Introduction to Unix

- A brief history of UNIX
- Why use UNIX?
- A model of the UNIX environment
- The UNIX file system (directories, commands)
- File attributes (permissions)

A brief history of UNIX

- UNIX is a:
  - multi-user, multi-tasking operating system
  - machine-independent operating system ("portable")
- the "UNIX" trademark:
  - owned by AT&T
  - passed to the "Unix System Laboratories" (USL)
  - passed to Novell
  - passed to X/Open Company, Ltd. (1993)
  - X/Open + Open Software Foundation (OSF) → The Open Group
  - The Open Group (1996), http://www.opengroup.org/
- So every manufacturer calls it something else!

A brief history of UNIX (2)

- AT&T / Bells Labs (was Lucent Technologies, now Avaya & Alcatel-Lucent)
  - Unix created by two researchers for their own personal use (Thomson & Ritchie, 1970)
  - academic/research operating system
  - Initially, pros: flexibility, extensibility, file sharing
  - Initially, cons: security, robustness, performance
  - easy to use (in comparison with contemporaneous OSes)
  - the first portable OS where "portable" == “recompilable and executable on another architecture”

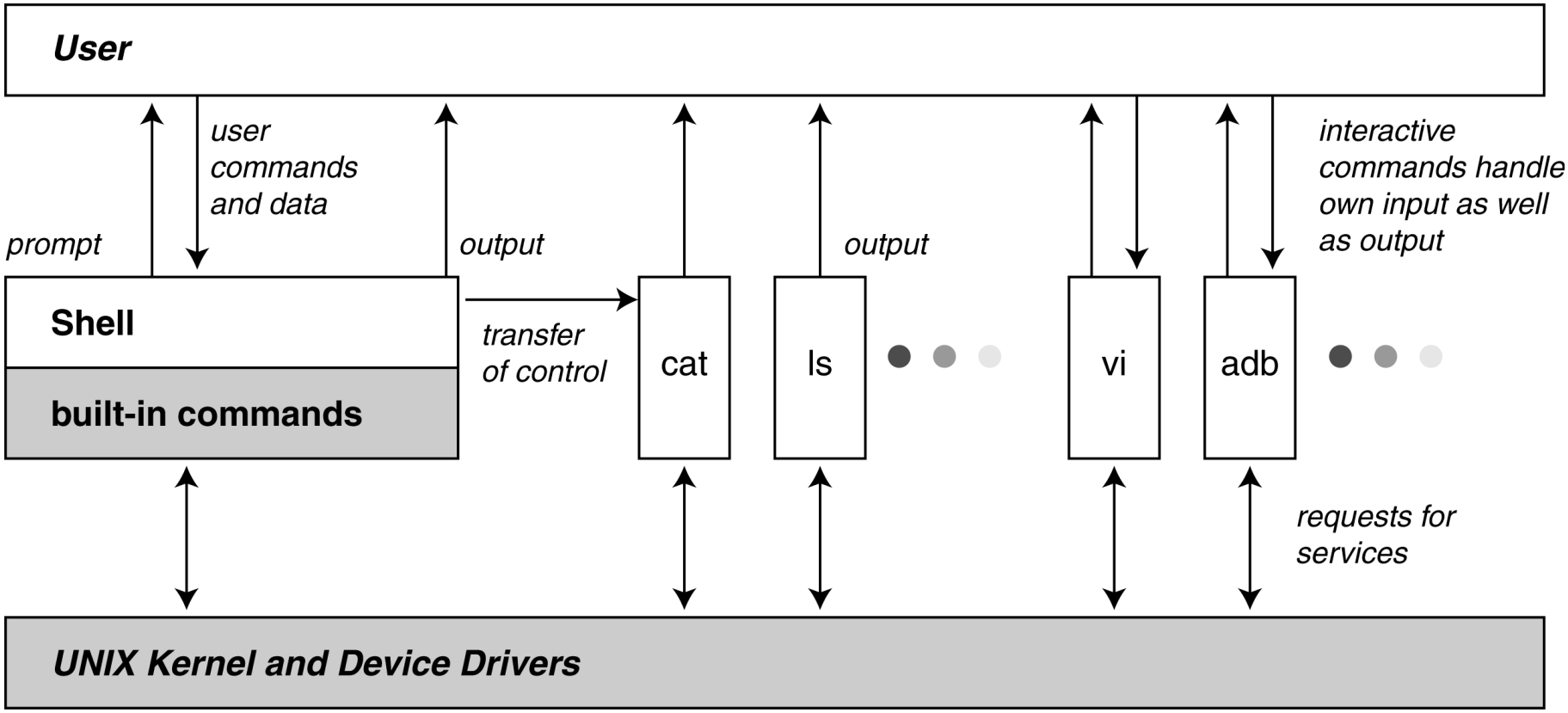
A brief history of UNIX (3)

