Tricky Examples

- The wc command can take multiple files: wc names.txt student.txt
  - Can we use the following to wc on every txt file in the directory?
    - `ls *.txt | wc`

- Amongst the top 250 movies in movies.txt, display the third to last movie that contains "The" in the title when movie titles are sorted.

- Find the disk space usage of the man program
  - Hints: use `which` and `du`...
  - Does `which man | du` work?

Command Substitution

command1 `$(`***command2***`)`

- run ***command2*** and pass its console output to ***command1*** as a parameter;
- best used when ***command2***'s output is short (one line)

- Finish the example!
  - `du $(which man)`

xargs

command	description
`xargs`	run each line of input as an argument to a specified command

- `xargs` allows you to repeatedly run a command over a set of lines
  - often used in conjunction with `find` to process each of a set of files

- Example: Remove all my .class files.

```
find ~ -name “*.class” | xargs rm
```

- Find the disk usage of `man` using `xargs`
  - `which man | xargs du`

Text editors

command	description
`pico` or `nano`	simple editors
`emacs`	More advanced text editor
`vi` or `vim`	More advanced text editor

- you cannot run graphical programs when connected remotely
  - so if you want to edit documents, you need to use a text-only editor
- **most advanced Unix/Linux users learn `vi`**
  - I would recommend you try to pick up the basics.

Basic Vim Commands

- :w Write the current file
- :wq Write the current file and exit.
- :q! Quit without writing
- To change into insert mode: i or a
  - Use escape to exit
- search forward /, repeat the search backwards: N
- Basic movement:
  - h l k j character left, right; line up, down (also arrow keys)
  - b w word/token left, right
  - ge e end of word/token left, right
  - 0 $ jump to first/last character on the line
- x delete
- u undo

https://wiki.gentoo.org/wiki/Vim/Guide and http://tnerual.eriogerg.free.fr/vimqrc.pdf

Aliases

command	description
`alias`	assigns a pseudonym to a command

`alias` ***name=command***

- must wrap the command in quotes if it contains spaces
- **Do not put spaces on either side of the =**

- Example: When I type `q` , I want it to log me out of my shell.
- Example: When I type `ll` , I want it to list all files in long format.

```
alias q=exit
alias ll="ls -la"
```

- *Exercise* : Make it so that typing `q` quits out of a shell.
- *Exercise* : Make it so that typing `woman` runs `man`.

`.bash_profile and .bashrc`

- Every time you log in to bash the commands in `~/.bash_profile` are run
  - you can put any common startup commands you want into this file
  - useful for setting up aliases and other settings for *remote login*
- Every time you launch a non-login bash terminal (e.g. bash), the commands in `~/.bashrc` are run
  - useful for setting up persistent commands for *local shell usage*
  - often, `.bash_profile` is configured to also run `.bashrc`, but not always
- Similar functions, but they have different scopes and are executed at different times.

Note: a dot (.) in front of a filename indicates a normally hidden file, use ls –a to see

Exercise:Edit your .bashrc

- *Exercise* : Make it so that our alias from earlier becomes persistent, so that it will work every time we run a shell.
- *Exercise* : Make it so that whenever you try to delete or overwrite a file during a move/copy, you will be prompted for confirmation first.

Making Changes Visible

- After editing your `.bashrc` or `.bash_profile`, how do you make the aliases etc. in the file take effect?

  - `.bash_profile`
    - log on again or
    - `source .bash_profile`
  - `.bashrc`
    - start another bash shell (type: `bash`), or
    - `source .bashrc`

Users

Unix/Linux is a multi-user operating system.

- Every program/process is run by a user.
- Every file is owned by a user.
- Every user has a unique integer ID number (UID).

- Different users have different access permissions, allowing user to:
  - read or write a given file
  - browse the contents of a directory
  - execute a particular program
  - install new software on the system
  - change global system settings
  - ...

Groups

command	description
`groups`	list the groups to which a user belongs
`chgrp`	change the group associated with a file

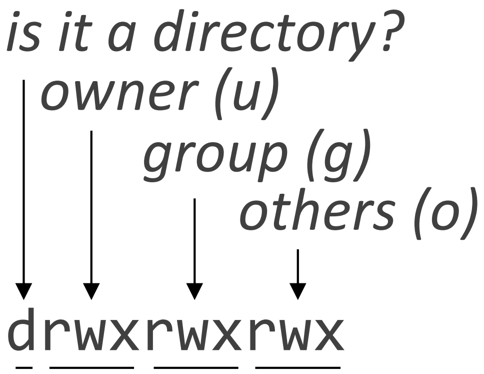
- **group**: A collection of users, used as a target of permissions.
  - a group can be given access to a file or resource
  - a user can belong to many groups
  - see who’s in a group using grep <groupname> /etc/group

