Linux Shell Scripting

MAKING THE ADMIN'S LIFE THAT MUCH EASIER

About the Instructor

- Nathan Isburgh
  - nathan.isburgh@edgecloud.com
- Unix user 25+ years, teaching it 20+ years
- Unix Administration and Software Development Consultant
- All around über-geek
- Goofy, forgetful (remember that)

About the Course

- 4 days, lecture/lab format
  - Hours: 8:30 - 4:00
  - Lunch: 12:00 – 1:00
- Breaks about every hour
  - Throw something soft at me if I get too long in the tooth
- Telephone policy
  - Take it outside, please
- Restrooms
- Refreshments

About the Students

- Name?
- Time served, I mean employed, in the IT field?
- Department?
- General Unix skill level? What about Linux?
- And familiarity with Bash?
- How do you use Linux in your position?
- What are you hoping to take away from this class?

Expectations of Students

- Strong foundation in basic Linux use and administration
  - Preferably through RHCSA
- Strong understanding of working in the shell
- Ask Questions!
- Complete the labs
- Email if you're going to be late/miss class
- Have fun
- Learn something

Intentionally Left Blank

Scripting Basic Concepts

Overview

- There are several basic concepts about the shell and scripting which must be understood before tackling more complex problems
  - Basic shell syntax
  - Shebang syntax
  - Quoting
  - Exit status and subprocesses
  - Variables
  - Commenting

Shell Syntax

- Shell scripting is simply placing a sequence of shell commands into a file, for future "playback"
  - Obviously there are plenty of details, which is what we will be exploring in this course
  - At the end, though, it all boils down to shell commands
- Therefore, it follows that you must already have a strong foundation in basic shell syntax
  - Quoting
  - Environment variables
  - Commands

Scripting 101

- Simple shell scripts simply run command after command, as if the user typed them in at the command line
  - More complex shell scripts actually make decisions about what commands need to be run, and might even repeat certain sequences to accomplish a given task
- Scripts start executing at the top and stop when there are no more commands to execute or when `exit` is called
  - Or due to a syntax error!

Example

- Here is a very simple shell script to consider

```
echo "Hello, what is your name?"
read NAME
echo "Hello $NAME, it's nice to meet you!"
echo -n "The current time is: "
date
```

- Using the `echo` command, this script asks a question.
- The `read` command accepts input from the user and stores it in the environment variable `NAME`
- The script finishes up with a couple more `echo` statements, greeting the user and announcing today's date

Running The Example

- If we put the example in a file called `myscript`, we can execute the script as:
  - `bash myscript`
- Which instructs your interactive shell to start a *new* shell, `bash`, to open `myscript` and execute each line as if the user had typed it in manually
- Running in this way, bash operates as an interpreter
  - Reading each line of the file, bash would *interpret* the words and perform the given action
- There are many interpreted languages available for scripting, including all of the shells, python, ruby, perl, etc.

Interpreters

- Following this idea, to run a script, you simply feed the file to the appropriate interpreter
  - `bash mybashscript`
  - `perl myperlscript`
- This works fine, but sometimes it's more user-friendly to allow the script to be run directly, removing the need for an external call to the interpreter...
  - `./mybashscript`
  - `myperlscript`
- How is this done?

Shebang!

- This is accomplished with the *shebang* (#!), also known as a hash bang, pound bang or hashpling.
- The basic idea is very simple
- When the kernel is asked to execute a file, the content must either be machine code (compiled software), or a file that starts with the shebang sequence
- If the first two characters of the file are a hash mark and an exclamation mark (shebang!), the rest of the line is expected to be a pathname for an interpreter, which will then be invoked to "run" the file as a script
  - Connecting the script to stdin of the interpreter process

Back to the Example

- So, add an appropriate shebang to the example:

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
echo "Hello $NAME, it's nice to meet you!"
echo -n "The current time is: "
date
```

- Then add execute perms so the script can be run directly:

```
[root@localhost ~]# chmod a+x myscript
[root@localhost ~]# ./myscript
Hello, what is your name?
Linus
Hello Linus, it's nice to meet you!
The current time is: Sun Jul 21 09:39:33 CDT 2013
[root@localhost ~]#
```

Details to Note

- Note the use of quoting in the example
  - Remember that everything in a shell script must follow shell syntax!
- If something would need to be quoted on the command line (due to whitespace or metacharacters), it will also need to be quoted in the shell script
- In addition to single and double quotes, remember your escape character: \ (the backslash)
  - Do you know the difference between the quoting mechanisms?

Exit Status

- Another important detail to internalize when shell scripting is the importance of exit codes (or statuses)
- Every single time a process is finished executing, it notifies the kernel via an `exit` system call
- There is a required parameter to the exit system call, known as the exit status
- The exit status is a number, and there are only two values meaningful to the kernel and shells:
  - Zero: Zero means a *successful* application exit
  - Non-Zero: Any non zero exit status implies a failure of some sort

Exit Status and Scripting

- The reason that the exit status is so important to shell scripting is because *all of the shell features used in scripting* are based on exit status
  - Conditionals
  - Looping
  - Intelligent command separators
- Note that the actual non-zero values a program might use, such as 14, -8, 2, etc, do not have standard meanings
  - The documentation for an application might specify the meaning of particular exit codes, which can then be checked in a script through the `$?` special environment variable

Variables

- Variables in shell scripting are nothing more than standard environment variables
- This is convenient; the known rules and capabilities apply
  - `NAME=value`
  - `NAME="quoted value"`
  - `ls $NAME`
  - `echo Hello ${NAME:-Sir/Madam},may I help you\?`
- The `set` and `env` commands are useful
- See bash manpage under heading "Parameter Expansion"

Commenting

- Commenting falls under the larger topic of coding style, which could be a class unto itself
  - Note that style is an individual attribute, developed over time as a software developer
  - It is also often lightly or strictly specified by organization
- To simplify this discussion, let us recall the Golden Rules of Commenting…

The Golden Rules of Commenting

- Always comment code which is not obvious to a non-author reader
  - You should not comment "`i=i+1`"
  - You should comment "`rsync -vazpc $WHAT $WHERE`"
- Always comment functions: their purpose, use, arguments, expectations and results
- Always comment the overall program's purpose and behavior at the top of the file
  - Include dates and authors (maybe an abbreviated revision history?)
- Always comment when not sure if you should
  - They don't cost anything!

Lab

- Write a basic "Hello world" shell script
  - The script should greet the user by name, then welcome him to the world of scripting. Consider commands or environment variables which might obtain the user's login name.
  - Match the following output format, substituting the underlined values appropriately:
    - `Hello nisburgh. Welcome to the world of scripting.`
    - `The current date is Tuesday, July 04, 2017.`
- Follow all of the guidelines discussed
  - Make it a standalone executable using the shebang syntax
  - Comment appropriately
- Read documentation for assistance

Useful Tools in Scripting

Overview

- There are, of course, many, many tools to use while scripting, but some are more powerful, or more frequently used
- We will overview three of these tools now:
  - `awk`
  - `sed`
  - `xargs`

awk

- awk is an incredibly powerful tool, which contains it's own programming language
- One of the most commonly used features of awk, is to grab particular columns of information from stdin
- Consider the columns from `ps aux`:
  - `USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND`
- Using awk, grabbing particular columns is easy!
  - `ps aux | awk '{print $2" is using "$3"% of the CPU"}'`

sed

- sed stands for "stream editor" and that is exactly what it does
- sed takes an expression describing an operation to perform, and then applies that operation to each line of input
- It is commonly used to perform find/replace operations:

```
sed -e 's/example.com/mycompany.com/' /etc/httpd/conf/httpd.conf
```

- This example replaces every occurrence of example.com with mycompany.com
- sed can do way, waaaay more. Consult a google or get the O'Reilly book: "Sed and Awk"

xargs

- `xargs` is another very useful tool at the command line, and in scripting
- It takes a second to wrap your head around what `xargs` does:
  - Accepts input from stdin
  - For each line or lines of input, run a given command with the input lines as *arguments* for the command
- For example:

```
ps aux | fgrep bad_cron | awk '{print $2}' | xargs kill
```

- Let's discuss what's happening with the example

Intentionally Left Blank

Conditionals

To Execute or Not To Execute

- More advanced problems require the script to make decisions. There are two basic ways to make decisions with shell scripts:
- `if` statements
  - The most basic and powerful conditional
  - "If some condition is true, then do these things"
- `case` statements
  - A streamlined version of an if statement, mainly used to improve readability and maintenance of code
  - "Taking a given input and several possible values I'm interested in, which one matches? Then do these things based on that match"

The test Command

- Before we continue talking about decisions, we need to talk about the `test` command. This command actually performs the comparisons necessary to ask many common questions, such as:
  - `"string1" = "string2"` — *Is string1 identical to string2*
  - `$VAR -lt 45` — *Is $VAR numerically less than 45*
  - `-e path` — *Does path exists*
- The result of the test is in the exit status
  - *True* Exit 0
  - *False* Exit 1
- See the man page on `test` for additional details and more flags; there are many tests it can perform

The test and [Commands

- The `test` command has a functionally identical sibling:
  - The `[` command
- The `[` command is provided for improved readability in scripts – it has no additional features beyond `test`
- In bash, the `test` and `[` commands are actually built-in
  - They are also available as standalone binaries from the coreutils package, in the `/bin` folder

Shell Conditionals

- Identical to the `test` command, the shell can perform the same conditional checks using the `[[expression]]` syntax
- Consider:
  - `[[ -d /tmp/mytool ]] && mv logfile /tmp/mytool`
- See Conditional Expressions in the manpage for a complete reference on all of the available tests

The if Statement

- Basic syntax:

```
if list
     then list
     [ elif list
          then list ]
     ...
     [ else list ]
fi
```

Example

```
#!/bin/bash
echo “Hello, what is your name?”
read NAME
if [ “$NAME” = “Linus” ]
then
  echo “Greetings, Creator!”
elif [ “$NAME” = “Bill” ]
then
  echo “Take your M$ elsewhere!”
  exit
else
  echo “Hello $NAME, it’s nice to meet you!”
fi
echo -n “The current time is: “
date
```

- This script bases it’s response on the name given

The case Statement

- Basic syntax

```
case word in
     pattern ) list;;
     ...
esac
```

Example

```
#!/bin/bash
echo “Hello, what is your name?”
read NAME
case ”$NAME” in
  “Linus” )
    echo “Greetings, Creator!”
    ;;
  “Bill” )
    echo “Take your M$ elsewhere!”
    exit
    ;;
  * )
    echo “Hello $NAME, it’s nice to meet you!”
esac
echo -n “The current time is: “
date
```

- This script maintains identical behavior, but uses a case statement

Lab

- Write a shell script which uses an if statement to print a special message on the first and fifteenth of the month:
  - If it is the first or fifteenth of the month, the script should print:
    - `YAY! Payday!`
  - Otherwise, it should print:
    - `Boo.. Not yet payday..`
- To test, simply change the date of your machine
  - Check the first, second, tenth, eleventh, fifteenth, and twenty first
- Remember to comment appropriately

Looping

Looping

- Sometimes a certain sequence of commands need to be run repeatedly, either for a set number of times or while some condition is true. This is accomplished with:
- `while` loops
  - Most common and powerful loop form
  - "Check some condition and if true, run these commands. Then check again and if still true, run these commands again. Repeat until the condition is no longer true."
- `for` loops
  - Simple method for looping a given number of times or over a list
  - "Do this X times."
  - "Do this for each item in a list"

The while Loop

- The `while` loop is the most common, but be aware it has a brother, the `until` loop
  - The `until` loop is identical in operation, but the conditional requirements are reversed; execute while the conditional is *false*
- Basic while/until syntax:

```
while list;
      do list;
done
```

Example

```
#!/bin/bash
echo “Hello, what is your name?”
read NAME
while [ “$NAME” != “Linus” ]
do
  echo “I don’t know that person, what is your name?”
  read NAME
done
echo “Greetings, Creator!”
echo -n “The current time is: “
date
```

- This script will loop until the given name is “Linus”

The for Loop

- There are two major forms of the for loop
- Basic syntax of the first:

```
for (( expr1 ; expr2 ; expr3 ))
     do list;
done
```

Expressions `expr[1-3]` are defined as:

`expr1`	Initialization (sets up loop controls)
`expr2`	Conditional (defines loop requirements)
`expr3`	Iteration (steps controls towards requirements)

Example

```
#!/bin/bash
echo “Hello, what is your name?”
read NAME
for (( I=0 ; I<3 ; I++ ))
do
  echo “Hello $NAME!!”
done
echo -n “The current time is: “
date
```

- This goofy script repeats your name 3 times before giving you the date and time

The for Loop

- The second form iterates over items in a list
- Basic syntax:

```
for name in word ...;
     do list;
done
```

Example

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
for TIME in Three Two One
do
  echo "$TIME"
  sleep 1
done
echo "Hello $NAME!!"
echo -n "The current time is: "
date
```

- This goofy script counts down "Three…Two…One…" then yells the given name, followed by the date and time
- Note that you can execute a subcommand with the back quotes, and each line will become a list item:

```
for item in `ls /tmp`
```

Lab

- Write a script which uses loops and conditionals to announce the time every 10 seconds, on even 10 second divisions (0, 10, 20, 30, 40, 50)
  - It is 1:01:30pm!
  - It is 1:01:40pm!
  - *Etc…*
- Think of efficient ways to perform this operation, such as sleep statements. Do not "spin." Spinning is when a program runs as fast as it can in a loop waiting on some event to occur, rather than using more intelligent behavior such as alarms, blocks and timers to conserve CPU resources

Intentionally Left Blank

Special Variables

Special Variables

- The shell has many special variables to contain information
  - Positional parameters (arguments)
  - Exit status of previous command
  - Bash information
- There are also several ways of getting at the values of variables, known as parameter expansion