- Berkeley Standard Distribution (BSD)
  - freeware! (cheap for universities; only paid for distribution cost)
  - first UNIX to include standard network support
  - enhancements to interprocess communication (IPC), job control, security
- many flavours of UNIX in use today:
  - FreeBSD, NetBSD, XENIX, Solaris, SunOS, HP-UX, Linux, A/UX, AIX, Mac OS X
- continues to evolve
  - e.g., Single UNIX Specification (derived from POSIX standard)
- Free Software Foundation and GNU Unix
- Ubuntu – arguably the most popular Linux distribution

Why use UNIX?

- multiuser
- multitasking
- remote processing
- stable (some might even say "safe")
- highly modular
- some versions are free (open→freedom to modify)
- large user community, extensive experience
- “tools are mature”

UNIX model


User
user
commands
and data
prompt
output
output
interactive
commands handle
own input as well
as output
Shell
built-in commands
transfer
of control
cat
ls
vi
adb
requests for
services
UNIX Kernel and Device Drivers

shell

- responsible for communication between the user and kernel
- "shell" refers to its position in a diagram of UNIX structure
  - concentric rings
- reads and interprets user commands at the console (or from within a "shell script")
- implements job control
- many shells have been developed:
  - `sh, csh, ksh, tcsh, zsh, bash` …
  - in this course we use the `bash` shell
  - `bash` extends `sh`, the Bourne shell

kernel

- the kernel is the core of the operating system
- the kernel is itself a large and complex program
- clear demarcation between the “kernel” and a “user”
  - to access a computer’s hardware, user's commands must go through kernel
  - “user” must request the kernel to perform work
  - mediated by a command shell (e.g., **bash**), or the system library (compiled application)
- main responsibilities
  - memory allocation
  - **file system**
  - loads and executes programs (assumes a process model)
  - communication with devices (input, output)
  - bootstraps the system
  - **...**

