Week 4.1

Shells and scripts

School of Information Technology and Electrical Engineering
The University of Queensland

This Week

Lectures
- Unix shell
- Shell scripts

Pracs
- Assignment 1 (Due this Friday)

Outline (shells)

UNIX Shell + Shell Scripts

- Based on Glass & Ables – mostly chapter 4 but also 5 and 8

What is a Shell?

Interface between the user and the operating system

Provides

- Input/output redirection, pipes
- Wildcards
- Job/process management (e.g. background jobs)
- Command history
- Command line editing
- Some built in commands/functions
- Scripting functionality
- … plus more

What Shells are There?

Many

- Thompson Shell (sh) – the original UNIX shell
- Bourne Shell (sh) – from UNIX v7 (1977)
- Bourne-Again shell (bash) – superset of sh
  - What you're probably using on moss
- Korn Shell (ksh)
- Z shell (zsh)
- C shell (csh)
- TENEX C shell (tcsh)
- Scheme shell (scsh)
- …

What Does a Shell do?

- When invoked (e.g. by login process or manually)
  1. Reads startup file(s) and initialises
  2. Displays a prompt and waits for a user command
  3. If user indicates end-of-input (often Ctrl-D), shell terminates, otherwise executes user command and returns to step 2
- Shell scripts are similar, except commands come from a text file

Executable Programs vs Built-in Commands

Commands can be either

- Shell **built-in commands** (i.e. shell does something, no external process started), or
- Separate executables (i.e. shell starts up an external process)

All shells have this concept but the built-in commands available in each shell differ

Command Examples

- ls, sort, vim, pico, gcc, indent, which (external)
- cd, alias, type (builtin)
- echo, pwd, printf (either)

Bash built-in command `type` will tell you which type of command

Bash built-in command `help` will list the built-in commands (and can be used to get help on them)

Built-ins executed in preference to external commands unless

- disable built-in, or
- give full path to external command

Variables

Shell has two kinds of variables

- Local (or shell) variables
  - Used only by the shell
- Environment variables
  - Values passed to child processes

Variables are strings

Values accessed using `$varname` notation

- e.g. echo `$HOME $USER $SHELL`

Predefined Environment Variables

Include:

HOME – full pathname of home directory

PATH – colon separated list of pathnames to search for commands

USER – your username

SHELL – full pathname of your login shell

Defining a Variable

`variableName=value`

No spaces around the equals sign

Can change a shell variable to an environment variable using built-in command `export` , e.g.

```
      courseCode=CSSE2310
      export courseCode
```

Example to be given in class

No need to declare variables before use

Different shells use different syntaxes for defining variables (above is for Bourne, Bash)

Metacharacters

Some characters mean special things to the shell

When you type a command, these **metacharacters** are processed before the command is executed

If you don’t want the special treatment, use backslash before the character or quote appropriately (discussed later)

Metacharacters (cont.)

Wildcards (for filename matching)

- `*` = zero or more characters
- ? = any single character

Comment

- # = start of comment (goes till end of line)

Running commands

- & = run command in background
- `;` = used to separate commands
- `` `command` `` = substitute result of running command

Variable substitution

- `$varname` = substitute value of variable

Metacharacters (cont.)

Subshell

- `( … commands …)` = execute commands in a sub-shell

Conditional sequences

- || = execute command if previous command failed
- && = execute command if previous one succeeded

Command “succeeds” if returns zero exit status; “fails” if returns non-zero

Metacharacters (cont.)

Redirection & Pipes

- | = pipe, output of one program sent to the input of the next
- > = send standard output to a file
- < = read standard input from a file
- >> = append standard output to a file

There are other metacharacters also (and can vary by shell)

Examples

To be presented in class

- Wildcards (* ?)
- Redirection (< > >>)
  - /dev/null
- Pipes (|)
- Command sequences (; || &&)
- Subshell (())
- Command substitution (`)
- Background processing (&)

Shell Scripts

Shell script = series of commands in a regular text file

Can be made executable and executed like a regular command:

`chmod +x` *`script-file-name`*

`./script-file-name` *(if it's in current directory)*

How does the system know which shell to use? – depends on first line of the shell script

- `#` - use the current shell
- `#!pathName` – use the shell with the specified path
  - e.g. `#!/opt/local/bin/bash`
- anything else, use the Bourne Shell (`/bin/sh`)

Built-in Shell Variables

All shells support:

$$ - process ID of the shell

$0 – name of the shell script (if applicable)

$1 … **$9** – command line arguments (if applicable)

$* – all the command line arguments

Bourne shell/Bash support (amongst others):

$# - number of command line arguments

- excludes command name

$? – exit status of last command

$! – process ID of last background command

Quoting

Sometimes want to stop the shell replacing metacharacters

Single quotes – inhibit wildcard replacement, variable substitution, command substitution

Double quotes – inhibit wildcard replacement only

Can also use backslashes

Examples in class...

Startup Files

Files read at startup vary
- by shell (different shells use different files)
- by mode
  - login shell
  - interactive shell
  - non-interactive (shell script)

Files contain commands which are executed (or **sourced**)
- Not a separate process – commands are executed within the current shell – just like you'd typed them in
- To source a file within Bash, use either:
  - . filename
  - source file

Bash Startup

Login Shell

- `/etc/profile` (system wide settings)
- `~/.bash_profile` OR `~/.bash_login` OR `~/.profile`
- (`~/.bash_logout` executed on exit)

Interactive Shell

- `~/.bashrc`

Shell Script

- Looks for file named in `BASH_ENV` environment variable

Other features

- Arithmetic:
  - Use bash's let command or `expr` (see man pages)
- Conditional expressions:
  - Use bash's [ ] or `test` command.

Control Structures

```
for name [ in word... ]
do
      commands...
done
```

Means this part is optional

variable name is assigned each of the the words in turn

If words omitted, uses script arguments $1 $2 …

Examples…

CSSE 2310 7231

Control Structures

```
for name [ in word... ]
do
      commands...
done
```

Means this part is optional

variable name is assigned each of the the words in turn

If words omitted, uses script arguments $1 $2 …

Examples…

Control Structures (cont)

```
if commands1…
then
    commands2…
elif commands3…
then
    commands4…
else
    commands5…
fi
```

`elif` and **`else`** branches are optional

- Can have multiple **`elif`** branches

If last command in *commands1…* succeeds (exit status 0) then *commands2…* executed, etc

Example (commands are often test expressions)

Control Structures (cont)

```
while commands1…
do
    commands2…
done
```

commands1… commands executed and if last command has exit status 0, then *commands2…* executed – repeat until *commands1…* return non-zero exit status

Example …

Other Control Structures

case…

until … do… done

trap