- Every file has an associated group.
  - the owner of a file can grant permissions to the group
- Every group has a unique integer ID number (GID).
- *Exercise*: create a file, see its default group, and change it

File permissions

command	description
`chmod`	change permissions for a file
`umask`	set default permissions for new files

- *types* : read (`r`), write (`w`), execute (`x`)
- *people* : owner (`u`), group (`g`), others (`o`)
  - on Windows, .exe files are executable programs; on Linux, any file with `x` permission can be executed
  - permissions are shown when you type `ls -l`



People & Permissions

- **People**: each user fits into only **one** of three permission sets:
  - owner (u) – if you create the file you are the owner, the owner can also be changed (using chown)
  - group (g) – by default a group (e.g. ugrad_cs, fac_cs) is associated with each file
  - others (o) – everyone other than the owner and people who are in the particular group associated with the file

You are in the most restrictive set that applies to you – e.g. if you are the owner, those permissions apply to you.

- **Permissions**: For regular files, permissions work as follows:
  - read (r) – allows file to be open and read
  - write (w) – allows contents of file to be modified or truncated
  - execute (x) – allows the file to be executed (use for executables or scripts)

* Directories also have permissions. Permission to delete or rename a file is controlled by the permission of its parent directory.

File permissions Examples

Permissions are shown when you type `ls -l`:

```
-rw-r--r-- 1 rea fac_cs      55 Oct 25 12:02 temp1.txt
-rw--w---- 1 rea orca       235 Oct 25 11:06 temp2.txt
```

`temp1.txt:`

- **owner** of the file (rea) has read & write permission
- **group** (fac_cs) members have read permission
- **others** have read permission

`temp2.txt:`

- **owner** of the file (rea) has read & write permission
- **group** (orca) members have write permission (but no read permission – can add things to the file but cannot cat it)
- **others** have no permissions (cannot read or write)

Changing permissions

- letter codes: `chmod` ***who***(+-)***what*** `filename`

`chmod u+rw myfile.txt`	(allow owner to read/write)
`chmod +x banner`	(allow everyone to execute)
`chmod ug+rw,o-rwx grades.xls`	(owner/group can read and write; others nothing)

Note, no space after the comma!

- octal (base-8) codes: `chmod` ***NNN*** `filename`
  - three numbers between 0-7, for owner (u), group (g), and others (o)
  - each gets +4 to allow read, +2 for write, and +1 for execute

`chmod 600 myfile.txt`	(owner can read/write (rw))
`chmod 664 grades.dat`	(owner rw; group rw; other r)
`chmod 751 banner`	(owner rwx; group rx; other x)

chmod and umask

`chmod u+rw myfile.txt` (allow owner to read/write)

Note: leaves “group” and “other” permissions as they were.

`chmod 664 grades.dat` (owner rw; group rw; other r)

Note: sets permissions for “owner”, “group” and “other” all at once.

`umask` – returns the “mask” in use, determines the default permissions set on files and directories I create. Can also be used to set that mask.

```
% umask
0022
% touch silly.txt
% ls -l silly.txt
-rw-r--r-- 1 rea fac_cs 0 Oct 25 12:04 silly.txt
```

0022 means that files I create will have group and other “write bits” turned off:

1) Take the bitwise complement of 022_8 -> 755_8

2) AND with 666_8 for files (777_8 for directories) : 755_8 = 111 101 101

$$\begin{array}{r} 755_8 = 111\ 101\ 101 \\ 666_8 = \underline{110\ 110\ 110} \\ 110\ 100\ 100 = 644_8 \end{array}$$

(owner rw, group r, other r)

Directory Permissions

- Read, write, execute a directory?
  - **Read** - permitted to read the contents of directory (view files and sub-directories in that directory, run `ls` on the directory)
  - **Write** - permitted to write in to the directory (add, delete, or rename & create files and sub-directories in that directory)
  - **Execute** - permitted to enter into that directory (`cd` into that directory)
- It is possible to have any combination of these permissions:

Try these:

- Have **read** permission for a directory, but NOT **execute** permission
  - ????
- Have **execute** permission for a directory, but NOT **read** permission
  - ???

Directory Permissions

- Read, write, execute a directory?
  - **Read** - permitted to read the contents of directory (view files and sub-directories in that directory, run `ls` on the directory)
  - **Write** - permitted to write in to the directory (add, delete, or rename & create files and sub-directories in that directory)
  - **Execute** - permitted to enter into that directory (`cd` into that directory)
- It is possible to have any combination of these permissions:
  - Have **read** permission for a directory, but NOT **execute** permission
    - Can do an `ls` from outside of the directory but cannot `cd` into it, cannot access files in the directory
  - Have **execute** permission for a directory, but NOT **read** permission
    - Can `cd` into the directory, can access files in that directory if you already know their name, but cannot do an `ls` of the directory