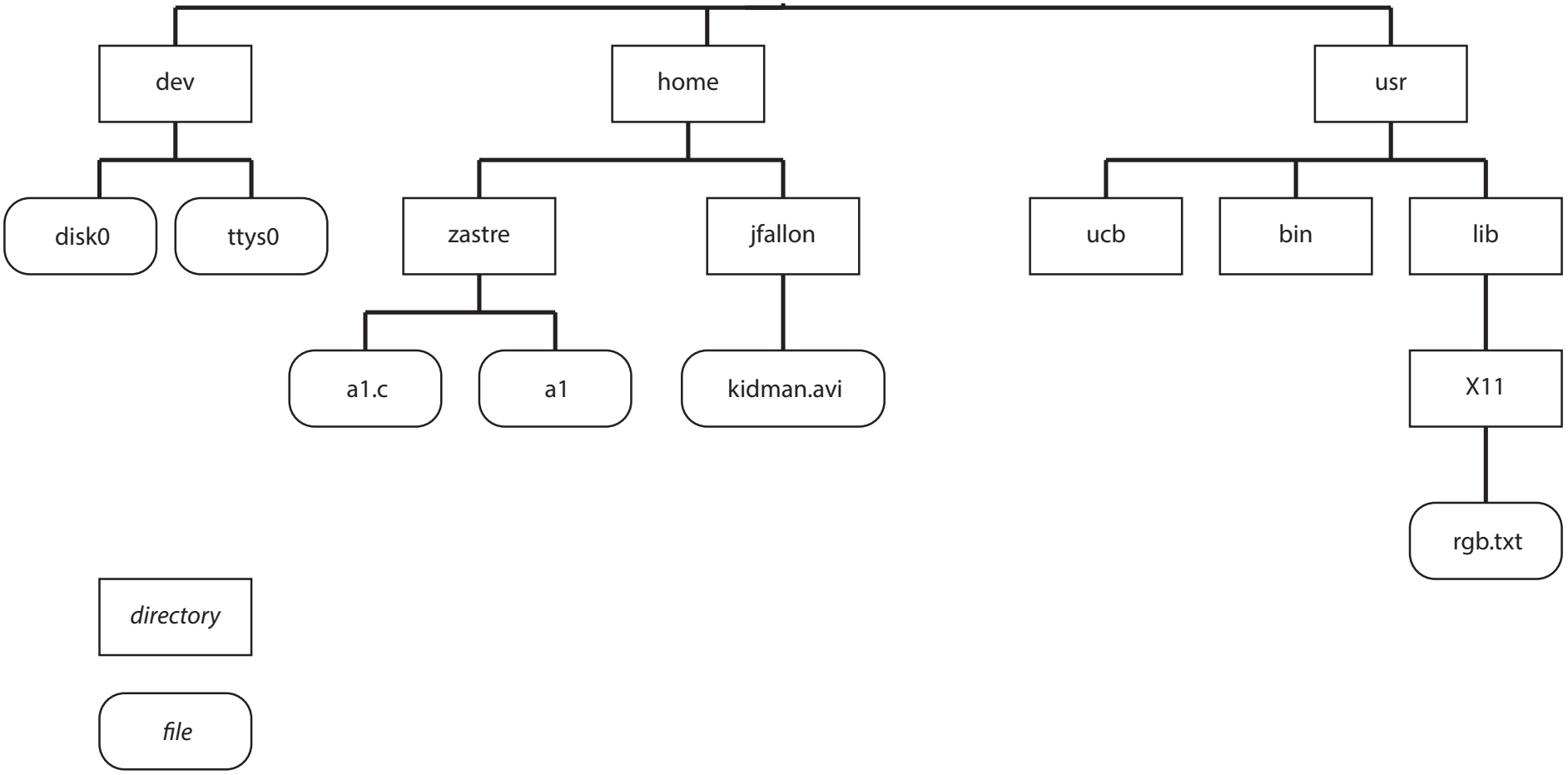
UNIX filesystem

- "file", "filesystem": **are key abstractions** of the UNIX computing model
- practically anything can be abstracted as a file (devices, programs, data, memory, IPC, etc.)
- mainly responsible for providing abstracting blocks of information on **physical** storage device (hard drive, flash memory) into **logical** blocks that user can manipulate
  - maps filenames to block numbers
  - handles block allocations; chains units together
  - provides methods to access data
- facilitates the "multiuser" view of the OS

UNIX filesystem

- arranged as a hierarchy (tree-like structure)
  - organized as a collection of directories; think of a directory as a folder
  - forward slash "/" is used to separate directory and file components (cf., Windows uses "\")
- the root of the filesystem is represented by the **root-directory**, which we denote by a single "/"

Part of a (hypothetical) UNIX filesystem tree



Some properties of directories

- directories are actually ordinary files
- information contained in a directory file simply has a special format
- every directory contains two special directory entries
  - ".." refers to parent directory in hierarchy
  - "." refers to the current directory (itself)
- '~' is used to denote a **home directory**
  - `% cd /home/user` ≈ `cd ~user`
    `% cd` ≈ `cd ~`

Directory commands (ignore %)

- Example:

```
% cd /home
```

- listing directories

```
% ls
zastre jfallon
% ls jfallon
kidman.avi
```

- relative pathnames

```
% cd /home
% open jfallon/kidman.avi
% open ./jfallon/kidman.avi
% open ./jfallon/../jfallon/kidman.avi
```

“working” vs. “home” directory

- “Working” directory is the directory you are “in” at any given time.
  - Eliminates the need to continuously specify full pathnames for files and directories
  - “Relative pathnames” are locations worked out in relation to (relative to) to the current working directory.
- “Home” directory is (usually) made to be your working directory upon logging into the system
  - Sometimes called the “login” directory
  - `/home/zastre` & `/home/seng265` are typical home directories

Directory commands (2)

