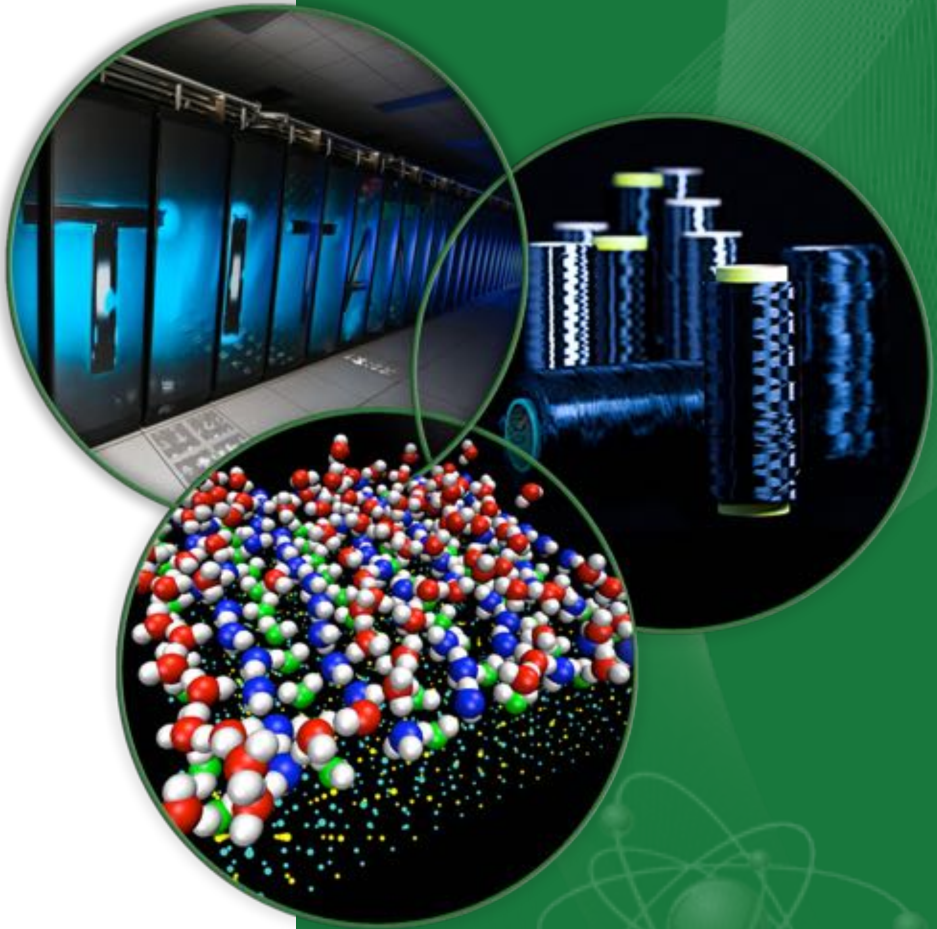
Introduction to *nix

Bill Renaud, OLCF



ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Background

- UNIX operating system was developed in 1969 by Ken Thompson and Dennis Ritchie
- Many “UNIX-like” OSes developed over the years
- UNIX® is now a trademark of The Open Group, which maintains the Single UNIX Specification
- Linux developed by Linus Torvalds in 1991
- GNU Project started by Richard Stallman in 1983 w/aim to provide free, UNIX-compatible OS
- Many of the world’s most powerful computers use Linux kernel + software from the GNU Project

References:
[1]www.opengroup.org/unix
[2]https://en.wikipedia.org/wiki/Linux
[3]https://www.gnu.org/gnu/about-gnu.html

This Presentation

- This presentation will focus on using *nix operating systems as a non-privileged user in an HPC environment
  - Assumes you're using a 'remote' system
  - No info on printing, mounting disks, etc.
- We'll focus on systems using the Linux kernel + system software from the GNU project since that's so prevalent
- Two-Part
  - Basics: general information, commands
  - Advanced: advanced commands, putting it all together, scripts, etc.

This Presentation

- May seem a bit disjoint at first
  - Several basic concepts that don’t flow naturally but later topics build upon
  - Hopefully everything will come together, but if not…
- I’ll cover (what I hope is) some useful info but can’t cover it all
  - People write thousand-page books on this, after all
- Please ask questions!

Basics

Terminology

- **User** – An entity that interacts with the computer. Typically a person but could also be for an automated task.
- **Group** – A collection of 1 or more users. Used for sharing files, permissions to do certain tasks, etc.
- **Shell** – A program that interfaces between the user and the kernel
- **Kernel** – The "main" OS program that's responsible for running the system

More Terminology

- **File** – A collection of data
  - Input/output, a program, etc.
- **Directory** – A logical structure to help organize files (think “folder”)
- **Filesystem** – A collection of files and directories
  - Context dependent-could mean full storage hierarchy, could mean a subset of that
  - Kernel maps physical hardware into the filesystem
  - End users typically deal w/the filesystem, not with physical storage media

Even More Terminology

- **Process** – A program running on the system
  - Every process is associated with a user and a group
  - Processes include programs built & run by users as well as commands provided by the OS or shell
- **Shell Script** – A file containing a list of commands to run
  - Similar syntax is used to launch a shell script & a program
  - Difference is “what” they do: run a series of commands vs. perform some novel calculation
- **Executable** – A common term for a compiled program that can be run (i.e. executed) by a user

Least User Privilege

- The OS operates on the principle of Least User Privilege
  - Gives the user the ability to do what he/she needs to do but limits the ability to affect other users, configure the system, etc.
  - By limiting administrative access, rogue processes are limited in what they can affect
- The root user is unrestricted/has full access to everything
  - NEVER use root for day-to-day work; only for tasks where absolutely necessary
  - `sudo` command can help here

Working Without a Net

- The system assumes you meant to type that
  - If you issue a command to delete all of your files, it will happily do so without asking for confirmation
  - Proofread before you press enter
    There's nothing quite like the feeling "Wow, that's taking a long time"
  - Be sure to back up important files
- Normally, the system only tells you when things fail

