bash, part 3

Chris GauthierDickey

UNIVERSITY OF
DENVER

More redirection

- As you know, by default we have 3 standard streams:
  - input, output, error
- How do we redirect more than one stream?
  - This requires an introduction to file descriptors

File Descriptors

- Recall that Unix uses files to represent many types of things, from devices to network streams
- Each process has its own set of streams which are numbered (file descriptor)
  - standard input: file descriptor 0
  - standard output: file descriptor 1
  - standard error: file descriptor 2

Redirecting Streams

- We can redirect any file descriptor using:
  - n> file, where n is a number from 0 to the maximum number of file descriptors
  - n< file, redirects the file contents to descriptor n
  - By default, > file and < file are the same as 1> file and 0< file
  - To redirect standard output and standard error:
    - wget http://www.google.com > outfile 2> errfile

Appending

- We can append instead of overwriting:
  - >> redirects standard out, but appends
  - n>>file, redirects the fd to file, but appends

- Why would we want to do this?

printf instead of echo

- Echo is useful, but printf gives us more control over writing
- printf works by reading a string and passing arguments to it for substitution

```
# the following prints
# hello world\n
printf “hello %s\n” world
```

Formatting strings with printf

- %c: ASCII character
- %d, %i: decimal digit
- %e: floating point ([-]d.precision)[+-]dd)
- %E: floating point ([-]d.precisionE[+-]dd)
- %f: floating point ([-]ddd.precision)
- %s: string
- %o: octal value (unsigned)
- %u: unsigned decimal
- %x: unsigned hex
- %%: a literal %

Reading user input

- We can get user input by using the ‘read’ command
- read a b c will take a line of input and assign the first word to a, the next to b, and finally to c
- If you input more words, the final variable will get the rest of the line

Example reading

- Try this next example using 2, 3, and 4 arguments
- If you don’t give it enough arguments, all the vars aren’t filled
- If you give it too many, the last one takes the rest of the line

```
# read in user input to
# a, b, and c

read a b c
echo “a: $a, b: $b, c: $c”
```

Reading Options

- If you want to read into an array, use the -a option
    - read -a args; echo ${args[0]} ${args[1]}
- If you want to separate lines by something other than newline, use -d
    - read -d , args; echo $args
        - Entering hello,world will echo hello

More reading options

- -s will prevent what the user types from being echoed (think password)
- -n tells read how many characters to read in
- -e tells read to use the readline facilities, which gives advanced editing features on the line

Redirecting to a loop

- Reading is great, but we can redirect from a file to act as input
- We can redirect to functions, loops, if-statements
  - Read will then take its input from the redirected item

Example: redirecting to a loop

- Here we redefine IFS to be : so we can read from /etc/passwd
- Notice how we redirect the file to standard input

```
# redirecting from a file
# to a loop
IFS=:
while read v1 v2; do
  echo “v1: $v1, v2: $v2”
done < /etc/passwd
```

Command blocks

- We can enclose any set of commands by { }, which turns that set of commands into a block.
- Once it's a block, we can redirect input or output:
  - { read v; echo $v } < /etc/passwd

Fun Places for Redirection

- /dev/null: This is the proverbial bit-bucket--anything sent to here just goes away
- /dev/random: This is a string of random data that you can read from

Process Handling

- Recall:
  - CTRL-Z suspends a running job
  - fg moves the last background job to the foreground
  - bg moves the last suspended job into the background
  - jobs lists all the jobs

Jobs

- Each job has a job ID, the jobs commands lists all your processes with their job ID
- %n will refer to job ID n
- %foo will refer to the job with the command name that begins with foo
- %?foo will refer to the job with the command name that contains foo
- %- is the most recent bg job, %+ is the 2nd most recent

Signals

- CTRL-Z is actually a signal: the suspend signal
- To list all the signals, type kill -l
- The only signals mapped to control keys are:
  - CTRL-C as SIGINT
  - CTRL-Z as SIGTSTP
  - CTRL-\ as SIGQUIT (stronger than INT)
    - stty can map signals to keys

The kill command

- kill sends signals to processes
- By default, kill sends SIGTERM
- You can specify a signal by number or by name if preceeded by a dash
  - kill -HUP 2125
- You can refer to a job by its process ID (just a number) or its job ID (%number)

UNIVERSITY OF DENVER

The ps command

- ps is like ls, but for processes
- By default, it lists a PID, TTY, time, and command
    - The time is processor time so far used by the process
- We can pass args to ps to get more info:
    - Just man ps for details!