- traversing directories

```
% cd /usr
% ls
ucb bin lib
```

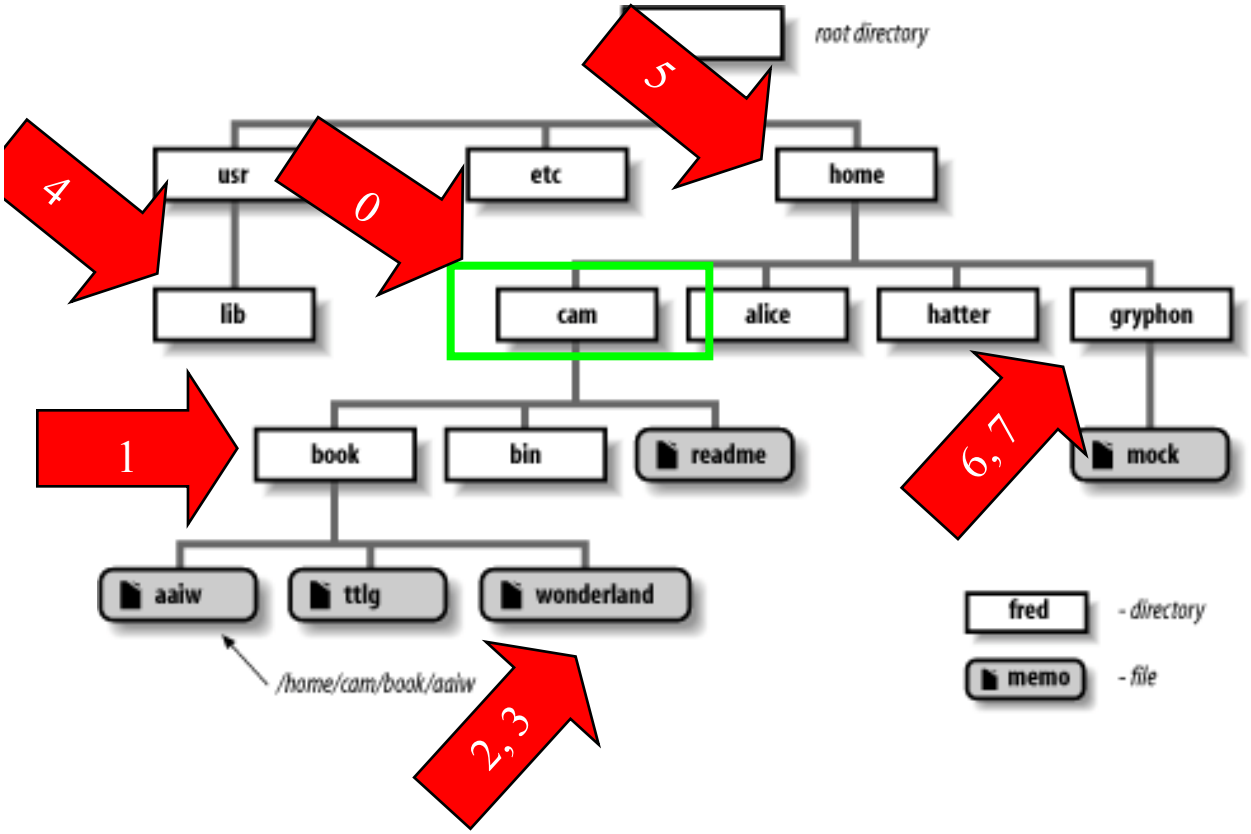
- display the **current working directory**

```
% pwd
/usr
```

- creating a **symbolic reference** to a file (i.e., like an alias"

```
% cd ~zastre
% ln -s a1 sample_solution
% ls
a1   sample_solution
```

working directories




```
# "cam" is the logged-in user
#
# Each of the following commands
# assumes Cam's current directory
# is /home/cam (i.e., every item
# below assumes we reference
# from at red-arrow 0).

% cd book                  #1

% vi book/wonderland       #2

% vi ~/book/wonderland     #3

% cd /usr/lib              #4

% cd ..                    #5

% cd ../gryphon            #6

% cd ~gryphon              #7

% cd alice                 # ??
```

File attributes

- every plain file and directory has a set of **attributes,** including:
  - user name (owner of file)
  - group name (for sharing)
  - file size (bytes)
  - creation time, modification time
  - file type (file, directory, device, link)
  - permissions

```
% ls -l unix.tex test
-rwxr-xr-x   1  joe  users     200  Apr 29  14:39  test
-rw-r--r--   1  dmg  users   21009  Apr 29  14:39  unix.tex
```

Who has permission?

- permissions can be set for
  - user ("u") `[-rwx------]`: the file owner
  - group ("g") `[----rwx---]`: group for sharing
  - other ("o") `[-------rwx]`: any other
  - all ("a"): user + group + other
- **user**: the owner of the file or directory; owner has full control over permissions
- **group**: a group of users can be given shared access to a file
- **other**: any user who is not the owner and does not belong to the group sharing a file

What kind of permissions?

- files:
  - **read (r)** : allows file to be read
  - **write (w)**: allows file to be modified (edit,delete)
  - **execute (x)**: tells UNIX the file is executable
  - **dash (-) :** owner/group/other have no permissions
- directories:
  - **read (r)**: allows directory contents to be read (listed)
  - **write (w)**: allows directory contents to be modified (create, delete)
  - **execute (x)**: allows users to navigate into that directory (e.g, with the **cd** command)
  - **dash (-)** : owner/group/other have no permissions

chmod: set file permissions

- there are several ways to use "chmod"
  - use letter symbols to represent "who" and "what"

```
% chmod o+rx ~/.www/ppt    # other can read and cd "ppt"
% chmod u+x run.pl         # script "run.pl" executable
% chmod go-rwx ~/private   # removing access group & other
% chmod u=rwx,g=rx,o=x  foobar.txt   # all permissions
```