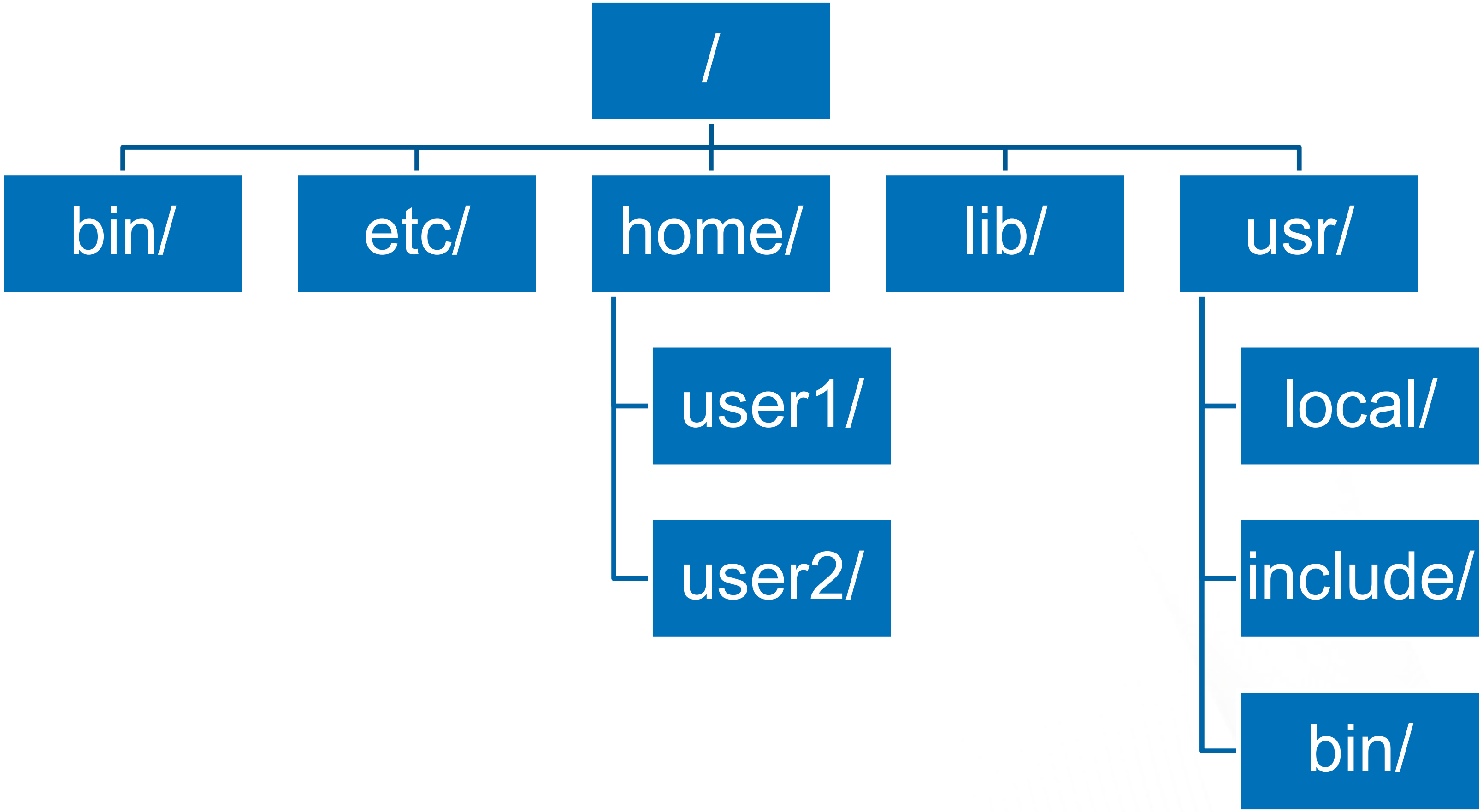
The Shell

- The shell is a computer program that acts as a layer between the user and the kernel
- Shells provide a rudimentary programming language with some control structures, variables, etc.
- There are many shells available and the choice is up to you
  - Not all may be available on all systems
  - Very likely that `bash` and `tcsh` are available

The Filesystem

- Think of the filesystem as a tree
- It starts at /, which is called the “root” directory
- The slash (/) is the directory separator; thus when we go into subdirectories it’s used to separate things (i.e. /home/user1/src)

Filesystem Hierarchy (Partial)



/

bin/

etc/

home/

user1/

user2/

lib/

usr/

local/

include/

bin/

The Filesystem

- There are some fairly standard directories:

Directory	Description
`/bin`	Programs for end-users. Many commands live here.
`/etc`	Configuration files
`/home`	User home directories (although not @ OLCF)
`/lib, /lib64`	Libraries
`/opt`	Often, “third party” software gets installed here
`/sbin`	Administrative programs/commands
`/usr`	Another location for user software/commands /usr/bin sometimes mirrors /bin; often has /usr/include and /usr/lib that contain files for software development

Files

- Files are the basic entity for storing data
- Might contain a program, data, configuration info, etc.
- Files have several attributes
  - Permissions: who can do what to/with a file
  - Owner: whose file is this
  - Group: to which group does this file belong

Special Directories

- Every directory contains two special directory entries: . and ..
- . is a reference to the current directory
- .. is a reference to the parent directory (so we can do things like `cd ..`)
- ~ can be a reference to home directories
  - ~/ is yours
  - ~user1/ is user1’s
- You’ll see how these are useful later

Wildcards

- When dealing with multiple files, it’s nice to type only one command vs. one command for each file
- Wildcards help with this: They are generic characters that “fill in” for other characters
  - `*` means match zero or more character
  - `?` Matches 1 character
  - Example follows (the `ls` command lists files in a directory, we’ll worry about specific later)

Wildcards (example)

```
$ ls
file1    file1a   file1b   file2    file2a   file2b   file3
file3a   file3b

$ ls file1?
file1a   file1b

$ ls file2*
file2    file2a   file2b

$ ls file?a
file1a   file2a   file3a
```

Quoting

- Quotes are often used in variable assignment
- Typically used to make the system recognize a string w/spaces as a single entity
- Different quotes do different things
  - Single quotes (apostrophe) make the string literal…characters like $ have no special meaning
  - Double quotes (quotation marks) apply special meaning to characters like $
  - Backquotes (`) run a command and assign the output to the variable
  - Backslash (\) removes special meaning from the next character