Some 'standard' ps args

- On the Linux systems, 'ps -e' lists all the processes by the user
- 'ps ax' does a similar thing, but includes all processes
- 'ps aux' adds user IDs

Trapping Signals

- Trapping signals can help your program deal with abnormal situations
- To trap signals, we use:
    - trap cmd sig1 sig2 ...
        - Here, cmd is the name of the command or function to call if one of the listed signals is reached
        - Execution returns to the command following the one where the signal was raised

Example Trap

- Here, the trap command defines a handler for INT
- The function inthandler is called whenever the process receives SIGINT
- Run it and try to kill it with CTRL-C

```
# trap SIGINT
trap inthandler INT

function inthandler
{
  echo “You hit CTRL-C!”
}

while true; do
  sleep 60
done
```

Ignoring a Signal

- The nohup command will cause the HUP signal to be ignored (called when you exit your shell)
- We can untrap a signal using -
  - trap - HUP

```
# Ignore any HUPs, similar
# to the nohup command
function ignorehup {
  trap "" HUP
  eval "$@"
}
```

Coroutines

- Let's say you have multiple cores and want to run commands simultaneously
  - We start each command in a script with &
  - However, as soon as the script continues, any remaining processes not complete will enter an orphaned state
    - foo &, bar, exit
      - If bar completes before foo, foo will become an orphan

Coroutines

- To fix this, we add a ‘wait’ command at the end
  - foo &; bar; wait
    - This forces the script to wait until all background scripts complete
    - wait can also take PID of the job
      - How do we get a PID of a process?

The PID variable

- $$ is always the process ID (PID) of the process that is running
- It's useful for making temporary files
  - cat 'junk' > /tmp/myfile$$

Subshells

- Instead of spawning multiple processes, we can also create subshells
  - The syntax of a subshell looks like a code block, but we use () instead
    - (exit); echo "testing"
      - Here, exit is run in a subshell, which doesn't cause the parent to terminate
      - subshells inherit environment variables, standard streams, signal traps and the current directory

More Tools

- Unix contains a host of programs that belong in your toolbox
- Over the next few slides, several of the more widely used tools will be presented

find

- ‘find’ is a command that searches the directory tree, performs operations, and can execute commands on results
    - Don’t forget: man find
- Basic syntax:
    - find <path> <expression>

Example Finds

- find . -name '*.txt'
  - Finds all the files from the current directory that end with .txt
- find . -name '*.swp' -exec rm {} \;
  - Finds all the files that end in .swp and removes them
    - {} is substituted with the filename, \; keeps bash from interpreting the ; on the command line

cutting things

- ‘cut’ is another simple utility that is useful for columned data
  - cut -d ‘:’ -f1 /etc/passwd
    - -d is the delimiter, -f is the field, which takes a list that is N, N-, N-M, or -M
      - that’s the nth column, nth to the end, nth to the mth, or 1st to the mth column
- By default, TAB is the delimiter

More tools

- ‘head’ lists the first lines of a file
  - head -n 20 myfile: lists the first 20 lines
- ‘tail’ lists the last lines of a file
  - tail myfile or tail -n 20 myfile lists the last 20 lines
- ‘sort’ sorts text files, various options can sort on columns, numerically, etc
  - sort myfile: by default it sorts each line alphanumerically

More tools...

- date: gives you the current date
- time: gives you the timing statistics for executing a command
- zdump: gives you time in a given time zone
- touch: creates a file or sets the modified time to the current time
- at: runs a job at a given time (usually for running a job just once)

More tools...

- sleep: suspends the process for some number of seconds
- cal: prints a calculator
- expr: an all-purpose calculator (just like $(()))
- dc: an arbitrary precision calculator that uses reverse polish notation (RPN)

More tools

- grep <pattern> file: searches for the regular expression in file and prints out the line which it's contained on
  - grep 'function foo' *.sh
- 'wc' gives word counts, line counts, byte counts, depending on the argument
  - wc -l myfile

UNIVERSITY OF DENVER

More tools

- ‘du’ will list disk usage--by default, it runs in your current directory
  - try du -h for more readable info
- And even more---where can you look?
  - /usr/bin, /usr/local/bin, /usr/sbin, /usr/local/sbin

getopts for better options

- To improve your ability to get options for your shell scripts, use getopts
- You give it letters that can be arguments (think -a -b)
- A colon after a letter means it needs an argument, which is put in $OPTARG

```
# the initial : here prevents silly
# error messages from getopts when
# it fails. opt is set to “?” if
# it was an illegal argument
while getopts “:ab:c” opt; do
  case $opt in
  a ) echo “arg a passed” ;;
  b ) echo “arg b with $OPTARG” ;;
  c ) echo “arg c passed” ;;
  \? ) echo ’usage: blah blah blah’
    exit 1
  esac
done
```

getopts continued

- getopts sets OPTIND to the argument number to be processed next each time it's called
- We can use a new command, 'shift', which left shifts all the arguments by a given number (1 by default)
  - Why do we need shift to do this? What use is it?
    - After using getopts, we may want to process the rest of the arguments, so we do a shift $((OPTIND - 1))
    - We also can't say 1=$2, for example

UNIVERSITY OF DENVER

Debugging bash scripts

- Here's a few things you can do now that your scripts are getting more sophisticated
  - Use the line set -o verbose or set -o xtrace at the start of your script
    - Verbose prints each line as it executes, xtrace prints the line with any substitutions in place

Fake Signals

- You can also trap 'fake' signals for debugging
  - EXIT, called when exit is called from the script
  - ERR, called when any command returns non-zero
    - saves the error code in $?, which you should save
  - DEBUG, called whenever the shell executes a statement
    - useful for monitoring a variable
  - RETURN , called when a script, function, or source finishes

Gotta catch ‘em all

- Not really, you just trap the ones you want
  - trap ‘echo script has exited’ EXIT
- Untrap them like other signals
  - trap - EXIT