TNE30019/TNE80014 – Unix for Telecommunications

The Unix Shell and CLI

Dr. Jason But

Swinburne University

Outline

What is the Shell?

Shell – definition

Command-line interpreter providing user interface for Unix OS

Choice of shell is configurable (user account details)

`sh`	Bourne shell
`bash`	Bourne-again shell
`csh`	C shell
`tcsh`	TENEX C shell

The Shell

- Process like everything else
- Interpreted programming language
- Commands/instructions are received from input pipe – `stdin`
- Output is delivered to output pipe – `stdout`
- Error output is delivered to error pipe – `stderr`
- Provides environment variables to store values, e.g. `env`
- Provides basic prompt – configurable using shell variables
- Provides built-in commands to perform basic tasks
  - `cd`
  - `pwd`
  - `exit`
  - ...

The Shell Process

- As process the shell has **process ID** (positive integer number)

Launching Processes

- When you run program, shell launches new process with new process ID
- New process has shell as its parent process
  - Parent process ID = shell process ID
- When shell terminates, all child processes are also terminated
- Often child processes are run in **foreground**
  - Shell process blocks until child process finishes
- You can launch shell from within shell
  - New shell replaces current shell
  - Current shell blocks until new shell terminates

Background Processes

- Processes can be run in **background** with "&" after command
- Both new and parent process run concurrently
- If parent process is shell – can continue issuing commands

Managing Processes

- Ctrl-C
  - Kills currently running process
- Ctrl-Z
  - Suspends currently running process
- `jobs`
  - Lists currently running and suspended *child* processes
- `fg`
  - Resumes specified/last process and moves it to foreground
  - Shell process blocks until foregrounded process finishes
- `bg`
  - Resumes specified job and moves it to background

Redirecting Output

- Like all other devices `stdin`, `stdout` and `stderr` live in `/dev`
- Always points to current process/virtual terminal
- Input/output can be redirected to file on filesystem

`myprog > ./myfile`

Run `myprog` and redirect output from `stdout` to file `./myfile`

`myprog >> ./myfile`

Redirect output from `stdout`, **appending** it to `./myfile`

`myprog < ./commands > ./myfile`

Run program taking input from `./commands` as if it were typed at keyboard, redirect output from `stdout` to `./myfile`

`myprog > ./myfile 2>&1`

Redirect output from `stdout` and `stderr` to `./myfile`

Piping Output

- Unix "invented" concept of **pipes**
- Also available in newer Windows shells
- Many programs accept input from character device (`stdin`) and direct output to character device (`stdout`)
- Should be able to direct output from one process directly to input of another
  - This is called piping
  - `stdout` of **process 1** is piped (*instead of displayed*) to `stdin` of **process 2** (*instead of reading from the keyboard*)

Piping Output

`myprog1 | myprog2 | myprog3`

- Shell launches three separate processes concurrently
- stdout output of `myprog1` is treated as input by `myprog2`
- stdout output of `myprog2` is treated as input by `myprog3`
- `myprog3` outputs to `stdout`

Example: `cat ./myfile | less`

- Show file page by page

Example: `find / | grep foo | wc -l`

- Runs `find` to display every single file on filesystem
- Output is piped to `grep`, which outputs only lines with "**foo**"
- Output piped to `wc`, which outputs number of lines input
- Result is count of files with word "**foo**" in filename

Piping Output

- Pipes can be extremely powerful
- Generic tools can be built and re-assembled as pipeline to perform complicated tasks
- True modular programming
- Many good standard tools provided with Unix
  - grep, wc, sed, find, gzip, pstopdf, wget
  - and many more...

Grouping Processes

`myprog1 ; myprog2`

Execute `myprog1`, when it finishes execute `myprog2`

`myprog1 && myprog2`

Execute `myprog1`, if it terminates normally (no error code) then execute `myprog2`

`myprog1 || myprog2`

Execute `myprog1`, if it terminates abnormally (error code) then execute `myprog2`

- Last two are handy when writing shell programs
- Do not confuse last with piping

Shell Scripts

- Shell is much more than interactive console for users

Shell is programming language interpreter

- Variables
- Functions with parameter passing
- Loops
- Conditionals

- Available on command line, but difficult to use
- Power of shell shows when program is pre-written

Writing and running Shell Scripts

- ASCII (text) – use your favourite text editor
- `/bin/sh script_name` (Bourne shell script)
- `/usr/local/bin/bash script_name` (BASH shell script)

Shell Scripts

Making Shell Scripts executable

- Executable bit(s) must be set
- First line must start with ASCII characters "#!"
- Rest of first line defines interpreter used to execute script
- Process launcher will launch specified interpreter with script as argument

Example: myprog

```
#!/bin/sh
find ${1} | grep "${2}" | wc -l
```

- Run with `./myprog <dir> <text>`
- Equivalent to running `/bin/sh myprog <dir> <text>`

Running Scripts In Shell

- Not limited to standard shells
- Can be used with any interpreter

PERL Programming Language

```
#!/usr/bin/perl
```

PYTHON Programming Language

```
#!/usr/local/bin/python
```

PHP Programming Language

```
#!/usr/local/bin/php
```

- Normally Web scripting but can be used for shell scripting

Shell Script Variables

Command line arguments

- Accessible with $<number>
- $0 is the script name itself
- $1 is the first command line argument
- ...

Variables

- Syntax depends on shell (here Bourne shell)
- Assignment: `WORD=$1`
- Use: `$WORD` or `${WORD}`

Example: myprog

```
#!/bin/sh
WORD=$1
find / | grep "$WORD" | wc -l
```


Fun With Quotes In Shell Scripts

Single quotes ' '

Preserves literal value of characters in quotes

```
$ echo '$SHELL'
$SHELL
```

Double quotes " "

Preserves literal value of characters in quotes, except $, \, \, !

```
$ echo "$SHELL"
/usr/local/bin/bash
```

Back quotes/ticks \\

Everything in back quotes is executed before main command

```
$ echo `basename $SHELL`
bash
```

Example Shell Script

```
#!/bin/sh
# List all .html files and copy first lines
# of each file into one line of text file

FILE_HEADS=$1

FILE_LIST="`ls *.html`"
echo FILE_LIST: ${FILE_LIST}

RESULT=""
for FILE in ${FILE_LIST} ; do
   FIRST_LINES=`head -2 ${FILE}`
   RESULT="${RESULT}${FIRST_LINES}\n"
done

echo -e ${RESULT} > ${FILE_HEADS}
```