Unix - Introduction

Kurt Schmidt

Dept. of Computer Science, Drexel University

July 20, 2016

Content originated by Jeremy Johnson and Vera Zaychik

Preface

Quick Note

- Many flavors of Unix, some for the PC platform, including many distributions Linux.
  - Collectively, they will be referred to as **nix*
- Where there's a difference, these notes discuss Linux, and many of the utilities from the gnome tookit
- So, on some other *nix platforms, you might notice slightly different behavior, maybe some missing options, some other small differences
  - E.g., `emacs` is the default Linux editor, rather than `vim`
  - Linux pushes `info` pages (but still has `man`)

Flavors of Unix

There are many flavors of Unix used by many people. This is *not* a complete listing:

- SysV (from AT&T)
- BSD (from Berkeley)
- Solaris (Sun)
- IRIX (SGI)
- AIX (IBM)
- OSF1 (DEC)
- Linux (free software)
  - Thank Linus Torvalds

*nix and Users

- Most flavors of *nix provide the same set of applications and services (commands, shells)
- Although these programs are not directly part of the OS, they are standardised enough that learning your way around one flavor of *nix is sufficient
- Unix got its start in the early 70s
- Was used (and grown) by engineering and science types

Notes for Mac OS X Users

- Since OS X, Mac runs on BSD Unix
- You can get many of the gnome command-line utilities discussed here just by installing XTools
- Or, homebrew provides access to coreutils, and other gnu utilities
  - Installed separately, can be made default
- You can simply open a terminal window, `ssh` to the department machines, and work there

Notes for Windows Users

- Cygwin – A *nix-like subsystem, runs on top of Windows
- Wubi – An Ubuntu installer for Windows (until 2015)
- Ubuntu on Windows 10 – User space and bash shell, running natively on Windows[1]
- You can install some flavor of Linux on a partition of your disk
- Or, run Linux inside a Virtual Machine

[1] I've not tried this

Intro

*nix Programming Environment

Objective: Introduce students to the features of *nix, and the Unix Philosophy (a collection of combinable tools and environments that support their use

- Basic commands
- File system
- Shell
- Filters (`more`, `grep`, `sort`, `wc`)
- Pipes, file redirection

Operating Systems

An *Operating System* controls (manages) hardware and software

- Provides support for peripherals such as keyboard, mouse, screen, disk drives, etc.
- The OS typically manages (starts, stops, schedules, etc.) applications
- Software applications use the OS to communicate with peripherals (screen, networking, etc), and with other applications

Kernel (OS)

- Interacts directly with the hardware through device drivers
- Provides sets of services to programs, insulating these programs from the underlying hardware
- Manages memory, controls access, maintains filesystem, handles interrupts, allocates resources of the computer

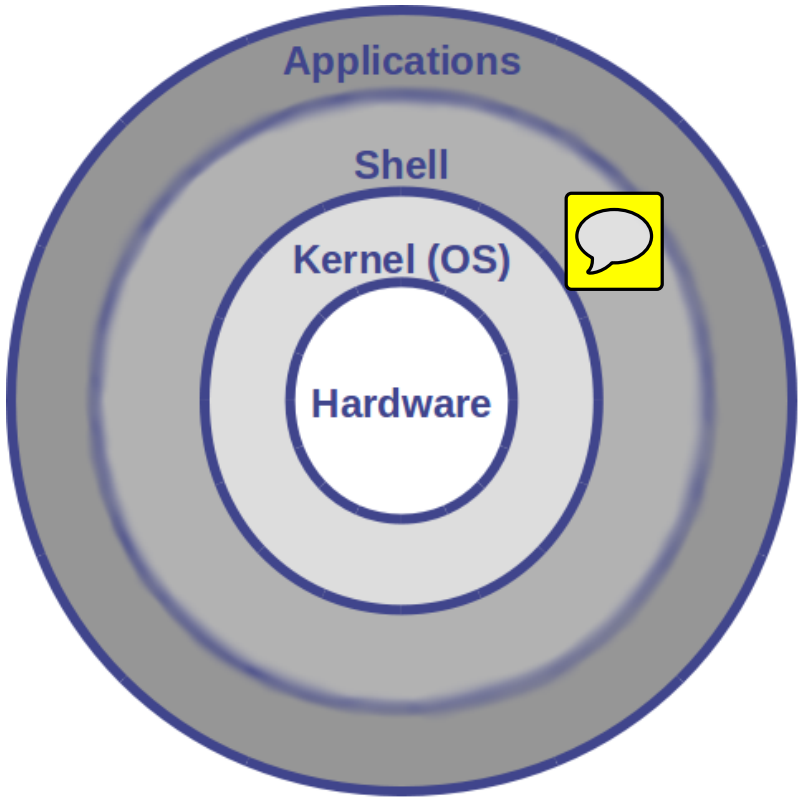
The Shell

- Helps manage user applications
- An *interactive shell* is the user interface
  - Responds to user commands
- A *desktop* is a GUI shell
- A shell *is* just another program

Structure of the *nix System

There are many standard applications:

- Filesystem commands
- Text editors
- Compilers
- Text processing



Logging In

Logging In

You can:

- Sit at the *console* (the computer itself)[1]
- Connect from any remote computer connected by a network, via  , e.g.
- Remember, usernames and passwords are case sensitive!

[1] Note, if you sit at any of the department Linux machines, your home directory will be mounted there. You shouldn't notice a difference

Incorrect Login

- You will receive the “Password:” prompt even if you type an incorrect or nonexistent login name
- Nothing will happen while you type your password. It’s fine

Can you guess why?

