COMP09024 Unix System Administration

Lecture 6: The Shell and Shell Scripting

Duncan Thomson/Hector Marco

UWS

Trimester 1 2020/21

Outline

6.1 Shells
- What is the Shell?
- Alternative Shells

6.2 The Shell as a CLI
- Command Line Editing
- Globbing
- File and Command Completion
- Aliases
- Job Control and Redirection

6.3 Variables and the Environment
- Variables
- Setting and Using Variables
- Special Variables
- Quoting
- Environment Variables

6.4 Shell Scripts
- What is a Shell Script
- Creating a Shell Script
- Parameters and Exit Values
- Shell Configuration

6.5 Control Structures
- Selection and `if`
- The `case` Statement
- Iteration
- The `for` Statement
- Functions

6.1 Shells

What is the Shell?

- The *shell* is the user's primary means of interacting with the system and its programs
- It provides a command line from which processes can be initiated and controlled
  - Job control (foreground, background)
  - I/O redirection (file redirection and pipes)
- It provides mechanisms to ease working with the system:
  - Customisation (prompts, aliases, search paths)
  - Shortcuts (command and filename completion, command history and editing)
  - Wildcard (or globbing) characters to select multiple files
- It provides the ability to write scripts (programs) using variables and control structures and other Unix commands

Alternative Shells

- The traditional Unix shell — found on all Unix systems — is `sh`, known as the Bourne shell
- Most systems include other shells with additional features
- `bash` — Bourne again shell is a `sh`-compatible shell commonly used on GNU and Linux systems; includes command completion, history and so on
- `csh` — the C shell was developed to provide a shell programming language with a more C-like syntax
- `ksh` — the Korn shell is similar to `sh` but with some features from the C shell
- `tcsh` — TENEX C shell adds command completion to `csh`
- `zsh` — Z shell (Z rhymes with C) — is another advanced shell with features of `bash` and `ksh`

6.2 The Shell as a CLI

Command Line Editing

- `bash` provides a command history
- You can scroll through the command history with the up and down arrows, or Ctrl-P (for previous) and Ctrl-N (for next)
- On a command line, the left and right arrows move back and forward one character at a time (also Ctrl-B and Ctrl-F)
- Move to start or end of line with Ctrl-A or Ctrl-E
- Move forward or backward one word with ESC F or ESC B
- Character deletion can be performed in the backward direction (with BS) or in the forward direction (DEL)
- There are many more keystrokes available for functions from capitalising, word deletion and so on — see `man bash` for details

Wildcards and Globbing

- The shell provides some wildcard characters in order to match a range if filenames — known in Unix as *globbing*
- There are **not** the same as the wildcard characters used in regular expressions (sorry!)
- The main globbing characters are:
  - `?` matches any single character
  - `*` matches any character sequence
  - `[charlist]` matches any character from *charlist*
- The filenames matching the pattern are expanded by the shell into a list of multiple filenames
- Hidden files starting with `.` are not normally matched
- Examples:
  - `*.mp?` — matches files ending with `.mpa`, `.mp3`, `.mpX`, `...`
  - `file[0-9]` — matches `file0`, `file1`, `...file9`

File and Command Completion

- `bash` supports automatic filename and command completion
- When part of a command is typed, and the user presses the TAB key:
    - If there is only one possible command, it is completed
    - If there are multiple possible completions, it is completed as far as possible — pressing TAB again will list all possible completions
- This works with filenames too
- This has two advantages:
    - Increases speed of entering long commands and filenames
    - Decreases likelihood of errors

Aliases

- The shell supports the use of command *aliases*
- This allows commonly used commands to be shortened to allow for quicker entry
- Aliases are created using the `alias` command eg:

```
# print double sided with staples
alias lpr2s=lpr -o sides=two-sided-long-edge \
                -o StapleWhen=EndOfSet
```

