CST8177 – Linux II

More Scripting Techniques
Scripting Process
Example Script

scripting so far

- arguments to scripts
- positional parameters
- input using `read`
- exit status
- `test` program, also known as `[`
- `if` statements
- error messages
- `while` loops
- `for` loops

scripting to come

- why write scripts in the first place?
- lists
- set and shift
- integer arithmetic
- scripting process in general

Why do we write scripts?

1. use when an alias gets too complex, impossible
   - aliases cannot use arguments in their replacement text
2. make new, specialized commands
   - look at /bin/gunzip and /bin/zcat
3. automate long and/or complex tasks
   - example, actually part of the boot system: /etc/rc.d/rc
   - backup scripts, bulk user creation, other sysadmin tasks
   - as we'll see, scripts are the contents of the "regularly scheduled tasks" directories:

     /etc/cron.{hourly,daily,monthly}/

Lists

- In the general form of the control statements(examples: `if`, `for`, `while`) in the bash man page we will see the notion of a **list**
  - `for name [ in word... ] ; do list ; done`
  - `if list1; then list2; fi`
- **From the man page for bash:**
  - A `list` is a sequence of one or more pipelines separated by one of the operators `;`, `&`, `&&`, or `||`, and optionally terminated by one of `;`, `&`, or <newline>.

Lists (cont'd)

- Hey, we already know what most of those mean:
  - `ls -al | more` # there's that | operator
    - | means redirect output of one command into another
  - `ls -al > foo &` # there's that & operator
    - & means run the command in the background
    - proceed immediately with any command after the &
  - `ls -al ; head myfile` # there's the ; operator
    - ; means run one command after another
    - wait for the first command to finish in the foreground, then when it's finished, proceed with any command after the ;

&& and ||

- The new ones are
  - command1 && command2
  - command1 || command2

&& example

- && means AND
- This means stop at the first command that has non-zero exit status
- command1 && command2 is the same as
  - if command1; then command2; fi
  - example, suppose we want to run process.sh on a file, and then remove it:

```
process.sh /root/testfile && rm -f /root/testfile
```

    - if any thing goes wrong with process.sh (non-zero exit status), the file is NOT removed
    - if the **process.sh /root/testfile** command goes well (has zero exit status), THEN the file will be removed

|| example

- || means OR
- This means stop at the first command that returns a zero exit status
- command1 || command2 is the same as
  - if command1; then : ; else command2 fi
    - : is a command that does nothing!
  - example, suppose we are writing a script that needs to process a file, and if that doesn't go well, then the script should terminate abnormally:
    - `process /root/testfile || exit 1`
    - if anything goes wrong with `process`, the script exits

&& means "and"

- Suppose you might qualify for a scholarship:
- Those who qualify are:
  - born on the moon, and *something*
    - if we weren't born on the moon, we don't need to know what the <something> is; we already know we don't qualify
  - Algonquin student and *something*
    - in this case, because we are an Algonquin student, we do need to know what *something* is to know whether we qualify
- or in other words
  - born on the moon `&&` *something*
  - Algonquin student `&&` *something*

|| means "or", opposite of &&

- As soon as we encounter "true", we can stop
- You qualify for a $1000 rebate under the following conditions:
  - born on the moon, or *??*
  - Algonquin student, or *??*
- In the first case, we need to know what the exit status of the *??* is, we need to run the *??* command
- In the second case, we can stop before running the *??* command

&& and || versus -a and -o

- && and || are used with commands that tend to get things done
  - to graduate, you
    - complete first year && complete second year
  - complete first year is a "command" that gets things done: you learn the first-year material
  - if you fail first year, you don't attempt second year
- `-a` and `-o` are used in test, and don't do things, just affect the exit status of test
  - you are a rich Canadian if
    - you are Canadian -a you are rich
  - checking whether or not you're Canadian doesn't get things done – but it does establish a truth value

Manipulating positional parameters

- In our script we can manipulate the positional parameters
- `set` command to set them
- `shift` command to process them in turn

set

- We can set the positional parameters with
  - `set - oneword secondword -e etc`
- This will set
  - `$1` to `oneword`
  - `$2` to `secondword`
  - `$3` to `-e`
  - `$4` to `etc`
- Set does take options (see bash manpage) and that first `-` tells set there are no more options (`-e` in this case is not an option)

shift

- shift
- moves all the arguments to the left
- shift n moves all the arguments to the left by n
- shift
  - $# is decreased by 1
  - the pre-shift $1 is lost
  - $1 becomes what was in $2
  - $2 becomes what was in $3
  - $3 becomes what was in $4
    - etc