Connecting Remotely

- From a *nix machine, or a Mac, just open terminal, use `ssh`
- Windows doesn't have SSH built in
  - Any SSH client would do
  - I recommend PuTTY
  - Windows 10 is supposed to have the SSH stack, but I've not seen it yet
- To avoid always typing your password, search the Web for `ssh-keygen`
- **Keep your passwords and keys safe!**

CS Dept. Machines

- See `http://www.cs.drexel.edu/~kschmidt/Ref/csLogin.html`
- All CS machines are running Linux
  - `tux.cs.drexel.edu` – a farm you may connect to from anywhere on the 'Net.
  - Lab machines – any of the desktop machines in the labs
- Your files are backed up daily (nightly)

Usernames

Typically (or, on tux, anyway):

- A sequence of alphanumeric characters (there might be some others)
- Length no more than 8
- The primary identifying attribute of your account
- Unique (so, typically how I know and refer to you)
- Used as your email address
- The name of your home directory is related
  - On the CS machines, if your ID is `abc123`, then your home directory is `/home/abc123`

Passwords

- A secret string, not even the system knows
  - System hashes (encrypts) the password, compares it to the stored hash
- Should have at least 6 characters
- Should contain upper- and lower-case letters, numbers, and even other characters
- Don't use anything that appears in any dictionary
- Don't use anything that can be gleaned from your past, or your current likes
- Consider a line in a song, or poem. Use the first letter of each word

Filesystem

User's Home Directory

- The user's personal directory
  - All home (users') directories on tux are in `/home` : E.g., `/home/kschmidt`
- Where all your files go (hopefully organised into subdirectories)
- Mounted from a file server – available on *any* department machine you log into

Home Directory

- Your *current working directory* (CWD) when you log in
- `cd` (without an argument) takes you home
- Location of many startup and customisation files:
  - `.bashrc .vimrc .forward .plan`

Files and File Names

- A *file* is a basic unit of storage (e.g., the disk)
- Every file has name
- Filenames are case sensitive
- Unix filenames can contain any character except the slash (/) and the null character
  - Some characters, like shell metacharacters, make it more difficult to refer to the file

File Names

- Every file has at least one name
  - See `ln`, inodes
- Each file *in the same directory* must have a unique name
- Files in different directories can have identical names
- Files that start with a `.` are, by default, hidden by `ls`, and other utilities

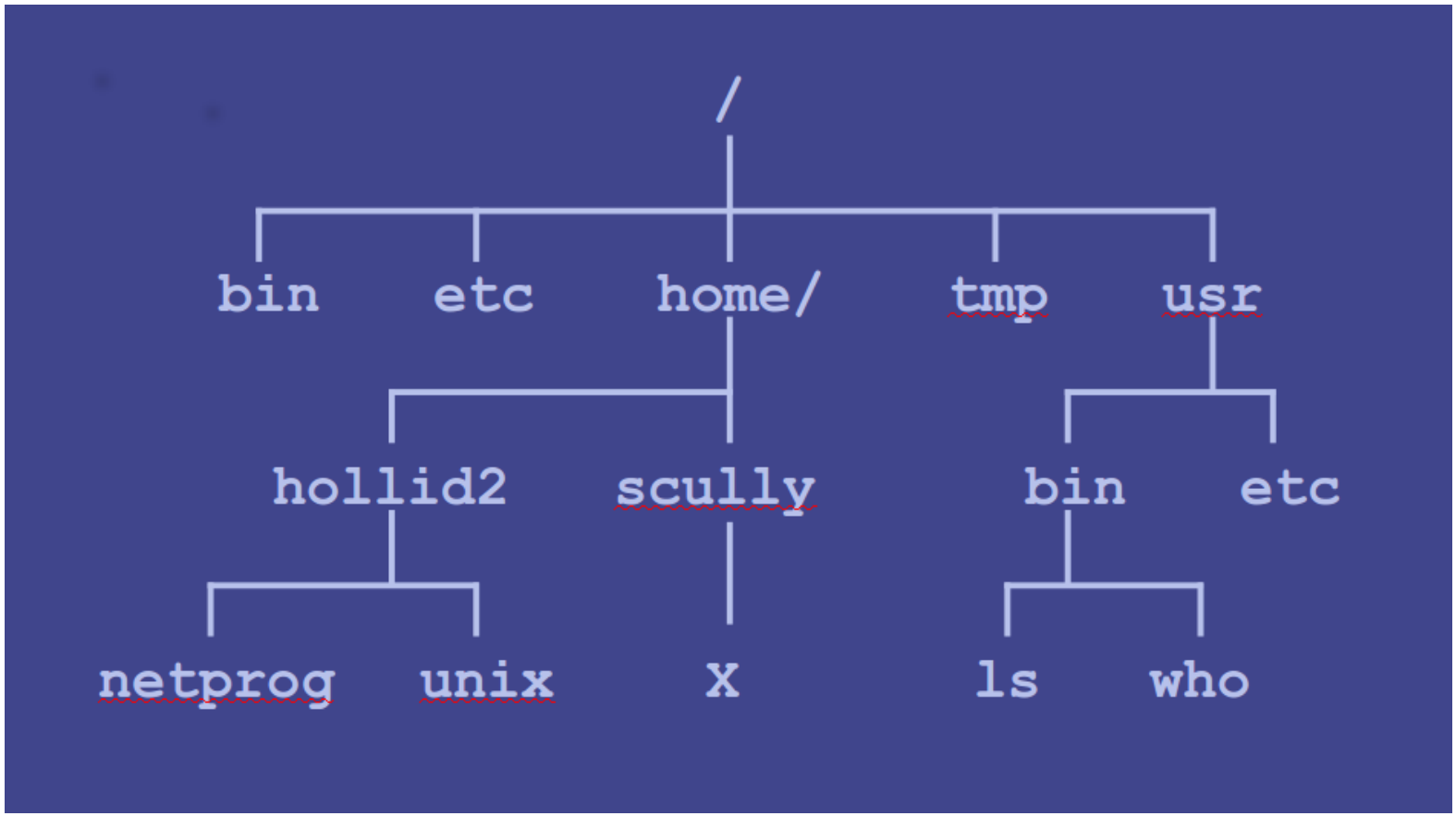
Directories

- Sometimes called a *folder*
- A *directory* is a special sort of file
  - holds information about other files
- Container for other files (including directories)
- Other file types include *symbolic links* (just like shortcuts), *named pipes*, *block special files* (disks, USB drives)

Unix Filesystem

- A hierarchical system of organising files and directories
- The top level in the hierarchy is called the *root*
  - Holds *all* files and directories in the filesystem
  - Its name is /
- Filesystem may span many disks, even across a network