Quoting

Assume X is set to 1234

Given	Y is set to
Y=‘Test $X’	Test $X
Y=“Test $X”	Test 1234
Y=\`date\`	The current datetime string
Y=$(date)	The current datetime string This is the same as backquotes & is preferred by many
Y=“\$X is $X”	$X is 1234

Where to Get Help

- We'll start talking about commands now
- Some help is available via an online manual
  - The `man` command
  - Example: Want info about `ls`?
    ```
    man ls
    ```
- Plenty is available via the web
  - You'll see this one again: stackoverflow.com

Basic Commands

- Commands are usually abbreviations of words (or a series of words)
  - cp for “Copy”
  - rm for “Remove”
- Commands tend to be single-purpose but can be combined for more specialized tasks
  - More in part 2
- Almost all commands take various options to control what they do

Basic File/Directory Commands

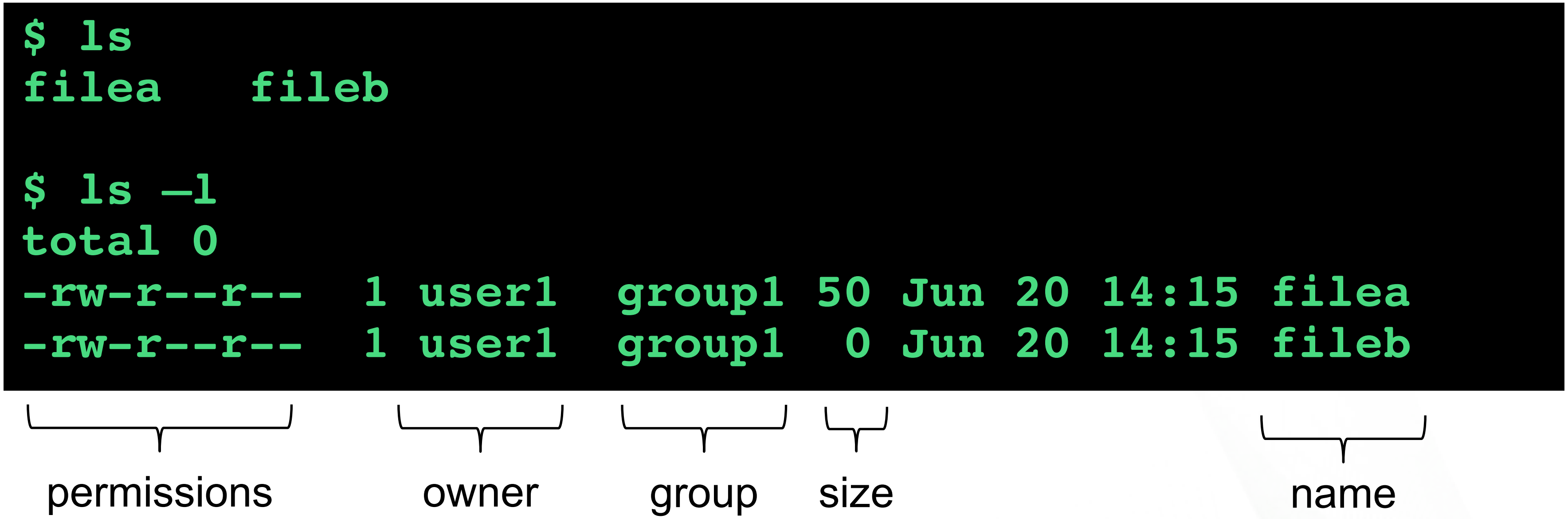
Command	Description
`ls`	**L**i**s**t files
`mkdir`	Create a directory (**M**a**K**e **DIR**ectory)
`rmdir`	Delete a directory (**R**e**M**ove **DIR**ectory)
`cp`	**C**o**p**y a file
`mv`	**M**o**v**e a file (also used to rename a file)
`rm`	**R**e**m**ove (delete) a file
`cd`	**C**hange (into a) **d**irectory
`pwd`	**P**rint **w**orking (i.e. current) **d**irectory
`cat`	Display the contents (con**cat**enate) a (hopefully text) file
`more`	Show a file a screenful at a time
`less`	`less` is `more`, but with a guaranteed ability to scroll backwards

Basic File/Directory Commands

Command	Description
`chown`	**Ch**ange the **Own**er of a file (only root can do this)
`chgrp`	**Ch**ange the **Gr**ou**p** of a file
`chmod`	**Ch**ange a file's **mod**e (permissions)
`echo`	Print a string to the terminal (Can be used to show setting of a variable)
`exit`	Quit the current shell (also used to close a window)
`groups`	List the groups to which the current user belongs
`whoami`	Display the current user's username (Don't laugh…this is useful)
`quota`	Show storage limits
`du`	Show **d**isk **u**sage
`df`	Show **d**isk **f**ree space

Basic Commands – ‘ls’

- Lists directory contents
- Helpful option: -l (shows many file attributes)



```
$ ls
filea    fileb

$ ls –l
total 0
-rw-r--r--  1 user1  group1 50 Jun 20 14:15 filea
-rw-r--r--  1 user1  group1  0 Jun 20 14:15 fileb
```

permissions owner group size name

Basic Commands – ‘ls’ (Other Useful Options)

Option	Meaning
`-1`	Show one file per line (helpful in scripting)
`-F`	Show file types (directories, links, etc)
`-a`	Show all files (including hidden files)
`-r`	Reverse the order of the listing
`-t`	Sort files by timestamp
`-d`	List the (attributes of) the directory itself rather than listing its contents
…And many, many (many) more	