Positional Parameters

- The positional parameters are the arguments to the script or a function
- They are assigned numerically, left to right
  - `script argA argB argC`
  - `$0` is the script name
  - `$1` is argA
  - `$2` is argB
  - `$3` is argC
- Also, there are a couple of related special variables
  - `$#` is the total number of arguments (not including `$0`)
  - `$@` expands to a space separated list of all arguments

Exit Status

- The exit status of the previously executed command can be obtained through the `$?` variable
- It is important to consider the meaning of this variable
- Every time you execute a command, it changes
  - If you `echo $?`, by the following line it's different already (the exit code of echo)
- For this reason, you will often see scripters storing the value in another variable for future examination:
  - *`command with important exit status`*
  - `ESTAT=$?`
  - `if [ $ESTAT -eq 5 ] ...`

Bash Information

- There are dozens of informational variables which are maintained by bash, including some useful ones:
  - `HOSTNAME`
  - `PWD`
  - `UID`
  - `BASHPID`
  - `BASH_VERSION`
- For a complete list of variables, see the manpage under various headings, including "Special Parameters" and "Shell Variables"

Expanding Variables

- Variables have several methods of expansion to values
  - `$NAME`
  - `${NAME}` to be more precise, or embed in another term
  - `${NAME:-word}` will expand to word if NAME is not set or null
  - `${NAME:=word}` will expand to and assign NAME to word if NAME is not already set or null
  - `${NAME:?word}` will fail with an error message of word if NAME is not set or null
  - `${NAME:offset:length}` fetches length characters from NAME starting at offset
  - `${#NAME}` returns character length for value of NAME
- See manpage under "Parameter Expansion" for complete details and additional options

Lab

- Modify the lab from the Loops module to accept two optional parameters
  - The number of total announcements to make before exiting (originally it would run forever, which should be the default)
  - A yes or a no, which indicates whether or not to also print the date with the announcement. Default of yes
- Example:
  - `myscript 5 yes`
  - Would report 5 times and exit, and each report line would say something along the lines of:
    - `It is 4:32:20pm, August 4, 2017!`

Intentionally Left Blank

Functions

Overview

- Functions are an important component of code organization and reuse
- A function allows you to group a series of statements under a name, then *call* the function at any time to execute the collected statements
- You can also pass arguments to the function for it to operate on
- Further, the function can return a value to the caller, indicating status or results

Example

```
#!/bin/bash

sayhello() {
  echo Hello $1
  return 5
}

sayhello Bob
```

- This script defines a function called `sayhello`, which it then uses to say hello to Bob
- Note how arguments are passed (through standard positional parameters)
- Note how a return value is generated
  - Default is the exit status of last command executed by function

Using Functions

- Functions are often collected in a file, and used by multiple scripts as a *library*
- To use a library, you need to *source* the file, using either:

```
source path-to-library
. path-to-library
```

- For an example, see the startup scripts in the `init.d` folder
- They all use the `/etc/init.d/functions` library for common operations like starting a service

Lab

- Modify the lab from the Special Variables module such that the reporting functionality is wrapped in one or more functions
- Place the function(s) in a library
- Get creative and add a few more functions to encompass some silly behaviors like using names, printing banners or doing file operations with redirection
- Write a new script which uses the library to offer behaviors to the user through a simple menu system

Intentionally Left Blank

Scripting Best Practices

Best Practices

- As our scripts grow in complexity, and get shared with colleagues, there are some guidelines which can help improve readability and maintainability
- This will produce more professional, less buggy scripts with greater features
- In addition, good code design leads to good code reuse, improving ROI
- A Style Guideline should be established within the organization, mandating basic requirements suitable across languages

Commenting

- Remember from an earlier lecture:
- Commenting falls under the larger topic of coding style
  - Note that style is an individual attribute, developed over time as a software developer
  - It is also often lightly or strictly specified by organization
- As a starting point, consider the Golden Rules of Commenting, but keep in mind plans for a style guideline

Proper Script Structure

- Scripts should *generally* be laid out as:

```
#!Shebang!
#
# Script comment block ( purpose, arguments, rev history, etc )
#

# Config variables with comments
CONFIG_VAR1=”user can tweak this”

# END OF CONFIGURATION – NO EDITS BELOW THIS LINE

# Function definitions
fail() { echo boohoo ; exit -1 }

# Main code block
if [ $# -lt 2 ] ; then fail ; fi
...
```

Always Initialize Variables

- You should always initialize your variables
  - It looks cleaner, and for complex scripts, a short comment can be left indicating the purpose of the variable
  - Security! If variables aren't initialized, an educated user can easily pre-initialize a variable from the command line and cause all sorts of problems, some maybe nefarious!