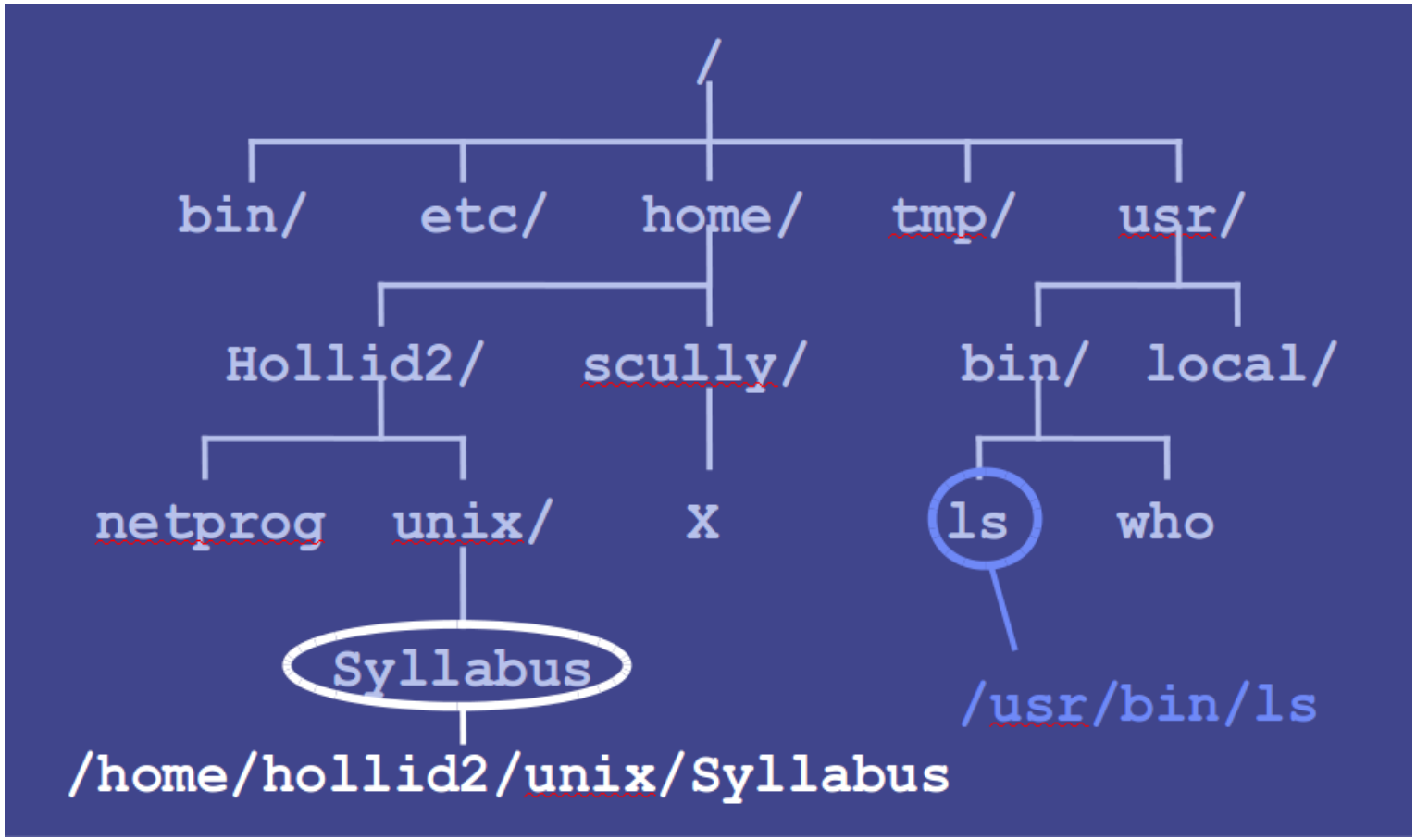
Filesystem – eg.



Pathnames

- The *pathname* of a file includes the name of the file, the directory that holds the file, the directory that holds *that* directory... up to the root
- The pathname of every file in a given filesystem is unique
- Absolute pathnames start at the root, drill down through successive subdirectories
- The forward slash, /, separates path components

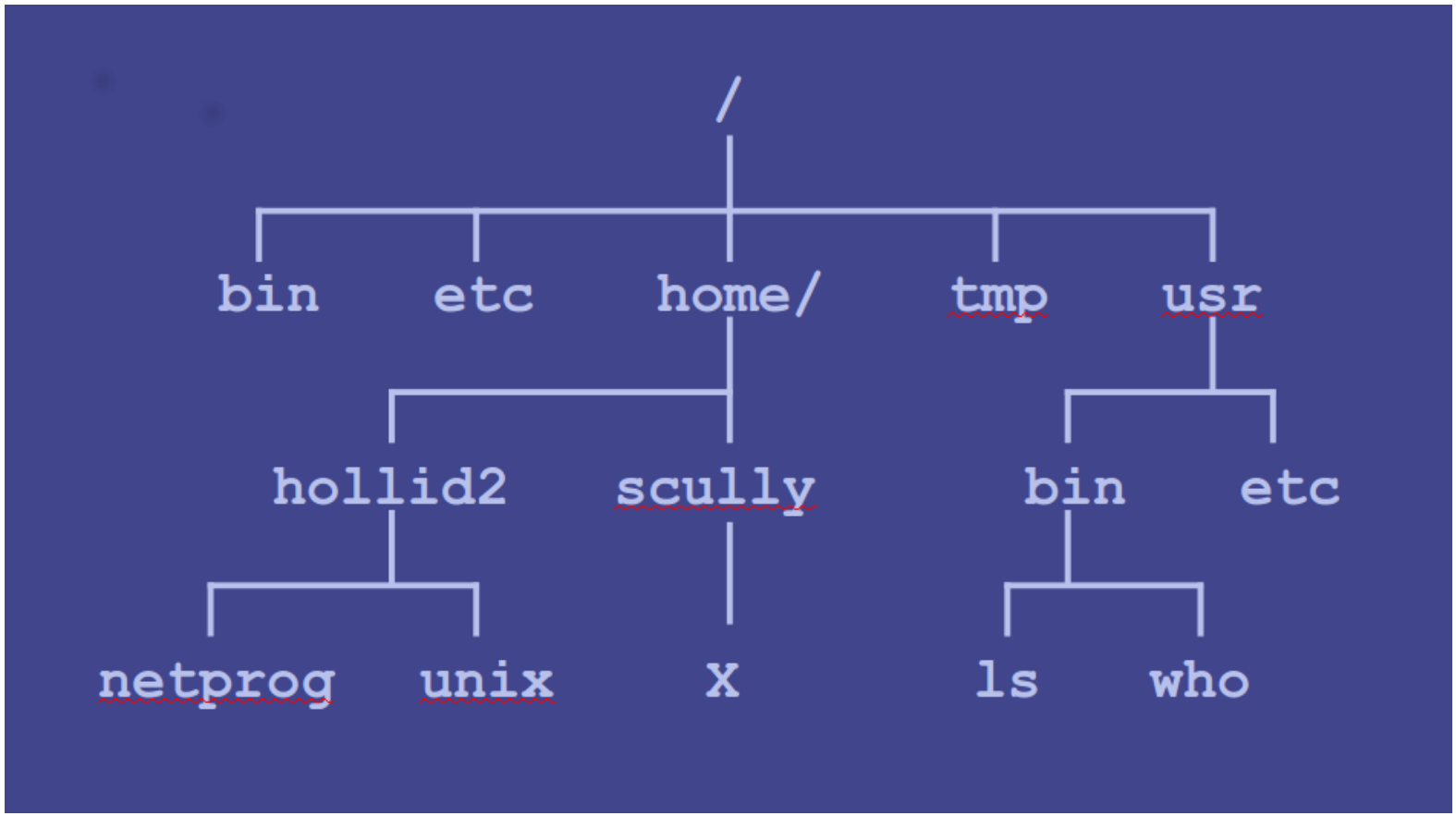
Pathnames – eg.