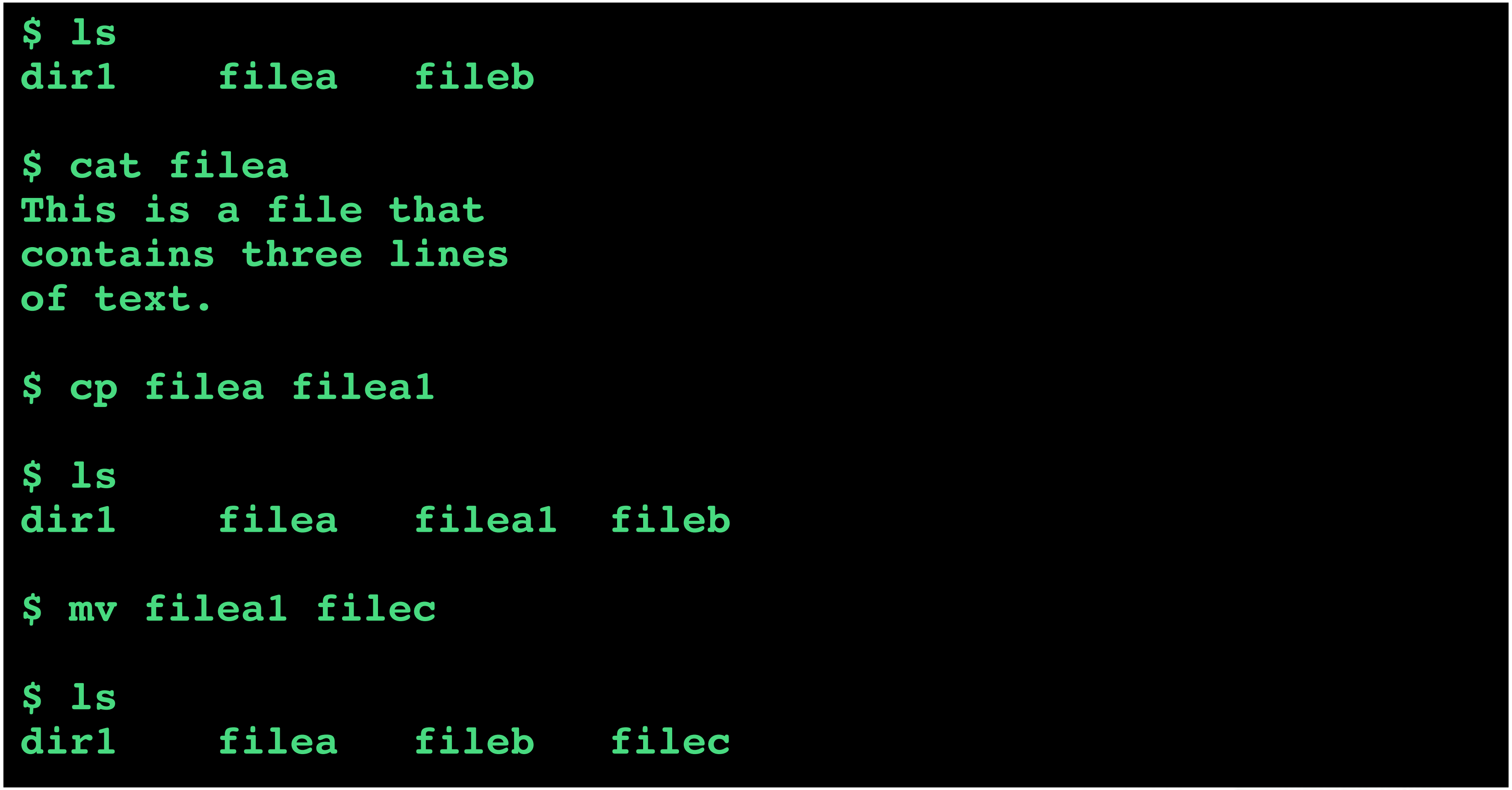
- You can combine options: `ls -altr` is the same as `ls -a -l -t -r` (but more concise & w/less typing)

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Basic Commands – Fun with directories

```
$ mkdir dir1

$ ls
dir1    filea   fileb

$ ls -ldF dir1
drwxr-xr-x  2 user1  group1  68 Jun 20 14:29 dir1/

$ cd dir1

$ pwd
/home/user1/dir1

$ cd ..

$ rmdir dir1

$ ls
filea   fileb
```

Basic Commands – Fun with files



```
$ ls
dir1    filea   fileb

$ cat filea
This is a file that
contains three lines
of text.

$ cp filea filea1

$ ls
dir1    filea   filea1  fileb

$ mv filea1 filec

$ ls
dir1    filea   fileb   filec
```

Basic Commands – Fun with files

```
$  cat filec
This is a file that
contains three lines
of text.

$ rm filec

$ ls
dir1    filea   fileb
```

Basic Commands

- Utilities such as `more`, `less`, and `cat` are intended for text files
- The system will not stop you from running them on a non-text file
  - If you do, you’ll get a screenful of unintelligible characters
  - You might get a recognizable prompt (you might not)
  - There’s no shame in closing that session’s window & re-connecting

File Permissions

- The system gives you the ability to specify who can access your files/directories (within limits)
- Every file has one owner and is associated with one group (we saw this in the discussion of `ls`)
- We can set permissions for the owner, the group, and everyone else
- There are three basic permissions: read, write, and execute
- Permissions have different meanings for files & directories

File Permissions

Permission	Meaning for files	Meaning for directories
read	Contents of the file can be displayed	Contents of directory can be listed
write	File can be modified or deleted	Files can be created in or deleted from directory
execute	File can be run like a program	Directory can be entered (i.e. cd into directory works)