Indentation

- Ah yes, good old indentation
  - Many a bloody nerd war has erupted over disagreements on indentation styles
- To avoid this same fate, let us agree on one simple rule:
  - Pick an indentation style, and stick to it 100% of the time
- The possibilities are endless:
  - Tabs, two spaces, four spaces? *Suggest: 2 spaces*
  - Indent all the blocks, only the multiline blocks, or? *Suggest: all*
  - Reserved words: same line, different lines, indented? *Suggest: different lines, indent the blocks only*
  - Etc, etc, etc

Check Those Arguments

- Users rarely do anything right – train yourself to expect that at all times, and you'll write better code. ☺
- Case in point: Arguments
  - Check for the expected number of arguments
  - Check for the expected types of data: numbers, strings, flags
  - Check argument *values* if appropriate, eg: if it is supposed to be a pathname, check that it's valid and exists
- On very large or complex scripts with many arguments, it might be prudent to consider an argument parsing library like `getopt` (external program, some inconsistencies) or `getopts` (shell builtin, consistent but no long arguments)

Check Commands and Versions

- If a script uses tools that are even remotely uncommon, it should check for their existence early on and error out if anything is missing
- Along the same lines, if there are any feature expectations, or important bug fixes tied to a version of a tool, library or even the shell itself, those version details should be verified early on
  - Note that this requires a judgment call – there is no need to check version information on every piece of software touched – just the ones that could be off. For example:
    - If a script relies on associative arrays, it should check that the bash interpreter is at least version 4 (EL5 ships with v3!)

Assign Exit Codes

- Exit codes can be extremely useful to the users of your script
- At the very least, always exit 0 for success and non-zero for failure
- Best case scenario: assign exit codes to different conditions, eg
  - 1: invalid arguments
  - 2: insufficient permissions
  - 3: missing required software
  - 4: httpd not running
  - 5: unknown error

Write Common Functions

- Write some common, useful functions, such as:
  - `fail(code, msg)` – Prints message to stderr and exits with given code
  - `succeed()` – Maybe print happy message, then exit 0
  - `cleanup()` – For complex scripts, cleanup things like logs, locks, etc. Usually called from `fail()` and `succeed()`
  - `debug(msg)` – Prints a debug message to stderr. Bonus: use a config variable and/or command line flag to control behavior
  - `usage()` – Print a detailed usage message to the user if there is a mistake in arguments, or -h/-? Passed
- Perhaps a good case for a *library*…

stderr

- USE IT! Correctly!
- Recall:
  - `stdout` – Normal command output/results
  - `stderr` – Warnings, errors, fails of any kind
- Quick and easy ways to output to stderr:
  - `printf blah > /dev/stderr`
  - `printf blah >&2`
- This is one of the benefits of writing those common functions!

Command Substitution

- Recall the awesomely powerful backtick, `
  - It runs the command in backtacks, takes its stdout and substitutes it, minus any trailing newlines, onto the calling command line
  - `echo `whoami``
    - becomes
  - `echo student`
- Very useful in many situations, and it is backwards compatible with some older shells
- But…

Command Substitution

- Try to avoid the backtick for command substitution
  - It is not POSIX compliant
  - It does not nest properly
  - Quotes can be a serious pain
- Instead, use the `$()` syntax:
  - `echo $(whoami)`
- Same behavior, but:
  - POSIX compliant
  - Nests
  - Handles quotes much more simply

Lab

- Put together a properly styled skeleton for a shell script, called `skel.sh`
- This should include:
  - All of the components discussed in lecture, and placeholders for the pieces which are not known yet (like config variables)
  - The various common functions
  - Come up with at least five common script failures, and assign them default exit codes (example: 'invalid arguments' assigned -2)
- Copy `skel.sh` to `health-report.sh`, with synopsis:
  - `./health-report.sh [-td] email`
  - `-t` will email one output iteration from top to the email address
  - `-d` will email the output of 'df -h' to the email address
  - `email` is the email address for the recipient of the report

Advanced Variables

Special Variables

- Recall that the shell has many special variables with useful information and settings
  - Positional parameters (arguments)
  - Exit status of previous command
  - Bash information
  - Feature control variables (IFS, OPT*, DIRSTACK, etc)
- During future labs, be sure to peruse the bash man page sections on:
  - Special Parameters - `@, #, ?, $, -`
  - Shell Variables - `LINENO, SECONDS, PIPESTATUS`
  - Parameter Substitution - `${#PATH}, ${INPUT:5:10}`