Absolute Pathnames

- The pathnames, above, are *absolute* pathnames
- Start at the root
- Uniquely identify files
- There are 2 absolute paths that don't, apparently, start at the root:
  - `~kschmidt/` $\Leftrightarrow$ `/home/kschmidt` (to refer to any user's home directory)
  - `~/` – **Your** home directory. So, relative to login, `$USER`

Filesystem – eg.



Relative Pathnames

- Relative to the current working directory
- `$PWD` is maintained by the shell, the `cd` shell builtin
- Prefixed with the current directory, `$PWD`
- Uniquely identify files
- There are 2 absolute paths that don't, apparently, start at the root:
  - `~kschmidt/` $\Leftrightarrow$ `/home/kschmidt` (to refer to any user's home directory)
  - `~/` – **Your** home directory. So, relative to login, `$USER`

Relative Pathnames

- Prefixed w/the current directory, `$PWD`
- So, relative to the current directory

```
$ cd /home/hollid2
$ pwd
```

```
/home/hollid2
```

```
$ ls unix/Syllabus
```

```
unix/Syllabus
```

```
$ ls X
```

```
ls:  X:  No such file or directory
```

```
$ ls /home/scully/X
```

```
/home/scully/X
```

Special Relative Paths

- . – the *current* directory
- .. – the *parent* directory

```
$ cd ~scully
$ pwd
```

```
/home/scully
```

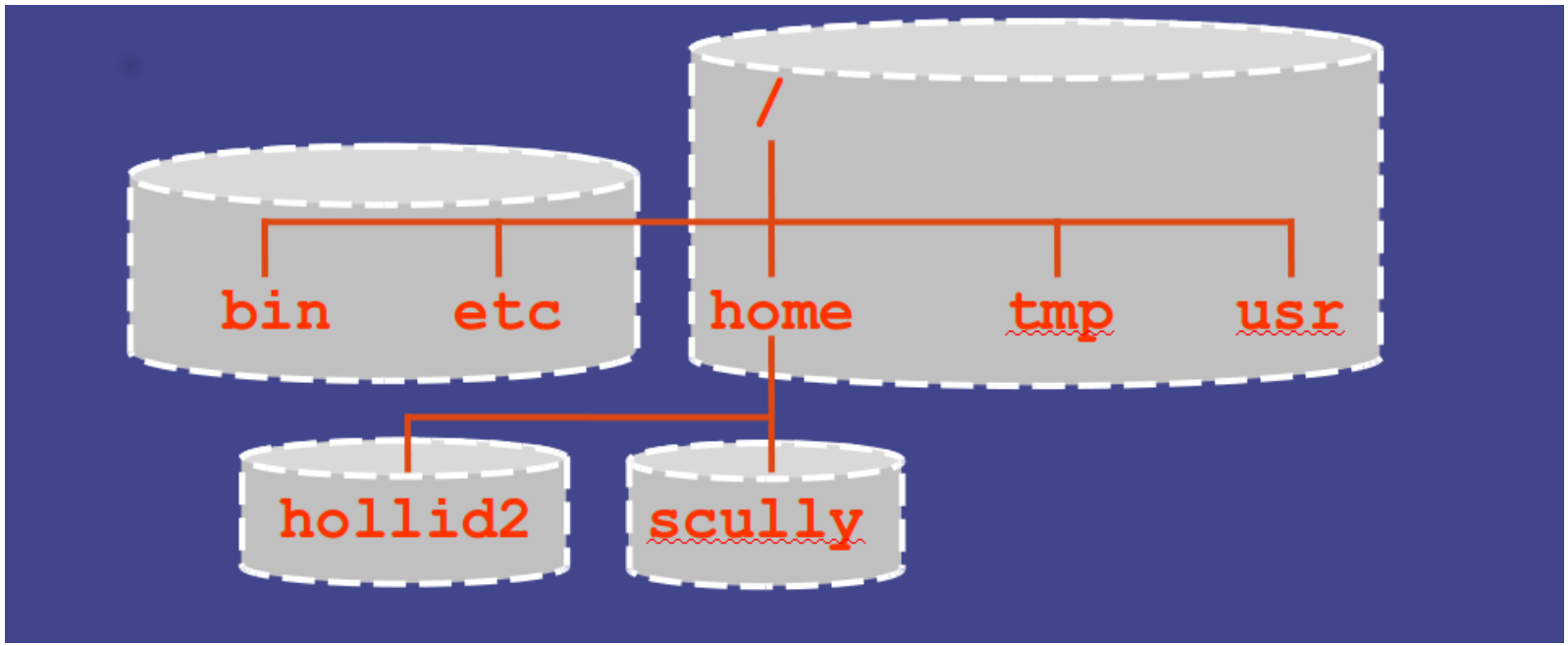
```
$ ls -F ../Hollid2/
```

```
netprog unix/
```

```
$ cp ../Holid2/unix/Syllabus . # copy that file here
```