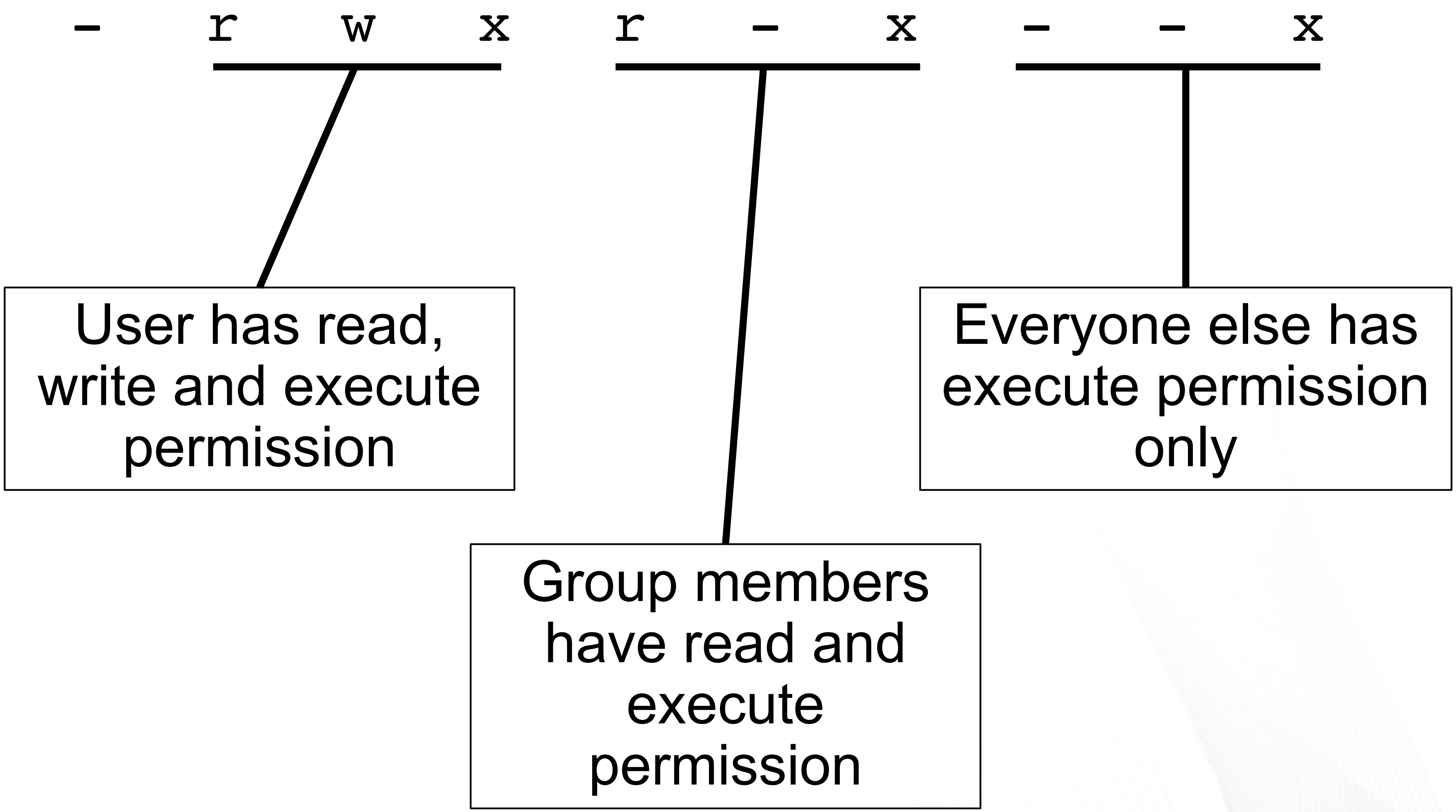
File Permissions

- Recall that for each file, `ls -l` showed a 10-character string similar to `-rwxr-xr-x`
- This shows the files permissions
- The leftmost character tells us what type of file it is: `-` for a regular file, `d` for a directory, `l` for a symlink (more about those later)
- The next 9 characters are three groups of three showing permissions for the file's user, group, and everyone else

File Permissions

- In the permissions string, the characters r, w, and x mean read, write, and execute permission is granted
- A - means the permission is not granted
- The permission groups always show read, write, execute in that order

File Permissions



`- r w x r - x - - x`

User has read, write and execute permission

Group members have read and execute permission

Everyone else has execute permission only

File Permissions

- File permissions are set/changed with the `chmod` command
- Two ways of using the command
  - Octal: Setting exact permissions
  - Symbolic: Using letters
- There are benefits/drawbacks to each method (although I prefer octal in most cases)

File Permissions

- Setting w/the octal method
  `chmod ### file`
  (where each # is a digit 0-7)
- First digit is user permission, second is group, third is other
- Digits are the sum of desired permissions; read is 4, write is 2, execute is 1
- Sum desired permissions
  - If ‘user’ should have read, write, and execute, the first digit should be 7

File Permissions

```
-   r   w   x   r   -   x   r   -   x

    4 + 2 + 1   4 +   + 1     +   + 1

        7           5           1
```

```
chmod 751 my_file
```

File Permissions

- Setting with the symbolic method
- Specify which settings you're changing (u)ser, (g)roup,(o)ther, or (a)ll
- Specify if you're adding (+) or deleting (-) permissions
- Specify the permissions to add/remove (r)ead, (w)rite, e(x)ecute

```
chmod [ugoa] [+-] [rwx] filename
```

File Permissions

- The symbolic way is so easy…why would anyone use octal?
- Consider a file with zero permissions that we want to change to match our example

  ```
  chmod u+rwx my_file
  chmod g+rx my_file
  chmod o+x my_file
  ```

  or

  ```
  chmod 751 my_file
  ```
- Additionally, with the symbolic method you need to know current permissions to know what to add/subtract; octal explicitly sets things

Verifying User & Group Information

- When considering permissions, you may need to double-check your group memberships or those of others
- You may also need to double-check your username (maybe you have multiple accounts w/different usernames)

```
$ whoami
user1

$ groups
user1 : group1 staff

$ groups user2
user2 : group2 users faculty
```

Storage

- The filesystem abstracts details about the actual storage media to an extent
  - You only deal with a directory name like /home, not details of the hardware (like C:\)
  - Sometimes different storage areas will have limits
  - All storage areas have finite size
- Wouldn't it be nice to know details about these things?