Arrays

- In addition to simple variables containing just strings and numbers, bash also supports array variables
- An array is just a *collection* of values, all stored within one variable, logically:
  - TEST → val1,val2,val3,val4,val5
- Traditionally, the different values in the array are referenced using numbers, called indexes, starting at zero:
  - TEST[0] → val1
  - TEST[1] → val2
  - …
- This is known as an Indexed Array

Indexed Array Example

```
# To create the array, just start assigning values:
MYDIRS[0]="/"
MYDIRS[1]="/home"
MYDIRS[2]="/usr"

echo $MYDIRS
# will just show "/" since that is first member

echo ${MYDIRS[1]}
# will show "/home"
# Note that you must use the braced expansion syntax, due to
# overloading of the square bracket characters ( pathname wildcard )

echo ${#MYDIRS[*]}
# shows 3, since there are three values in the array
```

Associative Arrays

- As of bash version 4, Associative Arrays are available
- An associative array uses *strings* to get at values, as opposed to numbers
- Associative arrays have to be created specially, using the declare builtin

```
declare -A MYDICTIONARY

MYDICTIONARY[apple]=fruit
MYDICTIONARY[carrot]=vegetable
MYDICTIONARY[linux]="Awesome operating system"
MYDICTIONARY[windows]="An operating system"
```

Lab

- Copy `skel.sh` to `proc-count.sh` and implement as:
  - `proc-count.sh [-f filter]... [-c] email`
- This script will count processes with command names that match one or more filters, emailing one of two possible reports, either a TSV (which is default) or a CSV (selected with the -c flag)
  - `filter  count`
    - *Or*
  - `filter,count`
- If no filter is given, all processes should be reported
- Use arrays as appropriate (perhaps filters, results?)

Advanced Expansions

Overview

- An expansion occurs when the shell acts on metacharacters in a command to automatically expand their contents based on rules
  - Sometimes, so the user does not have to type as much (wildcards)
  - Other times, to reference variables and other shell features
- There are seven different kinds of expansions in bash:
  - Brace expansion, tilde expansion, parameter/variable expansion, command substitution, arithmetic expansion, word splitting, and pathname expansion
- On operating systems that support named pipes (like Linux!), there is one additional form, known as process substitution

Brace Expansion

- Brace expansion allows for the automatic creation of arbitrary strings
- Consider:
  - `echo a{1..5}b`
    - `a1b a2b a3b a4b a5b`
  - `echo a{f,h,g}b`
    - `afb ahb agb`
- As seen in the examples, you can expand ranges of numbers or letters, as well as comma separated lists of values

Tilde Expansion

- You should already be familiar with tilde expansion, which evaluate to user home directories:
  - echo ~
    - /home/student
  - echo ~alice
    - /home/alice
- What you might not know is that tilde can be used to reference current directories (~+) and previous directories (~-):
  - `cd /home ; cd / ; echo ~+ ; echo ~-`
    - /
    - `/home`
  - *Started in `/home`, then moved to `/`. `~+` expanded to `/`, `~-` expanded to `/home`*

Parameter/Variable Expansion

- This topic was covered in depth previously
- Quick reminder:
  - `$PATH`
  - `${PATH}`
- The second form is more precise, and should generally be used anytime a variable reference is embedded within additional content, to protect from misinterpretation
  - Also note, the curly brace expansion syntax allows for extremely powerful capabilities, including arrays, searching, substrings, character counts, case manipulation and more

Command Substitution

- Command substitution is incredibly useful, as it instructs the shell to run a given command in a new shell, and capture its output in some particular manner
- Recall the backtick and $() from an earlier lecture:
  - `echo `whoami``
  - `echo $(whoami)`
- `whoami` will be run from a new shell, and it's standard output, minus any trailing newlines, will be substituted into the quoted/parenthesis section of the command line, which is then executed from the main shell, as:
  - `echo student`

Arithmetic Expansion

- Sometimes, it's incredibly useful to have the shell perform some simple math, and it's also incredibly easy to use:
  - `echo $((6*8))`
    - `48`
- Bash has a slew of operations available, including add/subtract/multiply/divide, exponentiation, bitwise operations including shifts, negations and logical operations, increments, decrements and more
- See the manpage under Arithmetic Evaluation

Word Splitting

- Word splitting is an interesting feature of the shell, that allows it to identify *words* within a parameter expansion, command substitution and arithmetic evaluation, and then split them out
- There is a shell variable known as `IFS`, which stands for Internal Field Separator
  - This variable defines the characters which can separate words, and the default IFS is '<space><tab><newline>'
  - Also note that the first character of IFS is used to separate the found words during splitting
- Try the following:
  - `echo $(w)`

Pathname Expansion

- Pathname expansion is nerd-speak for how wildcards work in the shell
- This shouldn't require review, but recall the three wildcards:
  - `*`
  - `?`
  - `[set]`