Filesystem v. Disk

- The hierarchy can actually span parts of many disk drives (partitions)
- Even partitions on other computers



Commands

Commands – Basic Syntax

Bash is the default shell, and the one we'll discuss here

- Tokens are separated by whitespace
- Shell expects the first token to be a command[1]
- All subsequent tokens are arguments
- Arguments that start with a dash, -, or two dashes, are called *options* (generally, Posixly)
  - Used to modify the behavior of the command
  - Note, not all utilities are Posix compliant (e.g, `tar`)
- Non-option arguments are data passed to the command

[1] Commands may be preceded by a sequence of variable assignments

Command Syntax

```
ls -a -l Labs/Unix Lectures
```

- `ls` – utility, to list contents of a directory
- `-a` – option, to include hidden files (all)
- `-l` – option, spit out long listing
- `Labs/Unix` – argument, directory to list
- `Lectures` – argument, another directory to list

Note, short options which don't require arguments (*optargs*) can generally be stacked:

```
ls -al Labs/Unix Lectures
```

Options, Optargs

Options come in 2 flavors:

- *Toggles*, or flags. On or off
- Options which, in turn, need information

```
tail -f -n30 error.log
```

- `-f` – A toggle, tells `tail` to update (follow)
- `-n 30` – Tells `tail` to display 30 lines

Traversing the Filesystem

- `ls` – lists file or contents of a directory (current directory by default)
  - `-a` – show hidden files (all)
  - `-o, -l` – long (and longer) listing
  - `-d` – directory (don't list out the contents)
  - `-F` – Decorate names depending on filetype
- `pwd` – print the working (current) directory
- `cd` – change directory[1]
  - By default, takes you home

[1] Also see Bash's `pushd` and `popd`

Getting Help – `man`, `info`

Man Pages:

- Get information on any properly installed utility (`ls`, `grep`, etc.)
- Can do a keyword search: `man -k music`
- Split into sections (note them)
- Flat, unexciting, but very useful

Info pages are often provided

- Hierarchical; not flat
- Navigation uses emacs-like bindings
- If no info page, it'll display the man page

Viewing Text Files

- `cat` – *concatenate*, send to stdout. View text files
- `less more` – paging utility. `h` for help, `q` to quit
- `od` – *octal dump*. For viewing raw data in octal, hex, control chars, etc. Useful for looking for non-printing characters in your code.

Copying, Removing, Linking

- `rm` – *remove* file
  - `-r` – *recursive*. Careful, here
  - `-f` – *force*. Ignore nonexistent files
- `cp` – *copy*
  - `-i` – *interactive*. Ask before overwriting destination file (if it exists)
- `mv` – *move*. Also, rename, you can give the file a different name as you move it
  - `-i` – *interactive*. Ask before overwriting destination file (if it exists)

Directories

- `mkdir` – *make directory*
- `rmdir` – *remove directory*
  - Safe; it won't remove non-empty directories
  - Compare to `rm -rf` (and be careful)
- Directories can be moved/renamed using `mv`
- Entire directories can be copied using `cp -r`
  - See `rsync`

Archiving

- `tar` – *tape archive*
  - makes one large file from many smaller files
- `gzip, gunzip` – One (of many) compression utilites
  - `bzip2 compress xz zcat zip`
- `tar` on Linux does gzip compression (and others) using the `z` option:

```
tar czf 571back.tgz CS571
tar xzf assn1.tgz # or .tar.gz
```

Filters

Programs that read some input, perform some transformation, write out the results

- `head`, `tail` – Displays first (last) n lines of input
- `grep` – Search input using *regular expressions*
- `sort` – Sorts input by lines (lexically, or numerically)
- `uniq` – *Unique*, removes identical, adjacent lines
- `wc` – *Word count* (line count, character count)
- `cut` – Select fields of a line
- `tr` – *Translate*

Some Other Utilities

- `date` – Print current date and time
- `time` – Does *not* show you the current time
- `who` – Print who is currently logged in
- `finger` *user* – more information about *user*
- `du -sh` – Disk usage summary, human readable

Files & Permissions

Directories

Every file has some attributes stored by the filesystem

- Times of creation, last change, last modify, last access
- Size
- Owner and group
- Permissions

ACLs

Time Attributes

- `stat` *`file`* shows all of these attributes
- `ls -o` shows the last modification time
- `ls -ot` sorts by modification timej
- See `find`'s `-ctime` `-mtime` `-atime`

ls -l


```
-rw-rw-r-- 1 kschmidt 265-inst 20749 Oct 30 11:37
unix.tex
```

- `-rw-rw-r--` – File type and mode bits
- `1` – Number of hard links
- `kschmidt` – owner
- `265-inst` – group
- `20749` – size (see `-h`)
- `Oct 30 11:37` – Modification time
- `unix.tex` – filename

File Permissions

- Each file has a set of permissions that controls who can do what to the file
  - Note, ACLs are newer, ride on top of these permissions
- There are three types of permissions
  - `r` – read
  - `w` – write
  - `x` – execute
- Permissions are set for these entities
  - user (the file's owner)
  - group (members of the file's group)
  - other (world; everybody else)

Type & Permission Bits

`-rw-rw-r--`

- `-` – Plain file
- `d` – Directory
- `s` – Symbolic link
- `p` – Named pipe 
- `c` – Character file (keyboard, mouse, etc.)
- `b` – Block (disk drives, USB, etc.)

user's permissions

group's permissions

others' permissions

rwx

Files:

- `r` – allowed to read
- `w` – allowed to write
- `x` – allowed to execute

Directories:

- `r` – Can list out the directory (view contents)
- `w` – allowed to create and remove files
- `x` – allowed to "enter" the directory, change to subdirectories, edit files

Changing Permissions – `chmod`

`chmod` *mode(s) file(s)*

- `chmod` command changes permissions on a file or directory
- Modes can be expressed symbolically, or as octal values

```
chmod 755 Public # Typical perms for a public directory or executable
chmod 644 README # Typical perms for a public file
chmod a+x script # Add execute permissions for everybody
```

- `-R` – `chmod` goes recursive

`chmod` – Octal Modes

Consider each set of permission bits as a 3-digit binary number:

- `r` – 4
- `w` – 2
- `x` – 1

A permission (mode) for all three sets is a 3-digit octal number:

- 755 – `rwxr-xr-x`
- 640 – `rw-r-----`
- 711 – `rwx--x--x`

chmod – Examples


```
$ chmod 700 CS571
$ ls -o Personal
```

```
drwx------ 10 kschmidt 4096 Dec 19 2004 CS571/
```

```
$ chmod 755 public_html
$ chmod 644 public_html/index.html
$ ls -ao public_html  # $
```

```
drwxr-xr-x 16 kschmidt 4096 Jan 8 10:15 .
drwx--x--x 92 kschmidt 8192 Jan 8 13:36 ..
-rw-r--r--  5 kschmidt 151 Nov 16 19:18 index.html
```

```
$ chmod 644 .plan
$ ls -o .plan
```

```
-rw-r--r--  5 kschmidt 151 Nov 16 19:18 .plan
```

`chmod` – Symbolic Modes

Can modify (add or remove) permissions, or set permissions absolutely

`[ugoa] [+-=] [rwx]`

- `u` – user
- `g` – group
- `o` – other
- `a` – all

- `+` – add permission(s)
- `-` – remove permission(s)
- `=` – set permission(s)

chmod – Examples


```
$ ls -al foo
```

```
-rwxrwx--x  1 hollingd grads foo
```

```
$ chmod g-wx foo
$ ls -al foo
```

```
-rwxr----x  1 hollingd grads foo
```

```
$ chmod u-r .
$ ls
```

```
ls: .: Permission denied
```

Shell–Bash

Shell as a User Interface

- A shell is a command interpreter
- Interface between a human (or another program) and the OS
  - Runs a program (say, `ls`, or a Solitaire game or Web browser)
  - Can establish alternative sources of input and destinations for output of programs
- Is, itself, just another program

Shell as a Scripting Language

Has features commonly found in languages for structured programs

- Allow shell scripts to be used as filters
- Control flow, variables
- Control over all I/O file descriptors
- Control over signal handling
- The environment allows context to be established at startup
  - Provides a way for scripts to pass information to processes w/out using positional parameters

Bourne Again Shell (bash)

We'll teach Bash in this course

- Extension of the Bourne Shell
- Contains many of the Korn Shell (ksh) extensions
- There are other shells: tcsh (Tenex C Shell), ksh (Korn Shell), zsh, dash

`bash` Customisation

- The shell supports various customisations
- Set through shell options or evnironment variables
  - User prompt
  - Bindings for command-line editing
  - Aliases (shortcuts)
  - Functions – like little scripts[1]
  - Other behaviors

[1] But they run in the *current* shell

`bash` startup files

- Place customisations in *startup* files
  - `/etc/profile` – system-wide
  - `/etc/bash.bashrc` – system-wide
  - `~/.bash_profile` – user
  - `~/.bashrc` – user
- Read the Bash manpages to see when each is invoked

Interpreting Commands

- Shell prints a prompt, awaits a command
- When the shell gets a line of input
  1. It expands aliases (recursively)
  2. Checks to see if command is a shell *builtin* (or a function)
  3. If not, assumes it is a disk utility (e.g., `ls`)

Shell Builtins

- A *shell builtin* is a command the shell will do for you
  - `cd`, `type`, `pushd`, `set`, `pwd`, ...
- They are faster
- The shell provides builtins for some common disk utilities
  - `echo`, `printf`, `test`
  - Use a path to invoke the disk utility (`/bin/echo`)
- The builtin `type` will determine if a command is a builtin, or tell you where the utility is on disk
- The `help` builtin will give you help on any builtin, or show you all of the the shell builtins

Running Programs from Disk

- Disk programs are run in a subshell
- The subshell *execs* the program
  - Replaces itself with the program
- If the command isn't a shell builtin, the shell will search for a disk utility (using your `$PATH`)
- If the command token contains a path, then that utility will simply be run

```
$ /usr/bin/firefox & # kick firefox off in the background
$ /usr/bin/python myScript.py # invoke the python interpreter
$ ~/bin/cow-sample # Invoke my script to see cows
$ ./a.out # run a program I just compiled, in this directory
```

Logging Off

Use the `exit` builtin

- Exits the shell
- If it is the login (top-level) shell, then it disconnects you
- A shell is just another program
- Can recursively invoke shells
- Don't just disconnect w/out exiting
- `ctrl-D` (end-of-file) will also log you out
  - Unless you have the `ignoreeof` shell option set

Standard I/O

- Shell manages I/O
  - Programs and scripts run in a subshell
- The shell establishes 3 I/O *channels*:
  - `stdin`, file descriptor 0, default is the keyboard
  - `stdout`, file descriptor 1, default is the screen
  - `stderr`, file descriptor 2, default is the screen
- These *streams* may be redirected to or from another file
- Can also be redirected to or from another process

Terminating Input

- `stdin` is read like any other file[1]
- If `stdin` is the keyboard, use Ctrl-D (`^D`) to signal EOF
- Many utilites, filters, will read `stdin` if not given a filename(s) to open
  - `cat head grep awk sort` ...
- If it appears a program "isn't doing anything", it's possible that it's waiting on you

```
$ grep the   # no filenme
What's this?
Is this the line?
Is this the line?
That's not funny.
Maybe there should be a law
Maybe there should be a law
^D  # ctrl-D, EOF
```