Checking Limits - quota

- The quota command shows limits for your account on each filesystem
- Two types of limits: total size and number of inodes (more later, but essentially # of files)
  - Limiting number of kB/MB/GB stored makes sense
  - Why limit inodes? Just as space is limited, # of inodes is limited
- Usage

```
quota [options]
```

Checking Usage - du

- The du command (disk usage) shows storage usage
- By default, it'll show info for every subdirectory (the `-s` option summarizes usage for the whole directory structure)
- Can take other options like `-k` (show in kB) or `-h` (show "human friendly" form)
- Usage

```
du [options] [directory]
```

Example of quota and du

```
$ quota –s
Disk quotas for user user1 (uid 19283):

  Filesystem  blocks  quota  limit  grace  files  quota  limit  grace
/nccs/home1        8 51200M  51200M            2  4295m  4295m
/nccs/home2   44794M 51200M  51200M         188k  4295m  4295m
```

```
$ du -sk .
45947600        .

$ du –sh .
44G     .
```

Checking Free Space - df

- The df command (Disk Free) shows how much space is available
- Common options are -k (show in kB), -h or –H (show “human-friendly” format)
- Can either show you all filesystems or a specific one
- Usage

  ```
  df [options] [path]
  ```

Checking Free Space - df

```
$ df .
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda5      669329936 264447700 370882264  42% /opt



$ df -H .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda5       686G  271G  380G  42% /opt



$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda5       639G  253G  354G  42% /opt
```

Advanced

Advanced Permissions

- There are other special permissions
  - Set User ID (setuid)
  - Set Group ID (setgid)
  - Sticky Bit
- These “replace” user, group, and other execute bit
- Meaning depends on whether it’s a file or directory

Advanced Permissions

Permission	Meaning for files	Meaning for directories
setuid	When executed, it will run with file's owner rather than as the invoking user	Files created *might* inherit directory's owner rather than be created as the creating user*
setgid	When executed, it will run with the file's group rather than the invoking user's	Files created will inherit directory's group, not user's
Sticky bit	Essentially meaningless on modern systems	Users cannot delete other users' files, even if they have write permission

* https://www.gnu.org/software/coreutils/manual/html_node/Directory-Setuid-and-Setgid.html

Advanced Permissions

- Set with the chmod command
- Add a fourth (leftmost) digit
  - Calculate value just like you do for others
  - Setuid=4, setgid=2, sticky bit=1
- So, `chmod 2770 my_dir` means anyone in the group can create a file in the directory, and it'll inherit the directory's group
- In `ls -l`, setuid is shown as an s in the user execute spot, setgid with an s in the group execute slot, and sticky bit with a t in the other execute spot

Advanced Permissions

```
$ mkdir dir2

$ chmod 770 dir2

$ ls -ld dir2
drwxrwx---  2 user1 group1 68 Jun 20 22:46 dir2

$ chmod 2770 dir2

$ ls -ld dir2
drwxrws---  2 user1 group1 68 Jun 20 22:46 dir2
```

More Commands

Command	Description/Use
`touch`	Creates an empty file or modify file timestamp(s)
`ln`	Create links to files
`date`	Display or set the date (*very* flexible command)
`umask`	Somewhat control default permissions (More accurately, block certain default permissions)
`grep`	Search for regular expressions in files
`tar`	Combine multiple files into one
`bzip2` `gzip`	Compress files
`ps`	Show processes that are currently running
`kill`	Send a signal to a process
`bg`	Bring backgrounded process to foreground

Working With Dates

- The date command seems fairly bland
- Modern versions actually very powerful
  - Various formats
  - Show various dates/offsets
  - Useful in scripts
  - Be careful w/DST changeover

```
$ date ‘+%Y%m%d’
20180627

$ date –d’yesterday’ ‘+%Y%m%d’
20180626
```

Referencing Files

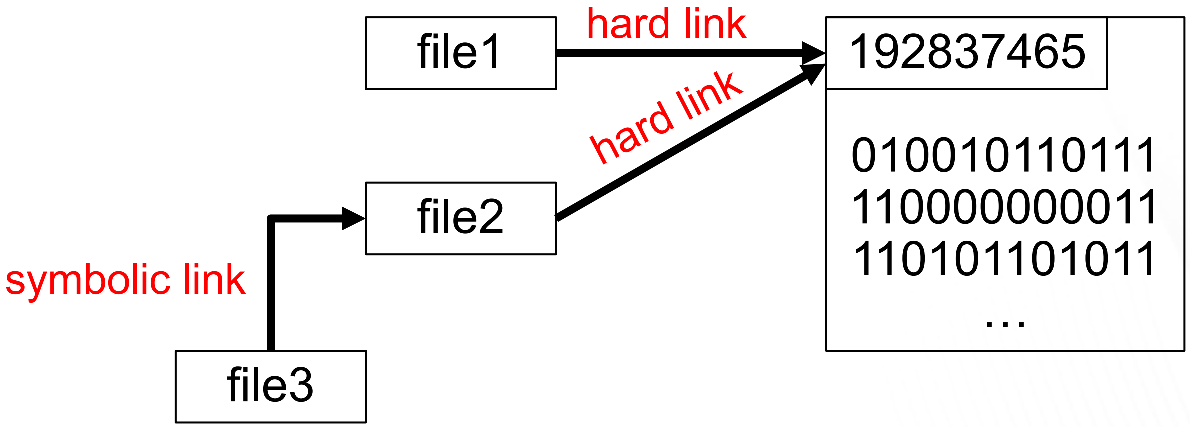
- File names are really just for ease-of-use by humans
- The system uses numbers called inodes to refer to files
  - It also uses numbers to refer to users, groups, other computers, etc.
  - That’s not really important but it sets up the next few slides ☺
- Thus, the filename is really just a reference to an inode

Referencing Files

- What if we want our current directory to contain a reference to a file in some far-off directory?
- We can create a link (similar to a shortcut in various other OSes)
- Two types of links: hard and symbolic (a.k.a. soft)
  - Hard link is a reference to an inode
  - Symbolic link (or symlink) is a reference to a file name
- Files (inodes) aren't deleted until all hard links referencing them are deleted
  - But deleting any given hard link might break a symlink
- …*What?*

Referencing Files

- Deleting `file1` -or- `file2` will leave data intact (although deleting `file2` will break the symlink even if `file1` remains)
- Once `file1` and `file2` are removed, the disk can reuse the space (but won't actually delete anything)



Referencing Files

- So why both?
- Hard links are essentially copies of files and don’t depend on some other filename existing
- Inodes are per-filesystem, so hard links cannot span filesystems (even a modest laptop can/will have multiple filesystems)
- Symbolic links can span filesystems (but rely on some intermediate filename existing)
- The `ls -li` command can tell us lots about links (`-i` shows inode info)

Referencing Files

- The ln (link) command creates both hard and symbolic links
- Usage
  `ln [-s] target link`
  my mnemonic: create a link to *target* named *link*

```
$ ln filea fileb


$ ln -s fileb filec
```