  - can also use "octal" (base 8) notation, representing each three-bit field with an octal digit; ***r*** ∈ {0,4}, ***w*** ∈ {0,2}, ***x*** ∈ {0,1}

```
% chmod 751 foobar.txt      # specify all permissions
```

  - the following are different ways of setting "read-only" permission for a file

```
% chmod =r file
% chmod 444 file
% chmod a-wx,a+r file
```

A slight detour… the octal system

- Uses only 8 distinct counting 'octal numbers' ranging from 0 to 7 (i.e. 0,1,2,3,4,5,6,7)
- Each 'octal digit' represented by groups of 3 bits ∈{0,1}, with each group or set of bits having a distinct value of between 000 ("0") and 111 ("7"=4+2+1)
- Conversion from decimal to octal number

  would look like this:

  Where the octal number for a three bit group 'abc' is:

  $a*2^2 + b*2^1 + c*2^0$, e.g. 111-> $*2^2 + b*2^1 + c*2^0 = 7$
- Hence every three-bit permission configuration (e.g. "r—") for each of the User, Group and Other can be represented with a octal digit, eg.

  "r—" is "100" is 7; "-wx" is "011" is 5, "—x" is "001" is 1

  -> permissions expressed as 751 and

  -> % chmod 751 foobar.txt as shown on previous slide

Decimal number	3-bit binary number	Octal number
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7
8	001 000	10 (1+0)
9	001 001	11 (1+1)
Continuing upwards in groups of 3		

Various & Sundry

- UNIX file names are case-sensitive
  - e.g., `myFile` and `myfile` are two different names, and the `logout` command cannot be typed as `Logout`
- commands are available to change the **owner** and/or **group** of a file; e.g. `chown, chgrp`
- **pager** is a command (`less`, `more`) used to display a text file one page at a time

```
% less unix.txt
```

- quickly create a file (or update the timestamp of an existing file)

```
% touch unix.txt
% ls -l uxix.txt
-rw-r--r--  1 damian users  0  Aug 29 14:39  unix.tex
```

Introduction to UNIX (contd)

- The shell
- Basic command syntax
- Command types
- Getting help on commands
- I/O streams
- Redirection and pipelining
- Command sequences
- Console

the shell (again)

- the shell is the intermediary between you and the UNIX OS kernel
- it interprets your typed commands in order to do what you want
  - the shell reads the next input line
  - it interprets the line (expands arguments, substitutes aliases, etc.)
  - performs action
- there are two families of shells:
  - "sh" based shells, and "csh" based shells
  - they vary slightly in their syntax and functionality
  - we'll use "bash", the Bourne Again SHell (derivative of "sh", known as the "Bourne shell")
  - tip: you can find out what shell you are using by typing:
    `echo $SHELL`

basic command syntax

```
% cmd [options] [arguments]
```

- `cmd` is a builtin-shell or UNIX command
- `[options] = option*`
- `[arguments] = argument*`

option	*example*
`opt`	`a`
`-opt`	`-v`
`--optname`	`--verbose`
`-opt arg`	`-s 5`
`--optname arg`	`--size 5`

basic command syntax (2)

- `opt` is a character in `{a..zA..Z0..9}`
- `optname` is an option name; e.g., `--size, --keep`
- `argument, arg` is one of the following:
  - file name
  - directory name
  - device name, e.g., `/dev/hdb2`
  - number, e.g., `10, 010, 0x1af, …`
  - string, e.g., `"*.c", "Initial release", …`
  - `…`

command types

- commands can be:
  - built into the shell (e.g., `cd, alias, bg, set,`…)
  - aliases created by the user or on behalf of the user (e.g., `rm='rm -i', cp='cp -i', vi='vim'`)
  - an executable file
    - binary (compiled from source code)
    - script (system-parsed text file)
- Use the `type` command to determine if a command is builtin, an alias, or an executable.

```
% type rm
rm is aliased to ‘rm -I’
```

some simple commands

`% cat [file1 file2 ...]`

- (catenate) copy the files to stdout, in order listed

`% less [filename]`

- browse through a file, one screenful at a time

`% date`

- displays current date and time

`% wc [filename]`

- (word count) counts the number of lines, words and characters in the input

`% clear`

getting help on commands

- You can ask for help in several ways.
- Display a description of a shell command (builtin)

```
% help times
```

- Display a long description of a command (from section ***n*** of manual)

```
% man [n] chmod
```

- Display a one line description of a command

```
% whatis gcc
gcc   gcc (1) - GNU project C and C++ compiler
```