Doing integer arithmetic

- The `expr` command is a common and traditional way to evaluate arithmetic expressions
- The arguments to the `expr` command constitute the arithmetic expression

expr examples

- examples of using `expr` command:

```
a=`expr 3 + 4`   # a gets set to 7
a=`expr 3 - 4`   # a gets set to -1
a=`expr 3 * 4`   # a gets set to 12
a=`expr 13 / 5`   # integer division: 2
a=`expr 13 % 5`   # remainder: 3
```

- increment the integer in variable `a`

```
a=`expr $a + 1` # a gets set to a+1
```

let

- let is not the preferred way to do arithmetic in this course
- let arg [arg...]
  - each arg is an arithmetic expression to evaluate
  - see ARITHMETIC EVALUATION of bash man page
  - shell variables are allowed with or without $
  - arithmetic expressions can involve assignment of values to variables, where "=" is the basic assignment operator
- examples:

```
let a=1
echo $a
let a=a+1
echo $a
```

let not preferred

- let seems great, why not use it?
  - because it's not as portable as the alternatives
  - "portable" means you can run your script unmodified on other systems, maybe running `/bin/dash` instead of `/bin/bash`, and it will work just as well
- `expr` is "standard" and works everywhere
- `$((arith))` is what you would use if you don't want to use `expr` (and it's faster than `expr`)
  - x=0
  - x=$((x + 1))
  - x=$(($x + 1))

((expression)) and $((expression))

- What's the difference between `((expression))` and `$((expression))`
- `$((expression))` is an expansion like a variable expansion, where `$((expression))` is replaced by the results of the evaluated `expression`

- `((expression))` is a compound command equivalent to `let expression`

expression examples

- You can put $((expression)) where you can put variable expansion like $myvar:

```
a=0
while [ "$a" -lt 10 ]; do
    echo username: "user$((a+1))"
    a=$((a+1))
done
```

expression examples (cont'd)

- You can use `((expression))` wherever you would use a command:

```
((a=0))       # set a to 0
((a=a+1))     # increment a to 1
a=((a+1))     # syntax error
a="((a+1))"   # set a to literal string "((a+1))"
a="ls"        # set a to literal string "ls"
```

- In these last two, there is what looks like a command, `ls` and `((a+1))`, but no command is run

Scripting Process

- Analyze and understand the problem
- Write pseudocode to document and understand the problem
- Start with a small number of lines in the script, and get that working properly
- Add a small amount to the script, and get it working properly again
- repeat previous step until script is complete
- run final tests

Analyze, understand, pseudocode

- Example from Sobell, A Practical Guide to Linux, Commands, Editors, and Shell Programming, 2nd Ed, Chapter 10
- `lnks` script:
  - Given a filename, find hard links to that file.
  - Search for hard links under a certain directory if one is given; otherwise, search for hard links in the current working directory and its subdirectories.

lnks script analysis

- Generally, for any problem, we need to deal with the following three phases

1. Input
2. Processing
3. Output

Input Phase

- Input is the information/data that a script needs before it can do its job
- What is the information?
  - name of a file
  - name of a directory
- How does our script get its information?
  - the description says "given a filename", and "directory if one is given"
  - this would imply that those items would be command line arguments
  - if the description said "ask for", then the script would prompt for the information after its running

Input Phase (cont'd)

- What's the data that will be processed?
- What work will our script do?
  - the script will look for hard links in the file system
  - therefore, the data is the contents of the file system
  - our script will look for hard links in the file system
  - our script will "read in" this data by running commands that access the file system (`ls`, `find`, etc)

Input Phase (cont'd)

- OK, we now understand the problem
  - take a file name and an optional directory location as arguments on the command line, and use Linux commands to find hard links at that the specified directory (or the current directory if none was specified) in the file system
- Simple, let's just tell the computer to do that!

The game of "what if?"

- Our scripts are like little robots that will do exactly what we tell them to do.
- Imagine you move into a new neighborhood and you tell your domestic robot to go to the corner store and bring back a turkey sandwich and a coke
- The robot comes back with a pound of butter and a can of crab juice
- You aren't happy, but whose fault is it?
  - the store had ham, but no turkey, and pepsi, but no coke
  - again, whose fault is it that you aren't happy?

What if?

- If they don't have coke, then get pepsi, and if they don't have pepsi, then nothing, etc, etc
- When we write our scripts, we end up happiest when we specify absolutely every single detail of every possible circumstance
- We need to make a conscious effort to predict all possible ways things can go wrong
- It's a common script beginner's mistake to think only about everything going right
- The major work is dealing properly with what goes wrong

What ifs for lnks script

- What if no directory name is given on command line?
  - use the current directory (hey, this is easy!)
- What if no file name is given on the command line?
  - print an error message (but then what?)
- What if the given file name is a directory and not a regular file?
  - print an error message and exit
- What if too many names are given?
- What if no hard links are found?
- What if there are no hard links to the file?
- What if the directory is unreadable? Doesn't exist?
- The more "what ifs" our script can handle, the better...