Referencing Files

```
$ ls –li
total 24

1597960 -rw-r--r-- 2 user1 group1 50 Jun 20 17:25 filea
1597960 -rw-r--r-- 2 user1 group1 50 Jun 20 17:25 fileb
1597969 lrwxr-xr-x 1 user1 group1  5 Jun 20 17:25 filec -> fileb
```

- Hard links
  - Note the inode (left) is the same for filea & fileb
  - The ‘2’ before user1 tells us there are 2 hard links to this inode
- Symbolic links
  - Indicated by ‘`l`’ at beginning of permissions string
  - `filec->fileb` tells us it’s named `filec` & links to `fileb`

Controlling Default Permissions

- Different files have different default permissions
- There's no list of defaults…it depends on the invoking program
- Sometimes we don't want certain permissions to happen
- Enter `umask`…

Controlling Default Permissions

- People often incorrectly think of `umask` as a default permission
- It’s not
- It’s a set of permissions that are blocked
- It takes numbers like the octal version of chmod, but these are the permissions you want blocked
- To block all other/world permissions on files

  ```
  umask 007
  ```
- To block group write and all other/world permissions on files

  ```
  umask 027
  ```

Searching Within Files - grep

- Sometimes you want to search for patterns/strings in a file
- Use the grep command for this
- The grep command searches for “regular expressions”…strings that contain characters with special meaning

Searching Within Files - grep

- Simple case: find lines with the string ‘user’ in file1

```
grep “user” file1
```

- More complex: show lines ending with ‘user’ in file1

```
grep “user$” file1
```

- …or perhaps lines beginning with ‘user’

```
grep “^user” file1
```

- As with other commands, `grep` takes many options

Searching Within Files - grep

- Normally, `grep` will treat anything beginning with a hyphen as an option…even if it's in quotes
- The workaround is the `--` option, which tells grep that you're done giving it options (and therefore any other hyphen is meant as an actual hyphen)

```
grep -- "-2" file1
```

Searching for files

- The `find` command lets you search for files on a huge variety of criteria
- It can also run commands on those files; this makes it one of the most powerful commands available

```
$ find . -name “*data*” -print

$ find . -newer some_file
$ find /home/user1 ! -user user1

$ find . -group users -exec chgrp staff {} \;
```

Other Commands - tar

- The `tar` (Tape ARchive) command is used to combine multiple files into a single file or extract files from such an archive
- Create `file.tar` containing (recursively) the `mydir` directory

```
tar cvf file.tar mydir
```

- Extract that file

```
tar xvf file.tar
```

Compressing Files

- Several utilities
  - `gzip` (GNU zip)
  - `bzip2`
- Those utilities both compress and decompress (bunzip2 is also available)
- They use different compression algorithms and are not interchangeable
- Usage

```
bzip2 file.tar
bzip2 -d file.tar.bz2
gzip file.tar
gzip -d file.tar.gz
```

Process Management

- All processes are identified by a process id or `pid`
- You can view process information with the `ps` command
- Foreground vs. background processes
- Type `Ctrl-Z` to send process to background
- Type `fg` to bring backgrounded process to the foreground

Process Management

- The `kill` command is used to send a signal to a process
  ```
  kill [options] pid
  ```
  - There are many signals that can be sent (only one of which is `SIGKILL`)
  - By default, `kill` sends `SIGTERM`, not `SIGKILL`
- Common signals shown on next slide
- Some signals can be “trapped”, others can’t
  - This permits custom reaction to certain signals
  - (Ignoring the signal is one such action)

Common Signals

Name	Number	Meaning
`SIGHUP`	1	Hangup (terminal went away)
`SIGINT`	2	Interrupt from keyboard (think Ctrl-C)
`SIGILL`	4	Illegal instruction
`SIGABRT`	6	Abort signal
`SIGFPE`	8	Floating-point exception
`SIGKILL`	9	Kill process (go away, period)
`SIGSEGV`	11	Segmentation fault (illegal memory access)
`SIGTERM`	15	Terminate process (please go away)
`SIGUSR2`	Varies	User-defined signal 2

http://man7.org/linux/man-pages/man7/signal.7.html

Variables

- Shells also support variables
- Some are “standard” things expected/used by the system
- We can also have user-defined variables
- Two major types
  - Environment
  - Shell

Variables

- An environment variable will be passed to a subshell (such as a shell script)
- Shell variables are not passed to subshells.
- Method of setting depends on the shell
- To use, use `$` followed by name, such as `$PATH`
- Placing the name in braces is a good practice (i.e. `${PATH}`) to avoid ambiguity
- You can display a variable's value with `echo`:

  ```
  echo $PATH
  ```

Setting Variables

- sh/bash/similar shells
  - Environment
    ```
    export VARNAME=value
    ```
  - Shell
    ```
    VARNAME=value
    ```
- csh/tcsh
  - Environment
    ```
    setenv VARNAME value
    ```
  - Shell
    ```
    set NAME=value
    ```

Setting Variables

- You can define a variable in terms of itself
- To append to a string, you might use
  `export MYVAR=“${MYVAR} and more”`
- This is often needed to prepend/append to variables like `$PATH`

Common Environment Variables

Variable	Meaning/Use
`$PATH`	A list of directories the system searches for executables
`$USER`	Current user’s username
`$LD_LIBRARY_PATH`	A list of directories to search for dynamic libraries

How Executables Are Located

- How does the system find the program you want to run?
- Two options
  - You provide the exact location
    - Absolute path: `/usr/bin/perl`, `/bin/ls`
    - Relative path: `./a.out`, `../bin/a.out`
  - You rely on the `$PATH` variable
    - If you don’t provide the exact location, the system searches for the program in each directory in `$PATH` and uses the first it finds

How Executables Are Located

- You should use `./a.out` to reference `a.out` in the current directory
  - “.” in `$PATH` is discouraged for security reasons
  - The system has no particular affinity for the current directory
- If you don’t provide the exact location, `which` can show you what the system will choose

```
$ which chmod
/usr/bin/chmod
```

Redirection

- Sometimes we want to take input from a file or write output to a given file
- Redirection lets us do this