[1] Sorta. You can't back up

Shell Metacharacters

A *metacharacter* is a characters which has special meaning to the shell

- Wildcards

  `* ? [ ]`

- I/O redirection

  `< > |`

- Others

  `& $ # ! \ ( )`

These characters must be escaped or quoted to inhibit their special behavior

```
$ ls "some file" another\&file 'and;yet;a;third'
```

```
some file another&file and;yet;a;third
```

Wildcards

Also known as *name globbing* and *filename expansion*

- `*` – matches 0 or more characters
- `?` – matches exactly 1 character
- `[`*list*`]` – matches any single character from *list*
- Wildcards are *not* regular expressions

E.g:

`ls *.cc` – list all C++ source files in directory

`ls ?a*` – list all files whose 2nd letter is 'a'

`ls [a-cf]*`.jpeg – list all JPEGs that start with a, b, c, or f

`ls [!ac-e]*`.jpeg – list all JPEGs that do *not* start with a, c, d, e

`ls *` – Try it with non-empty subdirectories present

Shell Variables

Called *paramters*

- Bash uses shell variables to store information
- Used to affect the behavior of the shell, and other programs
- Simple mechanism, just stores text
- Bash does have arrays and associative arrays (see `declare` builtin)

Setting & Viewing Parameters

- To assign a variable (in `sh`, `ksh`, `bash`)
  - Note, *no* whitespace around the =

```
VAR=something
OTHER_VAR="I have whitespace"
```

- Precede with $ to view (dereference) a parameters:

```
$ echo $OTHER_VAR
```

```
I have whitespace
```

```
$ echo "My name is $USER"
```

```
My name is kschmidt
```

Common Parameters

- `PATH` – list of directories searched by shell for disk utilities
- `PS1` – primary prompt
- `USER` – user's login name
- `HOME` – user's home directory
- `PWD` – current working directory

Other Useful Shell Variables

- `SHELL` – The *login* shell
- `$$` – The PID of the current shell
- `$?` – The return value of the last command
- `TERM` – Terminal type (what the shell thinks the terminal interface is)
- `HOSTNAME` – Machine's hostname (see `uname`)
- `EDITOR` – Some programs (`mutt`, `sudoedit`, `git`, etc.) might look here, when opening a text file
- `SHELLOPTS` – Status of various Bash options (see the `set` builtin)

Bash `set` Command

- The `set` builtin with no args displays all shell variables and functions
- Can be used to set various options. E.g.,
  - `-o noclober` – Won't let re-direct overwrite an existing file
  - `-o ignoreeof` – Shell won't exit on `^D`
  - `-o vi` – Use vi-like keybindings for editing the command line. `emacs` is the default
  - `-n` – Dry run. Just parse, but don't execute. Handy for debugging scripts
  - `-x` – Echo on. Shows commands in script as they execute

\ – The Escape Character

Use the backslash to inhibit the special meaning (behavior) of the metacharacter that follows.

```
$ echo $USER
```

```
kschmidt
```

```
$ echo \$USER
```

```
$USER
```

So, now \ is a metacharacter. Escape it to get just the character:

```
$ echo a\\b
```

```
a\b
```

\ Followed by Newline

The backslash, when followed immediately by a newline, effectively removes the newline from the stream

```
$ echo On the bloody morning after\
> One tin soldier rides away
```

```
On the bloody morning afterOne tin soldier rides away
```

Use quotes, if you want the newline in the output:

```
$ echo "On the bloody morning after
> One tin soldier rides away"
```

```
On the bloody morning after
One tin soldier rides away
```

Weak Quoting

Double quotes inhibit all but \ ‘ $!

```
$ echo "$USER is $USER"
```

```
kschmidt is kschmidt
```

```
$ echo "\$USER is $USER"
```

```
$USER is kschmidt
```

```
$ echo "I said, \"Well, we shan't\""
```

```
I said, "Well, we shan't"
```

```
$ echo "It is now $(date '+%H:%M')"
```

```
It is now 19:27
```

Strong Quoting

Single quotes preserve the literal value of all enclosed characters

- May not contain a single quote (can't be escaped)

```
$ echo 'I said, "Wait!"'
```

```
I said, "Wait!"
```

```
$ echo 'My name is $USER'
```

```
My name is $USER
```

Redirecting I/O

The shell can read `stdin` from sources other than your keyboard

- Input can be redirected from a file
- Input can even be taken from the output of another process, though a *pipe*

Similarly, `stdout` (and `stderr`) can go places other than your screen

- Redirected to a file
- Piped to another process to read as input

Redirecting `stdout`

- `stdout` is file descriptor 1
- Use `>` after a command (and its arguments) to redirect the output to a file:

```
$ ls > list.out
```

- If `list.out` previously existed it will be truncated (gone)
- Use `>>` to append the output to the file.

```
$ ls >> list.out
```

Using `echo` to Write Files

- `echo` (builtin, and disk utility) writes a line to `stdout`
  - `-n` suppresses the newline
  - `-e` permits expansion of escape chars (`\t`, `\n`, etc.)
- The `printf` utility is handy for formatting output

```
$ idx=127
$ echo "First line" > "$logfile"
$ echo "Another line" >> "$logfile"
$ printf '%-15s formatted line %5x\n' "$USER" $idx >> "$logfile"
$ cat "$logfile"
```

```
First line
Another line
kschmidt        formated line   7f
```

Create Files with `cat`

- `cat`, in the absence of command-line args, reads from `stdin`, writes to `stdout`
- We can use this to write to a file
  - Use `^D` (Ctrl-D) to end input

```
$ cat > "$ofile"
This is line one
Another line

Okay, that's enough
^D
```

- Handy way to concatenate files:

```
$ cat part1 > result
$ cat part2 >> result
```

Redirecting `stderr`

`stderr` is file descriptor 2, so:

```
$ gcc buggy.c 2> error.log
$ grep '[Vv]era' *.html > results 2> error.log
```

To send both to the same place:

```
$ find . -name 'core*' > core.list 2>&1
```

- Note, the order matters
- Bash has syntactic sugar for this move:

```
$ find . -name 'core*' &> core.list
```

Redirecting `stdin`

- < redirects `stdin` from a file
  - File descriptor 0

```
$ sort < nums
$ mail -s"Meaningful subject" $id < msg
```

- You can do both

```
$ sort < nums > sortednums 2> sort.errors
$ tr 'a-z' 'n-za-m' < code.rot13 > decoded
```