Test Plans

- Speaking of "what if"s...
- A Test Plan is a document that describes a process for verifying that the script does the right thing every time no matter what
- In simple cases it can be a list of tests, which would have this form:
  - test name
  - short description of what's being tested
  - command (and/or instructions) for running the test
  - expected result and/or output from running the test
  - the actual result (filled in when a test plan is executed)

Test Plan's (cont'd)

- As we have seen, the process of understanding the problem and generating a test plan both involve "what if" analysis, and therefore can proceed at the same time
- The plan for testing the final result begins to form early (maybe even before the problem is fully understood!)

Back to lnks script

- Start small and get that working, and then add to it
- Let's start with just the "input" phase:

```
#!/bin/sh -u
PATH=/bin:/usr/bin ; export PATH
umask 022

# identify links to a file
# Usage: $0 file [directory]

if [ $# -eq 0 -o $# -gt 2 ]; then
    echo 1>&2 "Usage $0 file [directory]"
    exit 1
fi
```

- Continue on real Linux system....

Processing

- After finishing the Input stage, we have our file name and directory under which to search
- How do we find the hard links?
- Think about hard links…
  - If we know the inode of the file, then all hard links to the same file will have the same inode
  - We can get the inode of the file using `ls`
  - We can look for files with the same inode using `find`

Processing (cont'd)

- Getting the inode number into a variable named `inode` so we can refer to it as `$inode`
- Consider the output of `ls -i`
- Assuming the filename is in `$file`, Each of the following will do the trick
- remember backticks, `command` is command substitution, same as $(command)
  - `inode=`ls -i "$file" | awk '{print $1}'``

  OR
  - `set - `ls -i "$file"``
  - `inode=$1`

  OR
  - `inode=$(ls -i "$file" | cut -d' ' -f1)`

Processing (cont'd)

- Now that we have the inode, how do we find other files with that inode?
- This literally has the `find` command written all over it! (`man find`)

```
find "$directory" -inum "$inode" -print
```

Output

- How does our script deliver its results?
- Should the names be put in a special file somewhere?
- It depends.
- In keeping with the Unix filter/pipeline philosophy, lets have our script print the names on the standard output
- Hey, `find` is already printing the names on its standard output
- Recall that the standard output of the script is the same as the standard output of the commands inside the script

Output (cont'd)

- Notice that errors and other messages should be printed on standard error
- The "goods" (the actual filenames we were looking for) are printed on standard output
- That's how we like it, because now we can redirect standard output to a file, or run it through grep or any other utility, and we won't have those messages mixed in
- All of those messages will go to standard error, which we can also redirect elsewhere if we want

Indenting

- Indenting: look at how the if statements are indented:

```
if list; then
    ls -l # indented 4 spaces
fi
```

- The same would be true of other control statements:

```
while list; do
    #indent 4 spaces
done
```

Variables (review)

- Variables in our scripts have lower case names
- Environment variables are indicated by their UPPER CASE names: SHELL, VISUAL, etc
- It's usually best to put variable expansions inside double quotes, to protect any special characters that might be inside the variable:

echo "$somevar"

  - if somevar contained the * character, the double quotes stop the shell from globbing it

Variables (cont'd)

- The following all mean different things:
  - run the `myvar` command with two arguments, `=` and `value`:

```
myvar = value
```

  - set the `myvar` variable to have value `""`, then run the `value` command with that variable setting in effect

```
myvar= value
```

  - run the `myvar` command with one argument, namely `=value`:

```
myvar =value
```

  - set the variable `myvar` to have value `value`

```
myvar=value
```

Debugging shell scripts

- -v option for bash/sh
  - sh -v myscript
  - shell will print each line as its read
  - loop statements are printed once

- -x option for bash/sh
  - sh -x myscript
  - shell will display `$PS4` prompt and the expanded command before executing it
  - each loop iteration is shown individually

testing Boundary Conditions

- test plan creation involves identifying the "Boundary Conditions"
- The "typical case" or "normal case" is a necessary test, and all such cases are considered equivalent (test one and you've tested them all)
- Boundary cases are all interesting:
  - present or missing
  - too small, just big enough, typical, almost too big, too big
  - MINIMUM, ... -1,0,1,... MAXIMUM

Boundary Conditions example

- Suppose you're looking for 8.3 filenames, where the "main part" is up to 8 characters, and the extension is exactly 3 characters
  - main part of filename
    - boundaries
      - 0, 1, 8, 9 characters
    - typical case (only one needed)
      - 2,3,4,5,6,7
  - extension
    - boundaries
      - 0,3,4
      - typical case (same as the boundary) 3