Super-user (root)

command	description
`sudo`	run a single command with root privileges (prompts for password)
`su`	start a shell with root privileges (so multiple commands can be run)

- **super-user**: An account used for system administration.
  - has full privileges on the system
  - usually represented as a user named `root`

- Most users have more limited permissions than `root`
  - protects system from viruses, rogue users, etc.

tar files

	description
`tar`	create or extract .tar archives (combines multiple files into one .tar file)

- Originally used to create “tape archive” files
- Combines multiple files into a single .tar file
- You probably always want to use –f option and IT SHOULD COME LAST

- To **create** a single file from multiple files:

```
$ tar -cf filename.tar stuff_to_archive
```

  - -c **creates** an archive
  - -f read to/from a file
  - stuff_to_archive - can be a list of filenames or a directory

- To **extract** files from an archive:

```
$ tar -xf filename.tar
```

  - -x **extracts** files from an archive

Compressed files

command	description
`zip, unzip`	create or extract .zip compressed archives
`gzip, gunzip`	GNU free compression programs (single-file)
`bzip2, bunzip2`	slower, optimized compression program (single-file)

- To **compress** a `file`:

```
$ gzip filename                produces:   filename.gz
```

- To **uncompress** a `file`:

```
$ gunzip filename.gz           produces:   filename
```

Similar for zip, bzip2. See man pages for more details.

.tar.gz archives

- Many Linux programs are distributed as .tar.gz archives (sometimes called .tgz)
- You could unpack this in two steps:
  1. `gzip foo.tar.gz` produces: foo.tar
  2. `tar -xf foo.tar` extracts individual files
- You can also use the tar command to create/extract compressed archive files all in one step:

```
$ tar -xzf filename.tar.gz
```

  - -x **extracts** files from an archive
  - -z filter the archive through gzip (compress/uncompress it)
  - -f read to/from a file

Shell scripts

- **script**: A short program meant to perform a targeted task.
  - a series of commands combined into one executable file

- **shell script**: A script that is executed by a command-line shell.
  - `bash` (like most shells) has syntax for writing script programs
  - if your script becomes > ~100-150 lines, switch to a real language

- To write a `bash` script (in brief):
  - type one or more commands into a file; save it
  - type a special header in the file to identify it as a script (next slide)
  - enable execute permission on the file
  - run it!

Basic script syntax

#!*interpreter*

- written as the first line of an executable script; causes a file to be treated as a script to be run by the given interpreter
  - (we will use `/bin/bash` as our interpreter)

- Example: A script that removes some files and then lists all files:

```
#!/bin/bash
rm output*.txt
ls -l
```

Running a shell script

- by making it executable (most common; recommended):

```
chmod u+x myscript.sh
./myscript.sh
```

  - fork a process and run commands in `myscript.sh` and exit

- by launching a new shell :

```
bash myscript.sh
```

  - advantage: can run without execute permission (still need read permission)

echo

command	description
`echo`	produces its parameter(s) as output (the `println` of shell scripting) **-n flag to remove newline** (`print` vs `println`)

- Example: A script that prints your current directory.

```
#!/bin/bash
echo "This is my amazing script!"
echo "Your current dir is: $(pwd)"
```

- *Exercise* : Write a script that when run does the following:
  - clears the screen
  - displays the current date/time
  - Shows who is currently logged on & info about processor

Script example

```
#!/bin/bash
clear      # please do not use clear in your hw scripts!
echo "Today's date is $(date)"
echo

echo "These users are currently connected:"
w -h | sort
echo

echo "This is $(uname –s) on a $(uname –m) processor."
echo

echo "This is the uptime information:"
uptime
echo
echo "That's all folks!"
```

Comments

# comment text

- bash has only single-line comments; there is no /* ... */ equivalent

- Example:

```
#!/bin/bash
# Leonard's first script ever
# by Leonard Linux
echo "This is my amazing script!"
echo "The time is: $(date)"

# This is the part where I print my current directory
echo “Current dir is: $(pwd)"
```

Shell variables

- ***name=value*** *(declaration)*
  - must be written ***EXACTLY*** as shown; no spaces allowed
  - often given all-uppercase names by convention
  - once set, the variable is in scope until unset (within the current shell)

```
AGE=64
NAME="Michael Young"
```

- ***$name*** *(usage)*

```
echo "$NAME is $AGE years old"
```

Produces:

```
Michael Young is 64 years old
```

Common errors

- if you misspell a variable's name, a new variable is created

```
NAME=Ruth
...
Name=Rob            # oops; meant to change NAME
```

- if you use an undeclared variable, an empty value is used

```
echo "Welcome, $name"  # Welcome,
```

- when storing a multi-word string, must use quotes

```
NAME=Ruth Anderson       # Won’t work
NAME=“Ruth Anderson"     # $NAME is Ruth Anderson
```