- Aliases can be removed using the `unalias` command

Job Control and Redirection

- We have already seen the shell's support for controlling jobs and their input and output streams
- For job control these include `&`, `fg`, `bg` and `jobs` (and the `%` operator to specify job numbers)
- For redirection these include:
  - `<`, `>` and `2>` for redirection of standard streams
  - « for here-files
  - `|` for pipelining commands
- It is also the shell which interprets command chaining operators such as `;`, `&&` and `||`

6.3 Variables and the Environment

Variables

- Variables allow the temporary storage of information
- This information can be used for several purposes:
  - Controlling how the shell itself works
  - Controlling how processes initiated by the shell work
  - For storing data in shell scripts (programs)
  - For storing parameters to shell scripts or functions
- Every variable has a name and value (or contents)
- The shell command `set` by default prints a list of all defined variables (and functions)

Setting and Using Variables

- Variables are assigned a value with `name=value`
- The `read` function can also be used to take the value of a variable from user input
- To use the value of a variable, a dollar sign `$` is normally placed in front of it: `$name`
- The name make be surrounded by braces (curly brackets) to avoid confusing with surrounding text: `${name}`
- Examples:

```
# hash signs are used for comments in the shell
today=Monday
read name
echo Hello $name
echo Today is ${today}, how are you?
```

Special Variables

- `$?` we have seen already — it contains the exit status of the last command executed
- `$0` contains the name of the command currently being executed
- `$1` – `$9` are positional parameters (see later)
- `$$` contains the PID of this shell
- `$!` contains the PID of the last backgrounded command
- There are also some special ways of using shell variables which can check to see they are defined first
- One example is `${`*`name-value`*`}` — this has the value of the variable *`name`* if it is defined, and the value *`value`* if not

Quoting and Variables

- There are several ways of quoting in the shell
- Single (forward) quotes (’ ’) protect all special characters from being interpreted by the shell (including variables)
- Double quotes (" ") protect all except $, !, \ and `
- Backslash (\) can escape single special characters
- Single (back) quotes (` `) or $(*command*) replace the quoted command by its standard output

```
echo $myVar                whoami
echo "$myVar"              whoami
echo \$myVar               $myVar
echo ’$myVar’              $myVar
echo `$myVar`              alice
echo $($myVar)             alice
```

Environment Variables

- The environment is the name given to the set of variables whose values are passed on into any initiated processes
- `set` prints a full lists of variables known to the shell
- `env` prints a list of only environment variables (those which will be passed on to processes)
- A variable can be made into an environment variable by using the `export` command:

```
# VAR1 is not an environment variable
VAR1=anne
VAR2=bruce              # VAR2 also isn’t... yet
export VAR2             # now VAR2 is
export VAR3=carol       # and so is VAR3
```

Shell Environment Variables

- `USER`, `USERNAME` and `LOGNAME` are used to indicate the name of the user
- `PATH` specifies the *search path* — which directories will be searched for commands, usually a list of directory names separated by colons
- `HOME` gives the home directory of the user
- `PWD` holds the present working directory of the user
- `TERM` specifies the terminal type for text-mode applications
- `DISPLAY` specifies the display ID for windowed applications
- `SHELL` holds the name of the shell
- `LANG` holds a definition of the language being used

6.4 Shell Scripts

What is a Shell Script

- It is possible to write programs using the shell as a programming language
- The shell provides a number of control strutures to help with this
- Variables and commands can be used as they would normally be within the shell
- Such programs are known as *shell scripts*
- Shell scripts are widely used for system administration
    - Batch jobs (such as for `cron`)
    - Start / stop scripts for various services
    - Instead of aliases for more complex tasks

Creating a Shell Script

In order to be used as a script, a file must follow three rules

1. The first line must begin with the characters `#!`
   - These are known as the hash-bang or ‘shebang’
   - These indicate that the file is a script file
2. The remaining characters on the first line must be the full path to the interpreter being used
   - For a shell script this would be `/bin/sh` or `/bin/bash`
   - But other interpreters can be used (Python, Perl, etc)
3. The file must be executable
   - Use `chmod` to do this, and `ls -l` to check