- Use "info"

```
% info gcc
% info ls
```

- Many commands provide their own help

```
% somecmd -h
% somecmd --help
```

input & output streams

- each UNIX program has access to three I/O "streams" when it runs:
  - **standard input** or **stdin;** defaults to the console keyboard
  - **standard output** or **stdout;** defaults to the console screen
  - **standard error** or **stderr;** defaults to the console screen
- the shell provides a mechanism for overriding this default behaviour (**stream redirection**)

stream redirection

- redirection allows you to:
  - take input from a file
  - save command output to a file
- redirecting from/to files using bash shell:
  - stdin:

```
% cmd < file
% less < ls.1
```

  - stdout:

```
% cmd > file                          # write
% ls -la >dir.listing
% cmd >> file                                  # append
% ls -la /home >>dir.listing
```

  - stderr:

```
% cmd 2> file                                  # write
% cmd 2>> file                                 # append
```

stream redirection (2)

- redirecting stdin and stdout simultaneously

```
% cmd < infile > outfile
% sort < unsorted.data > sorted.data
```

- redirecting stdout and stderr simultaneously

```
% cmd >& file
% grep 'hello' program.c >& hello.txt
% cmd 1>out.log 2>err.log
```

- UNIX gotchas:
  - symbols used for redirection depend on shell you are using
  - our work will be with the Bash shell (`bash,sh`)
  - slight differences from C-shell's (`csh,tcsh`)

Introduction to UNIX (contd)

- **Class Announcements**

Class Announcements

- **(revised) Office hours:**
  - **W 10:30-11:30 am (revised)**
  - **R 9:00-10:00 am (same as before)**

  - **Revised Midterm date: Oct 23 (Monday)**

Introduction to UNIX (contd)

- More on **File Permissions**
- **Pipes** and command sequencing
- **bash** command history
- readline command editing (**vim**)
- Job control
- Shell/environment variables
- Customizing your shell

pipes

- Pipes are considered by many to be one of the major Unix-shell innovations
  - excellent tool for creating powerful commands from simpler components,
  - does so in an effective, efficient way.
- Pipes route standard output of one command into the standard input of another command
- Allows us to build complex commands using a set of simple commands
- Motivation:
  - without pipes, lots of temporary files result

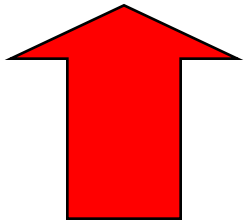
without pipes

- Example: How many different users are currently running processes on the server?

without pipes

- Example: How many different users are currently running processes on the server?

```
% ps aux > temp1.txt
% awk '{ print $1 }' temp1.txt > temp2.txt
% sort temp2.txt > temp3.txt
% uniq temp3.txt > temp4.txt
% wc -l < temp4.txt > temp5.txt
% cat temp5.txt
```



Off by one – need to mentally subtract one from the resulting number

with pipes

- Example: How many different users are currently running processes on the server?

```
% ps aux | awk '{ print $1 }' | sort | uniq | wc -l
```

- Note the structure of the command:
  - "generator" command is at the head
  - successive "filter" commands transform the results
  - this is a very popular style of Unix usage

A bit more about pipes

- Pipes can save time by eliminating the need for intermediate files
- Pipes can be arbitrarily long and complex
- All commands are executed **concurrently**
- If any processing error occurs in a pipeline, the whole pipeline fails

command sequencing

- multiple commands can be executed sequentially, that is: `cmd1;cmd2;cmd3;…;cmdn`

```
% date; who; pwd
```

- may group sequenced commands together and redirect output

```
% (date; who; pwd) > logfile
```

- note that the last line does not have the same effect as:

```
% date; who; pwd > logfile
```

filename expansion

- "shorthand" for referencing multiple **existing** files on a command line

`*`	any number of characters
`?`	exactly one of any character
`[abc]`	any character in the set `[abc]`
`[!abc]`	any character **not** in the set `[abc]`

- these can be combined together as seen on the next slide

filename expansion (2)

- examples:
  - count lines in all files having the suffix ".c"

```
% wc -l *.c
```

  - list detailed information about all files with a single character file extension

```
% ls -l *.?
```

  - send to the printer all files with names beginning in Chap* and chap* files

```
% lpr [Cc]hap*
```

filename expansion (3)

- * matches any sequence of characters (except those with an initial period)

```
% rm *.o          # remove all files ending in '.o'
% rm *            # remove all files in directory
% rm ../*-old*.c
```

- ? matches any single character (except an initial period)

```
% rm test.?       # remove test.c and test.o (etc.)
```