More Errors…

- Using $ during assignment or reassignment
  - `$mystring=“Hi there”      # error`

  - `mystring2=“Hello”`
  - …
  - `$mystring2=“Goodbye”      # error`

- Forgetting echo to display a variable
  - `$name`
  - `echo $name`

Capture command output

variable`=$(`***command***`)`

- captures the output of ***command*** into the given variable

- Simple Example:

```
FILE=$(ls *.txt)
echo $FILE
```

- More Complex Example:

```
FILE=$(ls -1 *.txt | sort | tail -n 1)
echo "Your last text file is: $FILE"
```

Double vs. Single quotes

Double quotes - Variable names are expanded & $() work

```
NAME="Bugs Bunny"
echo "Hi $NAME! Today is $(date)"
```

Produces:

```
 Hi Bugs Bunny! Today is Tues Apr 25 13:37:45 PDT 2017
```

Single quotes – don’t expand variables or execute commands in $()

```
echo 'Hi $NAME! Today is $(date)'
```

Produces:

```
 Hi $NAME! Today is $(date)
```

Tricky Example:

- `STAR=*`
  - `echo "You are a $STAR"`
  - `echo 'You are a $STAR'`
  - `echo You are a $STAR`

Lesson: When referencing a variable, it is good practice to put it in double quotes.

Types and integers

- most variables are stored as strings
  - operations on variables are done as string operations, not numeric

- to instead perform integer operations:

```
x=42
y=15
let z="$x + $y"          # 57
```

- integer operators: `+ - * / %`
  - bc command can do more complex expressions

- if a non-numeric variable is used in numeric context, you'll get 0

Bash vs. Java

Java	Bash
String s = "hello";	s=hello
System.out.println("s");	echo s
System.out.println(s);	echo $s
s = s + "s"; // "hellos"	s=${s}s

String s2 = "25";
String s3 = "42";
String s4 = s2 + s3; // "2542"
int n = Integer.parseInt(s2)
+ Integer.parseInt(s3); // 67
	s2=25
	s3=42
	s4=$s2$s3
	let n="$s2 + $s3"

set, unset, and export

shell command	description
`set`	sets the value of a variable (not usually needed; can just use `x=3` syntax)
`unset`	deletes a variable and its value
`export`	sets a variable and makes it visible to any programs launched by this shell
`readonly`	sets a variable to be read-only (so that programs launched by this shell cannot change its value)

- typing `set` or `export` with no parameters lists all variables

Console I/O

shell command	description
`read`	reads value from console and stores it into a variable
`echo`	prints output to console
`printf`	prints complex formatted output to console

- variables read from console are stored as strings

- Example:

```
#!/bin/bash
read -p "What is your name? " name
read -p "How old are you? " age
printf "%10s is %4s years old" $name $age
```

Command-line arguments

variable	description
`$0`	name of this script
`$1, $2, $3, ...`	command-line arguments
`$#`	number of arguments
`$@`	array of all arguments

- Example.sh:

```
#!/bin/bash
echo “Name of script is $0”
echo “Command line argument 1 is $1”
echo “there are $# command line arguments: $@”
```

  - `Example.sh argument1 argument2 argument3`

Arrays

```
name=(element1 element2 ... elementN)

name[index]=value          # set an element
$name                      # get first element
${name[index]}             # get an element
${name[*]}                 # elements sep.by spaces
${#name[*]}                # array's length
```

- arrays don't have a fixed length; they can grow as necessary
- if you go out of bounds, shell will silently give you an empty string

Functions

```
function name() {            # declaration
    commands                 # ()’s are optional
}

name                         # call
```

- functions are called simply by writing their name (no parens)
- parameters can be passed and accessed as $1, $2, etc.

for loops

```
for name in value1 value2 ... valueN; do
    commands
done
```

- Note the semi-colon after the values!
- the pattern after `in` can be:
  - a hard-coded set of values you write in the script
  - a set of file names produced as output from some command
  - command line arguments: `$@`

- *Exercise*: create a script that loops over every .txt file in the directory, renaming the file to .txt2

```
for file in *.txt; do
  mv $file ${file}2
done
```

for loop examples

```
for val in red blue green; do
    echo "val is: $val"
done

for val in $@; do
    echo "val is: $val"
done

for val in $(seq 4); do
    echo "val is: $val"
done
```

command	description
seq	outputs a sequence of numbers

if/else

```
if [ condition ]; then                # basic if
    commands
fi

if [ condition ]; then                # if / else if / else
    commands1
elif [ condition ]; then
    commands2
else
    commands3
fi
```

- The `[ ] syntax` is actually shorthand for a shell command called “***test***” (Try: “man test”)
- there ***MUST*** be spaces as shown:

  `if` space `[` space ***condition*** space `]`
- include the semi-colon after] (or put “then” on the next line)