Process Substitution

- Process substitution is a very neat shorthand for dynamically creating named pipes which are used for input or output
- Consider the first form:

```
cat /etc/passwd <(w) <(df -h) <(uname -a) > report
```

- The <( ) syntax creates the process substitutions
- What's really going on here, is that the inner command is executed, with its stdout connected to a named pipe dynamically created under /dev/fd
- That pathname is then substituted on the outer command line, which becomes an argument, and in this case, cat simply reads from the /dev/fd file like any other
  - Try: `echo cat /etc/passwd <(w) <(df -h) <(uname -a) \> report`

Process Substitution

- The second form of process substitution is similar, except the other direction
- The /dev/fd file is created to accept input from the outer command, and the file is attached as stdin on the inner command
- Consider:
  - `tar cf - . > >(gzip -9c > crazy.tgz)`
- It looks crazy, but just step through the operations
  - tar is outputting to stdout, which is redirected to the process substitution (which in reality is a /dev/fd pathname)
  - gzip is reading from stdin, which is the /dev/fd path

Sample Code

- Next, we will review various snippets of sample code
  - Learn from the good ideas, avoid the bad ones
  - Break down functionality
  - Explain behavior
  - Identify potential bugs or concerns

Sample Code – Swap Summarization

```
#!/bin/bash
# Show swap memory usage per-process

(
  echo "PID Mem(kB) Binary"
  for x in `ls /proc/ | grep -e '^[0-9][0-9]*$'`; do
    PID=$x
    SWAP=`grep VmSwap /proc/$x/status | awk '{print $2}'`
    PROC=`ps aux | awk '$2 ~ /^'$PID'$/ {print $11}'`
    if [ ! -z "$SWAP" ]; then
      echo "$PID $SWAP $PROC"
    fi
  done 2>/dev/null | sort -nk 2
) | column -t
```

Sample Code – Recursive Configs

```
APACHEDIR=/etc/httpd
TMPDIR=/tmp/conflist
FORE=1
AFT=0

cd $APACHEDIR
mkdir -p $TMPDIR
rm -f $TMPDIR/authlist
touch $TMPDIR/authlist
echo "conf/httpd.conf" > $TMPDIR/newfinds

while [[ $FORE != $AFT ]]; do
  FORE=`cat $TMPDIR/authlist | wc -l`

  rm -f $TMPDIR/grepping
  mv $TMPDIR/newfinds $TMPDIR/grepping

  for x in `cat $TMPDIR/grepping`; do
    grep -Ei '^\s*include\s' $x | awk '{print $2}' >> $TMPDIR/newfinds
  done

  cat $TMPDIR/grepping $TMPDIR/authlist $TMPDIR/newfinds | sort -u > $TMPDIR/tmp
  rm -f $TMPDIR/authlist
  mv $TMPDIR/tmp $TMPDIR/authlist
  AFT=`cat $TMPDIR/authlist | wc -l`
done

for x in `cat $TMPDIR/authlist`; do echo $x; done | sed '/^\//!s/^/\/etc\/httpd\//'
```

Sample Code – Trapping and Locks

```
LOCK_FILE=/tmp/`basename $0`.lock
function cleanup {
 echo "Caught exit signal - deleting trap file"
 rm -f $LOCK_FILE
 exit 2
}
trap 'cleanup' 1 2 9 15 17 19 23 EXIT
(set -C; : > $LOCK_FILE) 2> /dev/null
if [ $? != "0" ]; then
 echo "Lock File exists - exiting"
 exit 1
fi

#########################################
###         Main Script Body          ###
#########################################
```

Sample Code – Expansions and Math

```
PARENTPIDS=`comm -12 <(ps -C httpd -C apache2 -o ppid | sort -u) <(ps -C httpd -C apache2 -o pid | sort -u)`

for ParPID in $PARENTPIDS; do
  SUM=0
  COUNT=0
  for x in `ps f --ppid $ParPID -o rss | tail -n +2`; do
    SUM=$(( $SUM + $x ))
    COUNT=$(( $COUNT + 1 ))
  done

  MEMPP=$(( $SUM / $COUNT / 1024 ))
  FREERAM=$(( `free | tail -2 | head -1 | awk '{print $4}'` / 1024 ))
  APACHERAM=$(( $SUM / 1024 ))
  APACHEMAX=$(( $APACHERAM + $FREERAM ))

  (
  echo
  echo "Info for the following parent apache process:"
  echo "  "`ps f --pid $ParPID -o command | tail -n +2`
  echo
  echo "Current # of apache processes:        $COUNT"
  echo "Average memory per apache process:    $MEMPP MB"
  echo "Free RAM (including cache & buffers): $FREERAM MB"
  echo "RAM currently in use by apache:       $APACHERAM MB"
  echo "Max RAM available to apache:          $APACHEMAX MB"
  echo
  echo "Theoretical maximum MaxClients:  $(( $APACHEMAX / $MEMPP ))"
  echo "Recommended MaxClients:          $(( $APACHEMAX / 10 * 9 / $MEMPP ))"
  echo
  )
done
```