An example shell script (if executable!):

```
#!/bin/bash
echo "Hello world"
```

Parameters to Shell Scripts

- Shell scripts can accept command line arguments and flags
- These are available in the shell script as special variables
- `$0` represents the name of the shell script
- `$1` is the first parameters, `$2` is the second, and so on
- `$*` is a full list of all parameters
- The `shift` builtin command removes the first parameter (`$1`), and moves all the rest to the left one position

```
#!/bin/bash
echo $0 $1 $2
shift
echo $0 $1 $2
```

```
$ ./script.sh 1 2 3
./script.sh 1 2
./script.sh 2 3
```

Exit Values

- Each shell script has an exit value
- Exit values can be checked with the special variable `$?`
- A shell script can exit with a specific value using the command `exit` with a number afterwards
- These exit values can affect the outcome of control structures (which we'll look at shortly)
- Functions can also return exit values using the `return` command
- Usually a good idea to have an explicit `exit 0` at the end of a shell script to signify successful completion

```
#!/bin/bash
exit 0
```

Shell Configuration

- When `bash` starts, a number of other ‘script’ files are sourced (run) in order to set up the operating environment:
  - For login shells, `/etc/profile`, `~/.bash_profile`, `~/.bash_login` and `~/.profile`
  - For non-login interactive shells, `~.bashrc` (which usually reads `/etc/bashrc`, and is also usually read by `~/.bash_profile`)
- These scripts can do various things:
  - Set up variables, eg `PATH`, `PS1`, `PS2`
  - Set up aliases
  - Set up shell options (eg with `shopt`)
- Such scripts can be run at other times using `source` or `.`

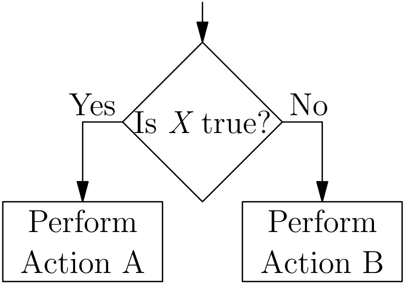
Example to show directory and time in the prompt:

```
user@debian:~$ PS1="\w(\t)\$ "
~(16:33:43)$
```

6.5 Control Structures

What is Selection?

- Selection allows shell scripts to take different actions based on:
  - Exit status of commands
  - Expressions (eg variable values)
- Implement *branching*



The `if` Statement

- The `if` statement begins by executing a command (immediately after `if`) — what happens next depends on the exit status of this command
- There then follows a `then` command
- Commands following `then` are executed if the exit status is `0` (normally indicating ‘success’)
- There may then be a final `else` command — commands following this are executed if the exit status is non-zero
- The overall `if` statement ends with a `fi` statement
- (There may be a set of `elif...then` sections before the final `else`)

An Example if Statement

- The following searches for the string root in /etc/passwd and prints an appropriate message
- Note that many commands can appear in each of the then and else (and elif) sections
- First command in each section need not be on a new line
- Indenting is used for clarity (but not required)

```
if grep -sq root /etc/passwd
then echo "root user present"
else
  echo "No root user"
  echo "perhaps create one?"
fi
```

The `test` Command

- The `test` command command is often used with `if`
- `test` can check for various conditions (on strings, numbers, files, or logical expressions)
- A *small* number of `test` capabilities are outlined below
- `test` *`condition`* can also be written `[` *`condition`* `]`

Test	Meaning
str1 = *str2*	Are the strings equal?
str1 != *str2*	Are the strings not equal?
-n *str*	Does the string have non-zero length?
int1 -eq *int1*	Are the integers equal?
int1 -gt *int1*	Is the first number greater than the second?
-d *file*	Is the file a directory?
-w *file*	Is the file writeable?
expr1 -a *expr2*	Are both expressions true (logical and)