Syntax	Meaning
`ls > file1`	Place ls output in a file named file1; overwrite if it already exists
`ls >> file1`	Append ls output to the end of file1
`./run.x <file1`	Run the program run.x, feeding file1 one line at a time the program

`./run.x <<EOF`
`Line1`
`Line2`
`EOF`
	“Here” document. Same as the line above, but instead of specifying an existing file, we provide the file “here”. The EOF string is a starting/ending token and not part of the file

Redirection

- Some redirection is shell-specific

bash & similar shells	csh/tcsh	Meaning
`./exe >>out 2>&1`	`./exe >>& out`	Run exe, append stdout and stderr to the file named out
`./exe >>out 2>>err`	`(./exe >> out) >>& err`	Run exe, append stdout to the file named out and stderr to the file named err

Reference:
https://linux.die.net/man/1/csh
https://linux.die.net/man/1/bash

Putting it all Together

- In general, system commands are simple and single-purpose
- User can “build” more complex commands
- This is similar to CISC vs. RISC computer architecture
- Combination can be simple or complex
  - A series of commands with output of one feeding input of another (pipes)
  - Several commands executed independently but in one shell script

Using Pipes

- Send ‘ls’ output through ‘more’ to show a page at a time

  ```
  ls -l |more
  ```

- Uncompress a .bz2 file then untar the resulting file

  ```
  bzip2 -d file.tar.bz2 |tar xf -
  ```

- Count the number of unique lines in a file (display the file, sort it, remove duplicates, and count lines)

  ```
  cat bigfile|sort|uniq|wc -l
  ```

Shell Scripts

- What if you always run a series of commands
  - It'd be nice to save them in some fashion
  - You don't want to keep re-typing these commands, so a pipe isn't appropriate
  - Enter the shell script
- A shell script is an executable file that contains a list of commands for the shell to run
- The first line begins with #!, a space, and the shell to use, for example:

```
#! /bin/bash
```

Shell Scripts

- Lines beginning with # are comments
  - You should document why you're doing what you're doing (for yourself and others)
  - The system will just ignore these lines
  - The #! on line 1 is a comment but the system processes it in a special way (if it's not on line1, it's just a normal comment)

Shell Scripts

```
#! /bin/bash

# Get today’s date in YYYYMMDD form
today =$( date “+%Y%m%d” )

# Go to /tmp, run getdata, postprocess the output, and
# put the results in ~/data in a file w/today’s date
cd /tmp
~/getdata.x > mydata.${today}
~/postprocess.x /tmp/mydata.${today} > ~/data/${today}

# Clean up /tmp
rm /tmp/mydata.${today}
```

Shell Scripts

- Suppose the file on the previous page is named "`daily_run.sh`" and has execute permission
- To run that series of commands, we simple run:
  `./daily_run.sh`
- Shell scripts are also used by utilities such as `cron` that let us schedule tasks on the system
  - Much easier to schedule a single item than (potentially) hundreds of individual commands

Shell Scripts

- Remember I mentioned shells provided some rudimentary programming structures?
- Your shell script can contain if and for statements among others
- These use either the `test` command or bracket syntax for the logic test
- Some control structures include `if`, `for`, and `case`
- Some (`bash`) examples follow

Shell Scripts

```
if [[ $i -lt 4 ]]; then
  echo “i is less than 4”
elif [[ $i –gt 4 ]]; then
  echo “i is greater than 4”
else
  echo “i equals 4”
fi


for i in 1 2 3
do
 echo $i
done
```

https://linux.die.net/man/1/bash

Shell Scripts

```
case $i in
1) echo ”i is one”;;
2) echo ”i is two”;;
3) echo ”i is three”;;
4) echo ”i is four”;;
esac
```

https://linux.die.net/man/1/bash

Exit Values

- (Almost) every command will return some status value
- Typically 0 if everything worked, nonzero otherwise
- Can be used in if statements to verify the previous command worked
- Variable `$?` contains exit value of last command
- Shell scripts typically return exit value of last command executed

For Further Investigation

- We’ve covered a small subset of handy commands
- Many, many more to research on your own
- Some advanced (but helpful) ones:

Command	What it does
`od`	Display files in various formats (hexadecimal, octal, ascii)
`dd`	Do bit-by-bit copy/conversion of files (sometimes called data/disk destroyer, so be careful…you have been warned)
`top`	Show what processes are consuming CPU/memory
`cron`	Set tasks to run at scheduled intervals
`sudo`	Run commands as someone else (typically root)
`alias`	Create a shortcut for certain commands

https://linux.die.net/man/1/dd

For Further Investigation

- There are also some special-purpose files

File	What it provides
`/dev/null`	The 'bit bucket'. Anything written/redirected here goes away
`/dev/random` `/dev/urandom`	Provides random bytes of data
`/dev/zero`	Provides an endless stream of zeroes

- These are remarkably useful for various tasks

For Further Investigation

- Create a 4MB file containing random data

```
$ dd if=/dev/urandom of=4MBfile bs=4k count=1024
1024+0 records in
1024+0 records out
4194304 bytes (4.2 MB) copied, 0.041256 s, 102 MB/s

$ ls -l 4MBfile
-rw-r--r--. 1 user1 group1 4194304 Jun 27 13:45 4MBfile
```

- Run a command, but get rid of error messages

```
$ ./run_command 2>/dev/null
```

For Further Investigation

- Other things we didn’t cover
  - Logging in
  - File transfer utilities
  - Batch queues
  - Programming (including parallel & GPU programming)
- Other sessions will cover/have covered that
- Plenty of resources for that on the web as well
  - stackoverflow.com is your friend

For Further Investigation

- There are many more options for variable manipulation
- Check the manual page for your preferred shell or do a web search for shell syntax

Where to Get Help

- Experienced users
- stackoverflow.com
- gnu.org (lots of info on GNU utilities)
- Websites for various linux distros (often will have a forum available)
- Too many other resources to list

Summary

- This has been a small overview into Unix-like operating systems
- These OSes run all kinds of computers, from small embedded systems to supercomputers
- These OSes provide basic commands & give you the ability to build more complex commands
- Things can get complicated, but there's plenty of help available