Lab

- Modify `health-report.sh` from the earlier lab:
  - Add a new flag, `-m`, to create a list of process names and memory percentages, sorted descending by memory usage.
  - Also, add a `-c` flag to indicate "collect only" mode. The user should not need to supply an email in this mode. In this mode, the script should produce the requested reports (from the other flags), but instead of emailing them immediately, it should collect them in a file under `/tmp` called `health-report.YYYY-MM-DD`
    - You can simply append each new report to the file, but include a header in front of each new report that has the date/time
  - Finally, add a `-r` flag which accepts a date in `YYYY-MM-DD` form, and emails the requested report to the supplied email address
  - Make sure to produce meaningful error messages for all failures

Intentionally Left Blank

Additional Topics

Overview

- There are a few other topics that should be covered, but did not fall under any of the previous topics
  - Here documents
  - Subshell executions
  - Command separators / control operators
  - Trapping signals
  - Terminal codes to get colors and special modes
  - Automagic logging with coproc

Here documents

- Here documents are a really convenient way to enter multiple lines of text at the command line, or from within a shell script
- Usage is fairly simple:

```
cat << samp
Everything I type will go to
stdin of the command
Until a line with just samp
samp
```

- Just be careful about spacing – everything is literal, and the delimiter (samp in this example) must not have anything else on the line
  - Also see <<– for indenting here documents

Subshell Executions

- Sometimes, it is convenient to execute a command within a subshell, which isolates it from the current shell
  - It can not impact the environment or working directory of the current shell
  - You can treat the subshell as an individual command, using redirection and pipes as needed
- Simple example:
  - `( cd /home ; ls a* ) | wc -l`
- This will list a count of the home directories starting with the letter a. The cd did not change the working directory of the main shell

Command Separators / Control Operators

- There are several ways to separate commands:
  - Semicolon (`;`)
    - This separates commands and does not provide any relation between the commands. They are simply executed one after another, left to right.
  - Ampersand (`&`)
    - This puts the left command in the background and starts executing the next command immediately
  - Double Ampersand (`&&`)
    - This will execute the right command if the left command exited with a zero/success
  - Double pipe (`||`)
    - This will execute the right command if the left command exited with a non-zero/fail

Trapping Signals

- Sometimes, it's useful to react to signals when they get delivered to your script by the kernel
- This is easily done with the `trap` command:
  - `trap "echo DING" ALRM`
  - `kill -ALRM $$`
- This instructs the shell to run the `echo` command when an alarm signal is delivered
- This technique is commonly used to trigger cleanup routines when the script is interrupted
  - One of the examples contained a good illustration of this technique

Terminal codes

- Most terminals support various colors and modes to display information to the user
- If you find the codes for the connected terminal, you can output text with different foreground and background colors, blinking, dim, underlined and more
- A common technique for this is to use hard coded codes in your strings:

```
echo -e "\033[31mRed\033[39m and \033[32mGreen\033[39m"
```

- This gets hard to read and do correctly, so variables are commonly employed

Terminal Codes with Variables

- Using Variables:

```
RED="\033[31m"
GREEN="\033[32m"
NORMAL="\033[39m"
echo -e "${RED}Red${NORMAL} and ${GREEN}Green${NORMAL}"
```

- With variables, things are a little easier to read, and the codes can be changed with the terminal
- Could functions help here too?
- For documentation and examples:
  - http://wiki.bash-hackers.org/scripting/terminalcodes

Automagic Logging

- This is a really neat trick to attach stdout of your script to both the terminal and a logfile at the same time

```
#!/bin/bash
# we start tee in the background
# redirecting its output to the stdout of the script
{ coproc tee { tee logfile ;} >&3 ;} 3>&1
# we redirect stdin and stdout of the script to our coprocess
exec >&${tee[1]} 2>&1
```

Final Lab

- Copy `skel.sh` to `kill-thread.sh`
- Implement `kill-thread.sh` to kill mysql connections based on certain parameters:

```
kill-thread.sh [-u user] [-h host] [-d db] [-c command]
```

- Just use `mysqladmin`, and assume there is no root password, or it is supplied by `~/.my.cnf` automatically
- Just do simple searches by the various columns, and if the user supplies more than one flag, all must match to kill the connection
- `./kill-thread.sh -h localhost -d test`
  - Would kill anyone connected to the test database from localhost
- Snaz up the output with colors!