- So to delete a file of the form ".filename" you can't use wildcards

```
% rm .viminfo
```

How do we delete a file named *?

quoting

- controls bash's interpretation of certain characters
- what if you wanted to pass **'>'** as an argument to a command?
- **strong quotes** – All characters inside a pair of single quotes (') are preserved.
- **weak quotes** – Some characters ($,`) inside a pair of double quotes (") are expanded (interpreted) by the shell.
- **backquotes** – substitute result of evaluation as a command

quoting

```
% echo $SHELL *
/bin/bash file1 file2 file3

% echo '$SHELL' '*'
$SHELL *

% echo "$SHELL" "*"
/bin/bash *
```

```
%echo `date`
Thu 12 Sep 2013 14:36:59 PDT
```

backslash escaping

- Characters used by **bash** which may need to be escaped:

  ~, `, #, $, &, *, (,), \, [,], {, }, ;, ', ", <, >, /, ?, !

- single characters can be "protected" from expansion by prefixing with a backslash ("\")

  `cmd \*` is the same as typing `cmd '*'`

- protecting special characters in such a manner is an example of **backslash escaping**

```
% cp ~bob/junk \* # make copy of junk named '*'
% rm '*'                 # remove '*' (not "delete all files")
```

command substitution

- backquotes (`) are used to substitute the result of evaluating a command into a quoted string or shell variable:
  - `% echo "Current date is: `date` "`
    `Current date is: Thu 15 Sep 2011 14:58:13 PDT`
  - `% BOOTTIME=`date``
    `% echo $BOOTTIME`
    `Thu 15 Sep 2011 14:58:25 PDT`

bash command history

- bash (and other shells like sh, tcsh, ksh, csh) maintain a history of executed commands
- uses the readline editing interface
- history will show list of recent commands
  - `% history   # print your entire history`
  - `% history n # print most recent n commands`
  - `% history -c      # delete your history`
- a common default size of the history is 500 commands
  - and the history is usually remembered across login sessions

Using history

- simple way: use up and down arrows
- using the "!" history expansion

  `% !!` repeat last command

  `% !n` repeat command number n

  `% !-n` repeat the command typed n commands ago

  `% !foo` last command that started with foo

readline editing interface

- command-line editing interface
- provides editing and text manipulation
- includes two default modes: `emacs` or `vi`
- select editing mode using either `set -o vi` or `set -o emacs`
- `vi` mode (enter editing mode by pressing `ESC` key):
  - 0 (zero) go to beginning of line
  - d$ erase from cursor to end of line
  - w advance one word
  - b go back one word
  - etc.
- you can customize keystrokes

shell variables

- a running shell carries with it a **dictionary** of variables with values
- some are **built in** and some are **user defined**
- used to customize the shell
- use `env` to display the values of your **environment variables**

```
% env
PWD=/home/bgates
GS_FONTPATH=/usr/local/fonts/type1
XAUTHORITY=/home/dtrump/.Xauthority
TERM=ansi
 HOSTNAME=c70
...
```

shell variables (2)

- many variables are automatically assigned values at login time
- variables may be re-assigned values at the shell prompt
- new variables may be added, and variables can be discarded
- assigning or creating a variable (var):

```
% var="value"
```

- to delete a variable:

```
% unset var
```

- To use the value of a shell variable use the `$` prefix:

```
% echo $PATH
```

PATH shell variable

- helps the shell find the commands you want to execute
- its value is a list of directories separated by ':' symbol
- when we intend to run a program, the directory of its executable should be in the PATH in order to be found quickly
- Example: assume that program `cmd` is located in directory "/`usr2`/bin"

```
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc
% cmd
 bash: cmd: command not found
% PATH="$PATH:/usr2/bin"
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc:/usr2/bin
% cmd
(... now runs ...)
```

- the shell searches sequentially in the order directories are listed

customizing the shell

- In your accounts there will be two files you can modify to customize the `bash` shell:
  - "`~/.bash_profile`" is evaluated by the shell each time you login to your account.
  - by default, "`~/.bash_profile`" **sources** (reads and evaluates) a second file "`~/.bashrc`"
  - **conventional wisdom** suggests that permanent shell/environment variables should be placed in "`~/.bash_profile`", and aliases should be placed in "`~/.bashrc`"
  - system administrators, for very sound reasons, often prefer that we don't modify "`~/.bash_profile`", but instead customize the shell by modifying "`~/.bashrc`" (adding shell variables, aliasing, etc.)
  - In both cases, the changes you make to these files will not take effect until you source the modified file

```
% source .bashrc
```

job control

- the shell allows you to execute multiple programs in parallel
- starting a program in the background ...

```
% cmd &
[1] 3141          # (jobid=1,pid=3141)
```

