefile`":

```
app: source1.o source2.o
     cc -o app source1.o source2.o

source1.o: source1.c header1.h
     cc -c source1.c

source2.o: source2.c header1.h header2.h
     cc -c source2.c
```

Then just type "`make`".

What `make` does

- `make` will figure out which nodes need rebuilding
- the `Makefile` consists of a set of rules
  - each rule has a target (left of colon)
  - followed by a set of prerequisites
  - then one or more recipes (executable actions to build the target)
- by examining time-stamps, `make` can tell if a target is older than one of its dependencies
  - such a target needs to be re-built
  - anything that depends on that target also needs to be re-built, etc.
- make executes the necessary recipes in the right order