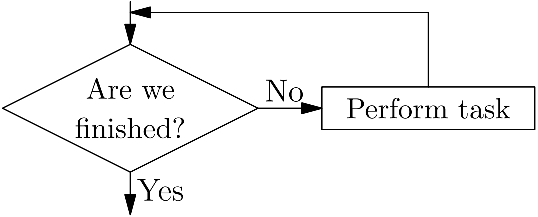
The case Statement

- The case statement allows checking a string expression against patterns (in parentheses), and executing specific commands (ending with ;;) in each case
- After the expression the word in is used
- The patterns may contain globbing wildcard characters
- The overall statement ends with esac
- An example illustrating the syntax is:

```
read userInput
case $userInput in
  (a) echo "You typed a";;
  (b*) echo "You started with a b" ;;
  (*) echo "You typed somthing else ;;
esac
```

What is Iteration?

- Iteration allows repetition of a set of commands, perhaps:
  - Until (or while) a condition is true
  - A set number of times
  - For a set of values



while and until

- while keyword introduces a list of commands
- The exit status of the last command in this list controls whether the loop is executed; successful status (0) means the loop executes
- do keyword introduces statements forming the loop body
- done keyword ends the loop body
- There is also an until statement which will repeat until the exit status is successful

```
sum=0      # add numbers until -1 input
while read num; [ $num -ne -1 ]
do
   sum=$(expr $sum + $num)
done
echo $sum
```

The for Statement

- The `for` statement has two forms
- It can iterate through values from a list:
  - The syntax is: `for` *`varName`* `in` *`item1 item2`* `...`
  - The variable *`varName`* takes each of the values in the list in turn, and the loop body is executed for each
- It can have an initialisation, a check, and an incrementer arithmetic expressions (like C, Java or JavaScript):
  - The syntax is `for ((` *`expr1`* `;` *`expr2`* `;` *`expr3`* `))`
  - *`expr1`* is first evaluated
  - Then *`expr2`* is evaluated — if non-zero, the loop body runs
  - Then *`expr3`* is evaluated after each run of the loop
  - After this *`expr2`* is evaluated again, and so on
- The commands in the loop body are contained between a `do` and `done` keyword (like for `while`)

Examples of for

```
for filename in `ls`
do
  if [ -f $filename ]
  then touch $filename
  fi
done
```

```
for (( c=1 ; c<=3 ; c++ ))
do
  echo $c potato
done
echo $c
```

What are Functions?

- Functions allow a divide and conquer approach to writing a shell program
- They also allow writing commonly used functionality once, which can then be reused when required
- Shell functions take a number of parameters and return an exit value (just like shell scripts)
    - Special variables `$#`, `$1`, `$2`, . . . reflect the function parameters (or arguments)
    - The `return` statement is used to return an exit status from the function
- Functions are not separate processes — they don't have their own `stdin` and `stdout`
- Unlike in some programming languages, functions can only return a numeric value (exit status)
- Any variables created have global scope by default; `local` can give them local scope only

Defining and Using Functions

- Functions are defined with the syntax:

```
function  funcName () {
   commands;  # this is the function body
}
```

- Function body is executed each time the function is called
- Functions are called as if they were shell commands — parameters follow the function name on the command line

```
function rename() {    # rename files
  for fileName in $1 ; do
    local baseName=$(basename ${fileName} $1)
    mv $fileName ${baseName}$2
  done
}

# now call the function
rename .JPG .jpg
```

Summary

- The role of the shell and different shells `sh`, `bash`, `ksh` ...
- Working on the command line: editing
- Filename matching with globbing characters: `*`, `?` and `[]`
- Automatic file and command completion with TAB
- Variables: setting and using; special variables; `$`
- Quoting with `'`, `"` and `` ` ``; `\` and `$( )`
- Environment variables and `export`
- Shell scripts: `#!/bin/bash` and executable mode
- Parameters and exit values of shell scripts: `$?` and `exit`
- Selection: `if...then...fi` and `case...esac`
- Iteration: `while` and `until` (with `do...done`)
- Iteration with `for...in` and `for (( ))` (two forms)
- Writing functions in the shell: `function`