... and bringing it to the foreground

```
% fg %1
```

- placing a running program in the background

```
% cmd
      ^Z
% bg %1
```

job control (2)

- stopping and restarting a program:

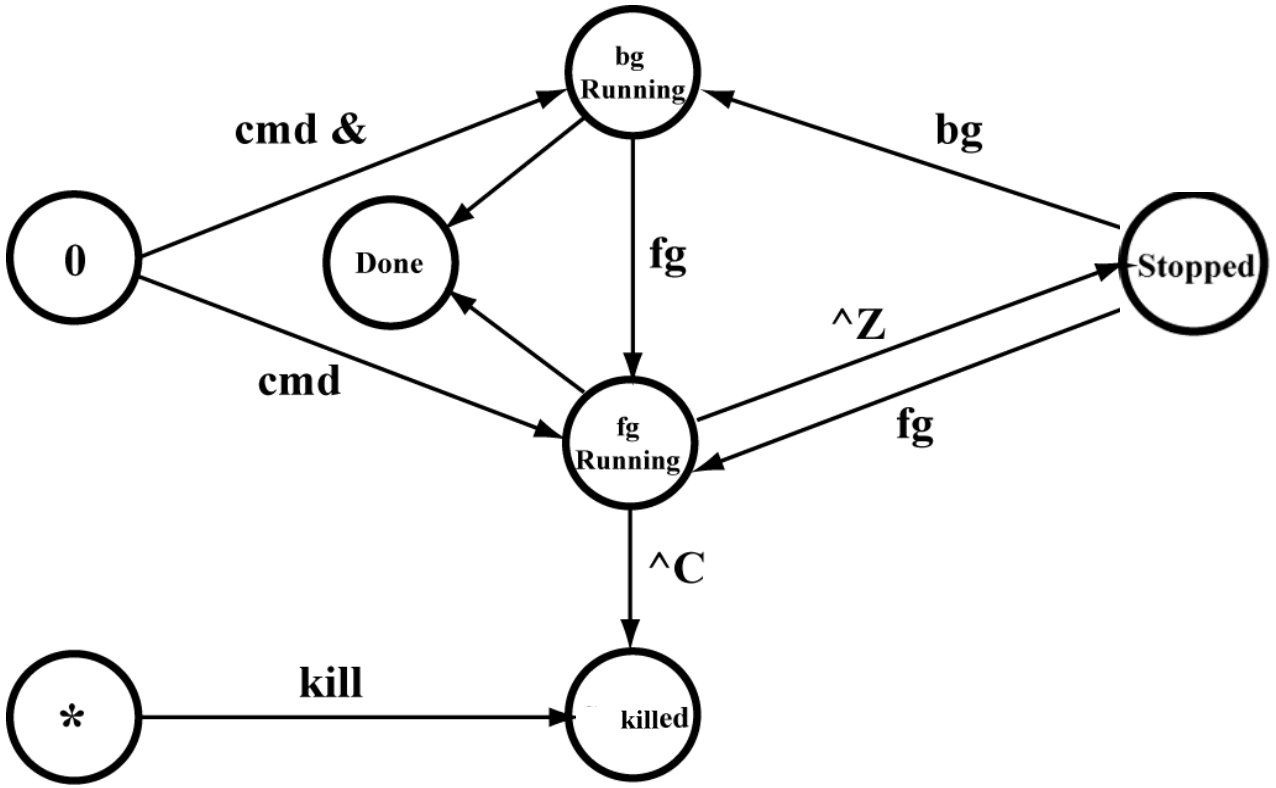
```
% vim hugeprog.c
^Z
[1]+ Stopped
% jobs
[1]+ Stopped  vim hugeprog.c
% gcc hugeprog.c -o hugeprog &
[2] 2435
% jobs
[1]- Stopped  vim hugeprog.c
[2]+ Stopped  gcc hugeprog.c -o hugeprog
% fg %1
[1]  vim hugeprog.c
```

- terminating (or “killing”) a job:

```
% kill %n  # use  kill -9 %n  if the job won't die!
% kill %cc # kill job that starts with cc
```

job control (3)

- job states



endnotes

- This was a brief introduction to UNIX with a heavy emphasis on the use of the "bash" shell
- you should try out the concepts presented in these slides
- you should read man pages and/or other sources of information
  - books
  - online resources
- you can learn from others
  - rarely is there a single way to do the same thing
  - especially true when constructing large commands using pipes
- Slides on bash shell scripting based on lectures by Kurt Schmidt (Drexel University)