- Note, the order matters

Here Documents/Strings

- *Here documents* are helpful in scripts
- Input is redirected using `<< [-]WORD`
  - *WORD* signals end of input
- We'll examine these further in a subsequent lecture

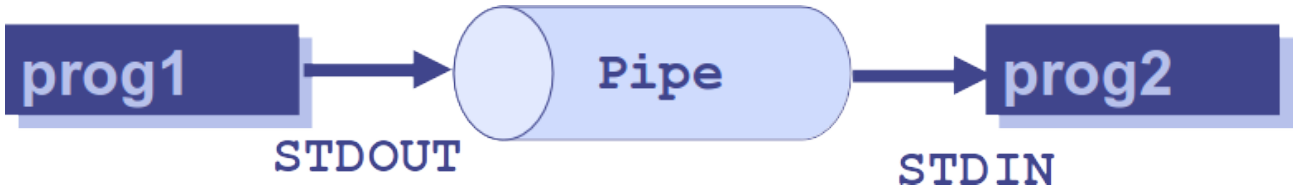
```
cat << EOS
Dear $NAME:
I am writing this slowly, since I know you can't read fast.

It was so windy here Tuesday the chicken laid the same egg
$EGG_CNT times.
EOS
```

- *Here strings* are convenient on the command line:

```
$ bc -l <<< "$x + s($d)"
```

Unnamed Pipes

- A redirector links a process to a file
- A *pipe* links a process to a *process*
- It's a stream of data



- Data written to `stdout` by *prog*1 is read on `stdin` by *prog2*
- *Much* faster than writing, then reading, intermediate files

Asking for a Pipe

- Separate 2 commands with |
- The shell does all the work

```
$ du -s * | sort -n
$ du -s * | sort -n > sorted.lst
```

- Processes can be strung together with pipes:

```
$ du -s * | sort -nr | head -n10 > 10_bigest_files
```

The Unix Phlosophy

The use of pipes and other features to combine "small, sharp tools" to accomplish larger tasks
– Ken Thompson (father of Unix)

"...at its heart is the idea that the power of a system comes more from the relationship among programs than from the programs themselves."
– Brian Kernighan & Rob Pike

"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."***
– Doug McIlroy (inventor of the Unix pipe)

Processes, Jobs

Process Control

- Processes are run in a subshell
- Subshells inherit exported environment
- Each process has an ID (PID) and a parent (PPID)
- Use the `ps` utility to look at processes:

```
$ ps
```

```
  PID TTY          TIME CMD
  350 pts/4    00:00:00 bash
22251 pts/4    00:00:00 vim
22300 pts/4    00:00:00 ps
```

Process Control (cont.)

- Use the `-f` option for a long (full) listing

```
$ ps -f
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
kschmidt   350   349  0 10:06 pts/4    00:00:00 -bash
kschmidt 22251   350  0 17:32 pts/4    00:00:00 vim myHomework
kschmidt 22437   350  0 17:36 pts/4    00:00:00 ps -f
```

- Use the `-e` option to see *all* of the processes (not just yours)

Killing a process

- The `kill` (built-in and utility) sends a signal to a process
  - By default, sends the `SIGTERM` (signal 15)
  - Send `SIGKILL` (9), won't be ignored, but, no cleanup
- To kill a process using its PID:

```
$ kill 29940
$ kill -n9 29940 # if it ignored your previous request
```

- See also `pgrep` and `pkill`

Job Control

The shell allows you to manage jobs

- Place a job in the background
- Move a job to the foreground
- Suspend a job
- Kill a job
- Use `jobs` to view current jobs (in a given shell)

```
$ jobs
```

```
[2]  Running evince unix.pdf & (wd: ~/CS265/Lectures/Unix)
[4]  Running gimp & (wd: ~/public_html/CS265/Lectures/Unix)
[6]- Running soffice CS265/Lectures/Unix/intro.ppt & (wd: ~)
[7]+ Stopped vi hello.tex
```

Job Control

- When a process is running, the shell is blocked
- So, we run, e.g., GUI programs in the background
- Processes that might take a while we can place in the background
- Place a `&` after a command to run it in the background:

```
$ firefox &
$ evince unix.pdf &
$ find ~/ -type f -mtime -1 > find.out # this might run a while
$    # Save output to a file
```

Suspending and Resuming a Process

- Use `^Z` to suspend the process in the foreground
- Use `fg` to bring the most recent process back to the foreground
  - Working in an editor, I'll save, `^Z` out, compile, then `fg` back to my editing session
- Or, type `%`*n*, where *n* is the index, from the `job` listing
- Use `bg` to put the most recently suspended process into the background

```
$ evince unix.pdf # Whoops! Forgot to put in background
^Z  # Suspend evince
$ bg   # Set it running in the background
```

Killing a job

- You can kill a job much as you might a process
- `SIGTERM` is often like closing the window, or choosing "Quit"
  - `SIGKILL`, on the other hand, can't actually be trapped
  - The plug will be pulled on the process, no chance to clean up
  - Not really nice
- Specify a job using %:

```
$ kill %4      # Give it a chance to exit itself
$ kill -n9 %4  # Just pull the plug
```

Editors

Editors

- In this course you will use either `emacs` or `vim`
- It is well worth learning a good, richly-featured editor
  - Syntax highlighting
  - Regular expression search and replace
  - Keyboard navigation
  - Extensible through macros
  - Much more
- GUI versions of `emacs` and `vim` exist
- Take the time to learn navigation, w/out the mouse and the arrows
  - You won't always have a GUI running
  - After a bit of practice, the mouse simply slows you down

emacs vs. vim

I, for good or ill, am a VI guy, so, I'll better be able to answer those questions

- vim – Vi IMproved
  - Was the standard Unix editor
  - Built on ed
  - Shares some syntax with sed, and many other utilities, including, amusingly, mutt and cmus, my mp3 player
- emacs is written in (Emacs) LISP
  - A bit more powerful than vim (you can run a shell inside, or, play Tetris)
  - The default editor for Linux

Both are excellent text editors. Both have a steep-ish initial learning curve. Put in the time, learn one!