Module 2

Basic Shell

Objectives

On completion of this module, you should be able to understand and use the Linux shell to create and combine tools.

Topics covered include:

- An overview of the command line
- The software tools model
- File names and types
- Shell programming
- Command scripts
- Job control
- I/O — pipes and redirection

2.1 Introduction

- The standard command line interpreter under Linux is `bash` (`/bin/bash` or `/bin/sh`)
- An enhanced version of the classic Bourne shell
- Shares most features of other shells (`C`, `Korn`, etc) and has some more advanced features
  - 'Plumbing' — transparent redirection and pipes
  - Background processes
  - Process suspension, resumption, termination
  - Filename completion and wildcard generation
  - History

2.2 Getting around the command line

- You can use the cursor keys to move around and edit the current line*
- By default, `bash` uses `emacs`-like keystrokes for navigation and editing. Here are 4 examples:

Keystroke	Action
`^a`	Move to the beginning of the line
`^e`	Move to the end of the line
`^k`	Delete to the end of the line
`^w`	Delete the previous 'word'

- To choose `emacs` or `vi`-like keystrokes:

```
$ set -o emacs
$ set -o vi
```

- `bash` man page gives details of all keystrokes

*This may not work on badly-configured systems

2.3 History

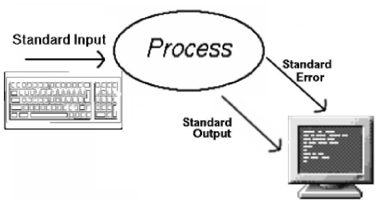
- Bash remembers used commands (in a 'history')
- Old commands are retrievable in different ways
- Repeat the previous command by typing `!!`
- Execute the *n*th previous command by typing `!-n`
- Typing `!string` repeats the last command beginning with `string`
- To view your history command by command, use the `up` and `down` cursor keys
- View your history at any time by typing `history`
- History is a *very* useful feature, if used well
  - Incrementally searchable using `CTRL-R`

2.4 Plumbing

- Processes typically start with three files open:

Name	Descriptor
Standard input	0
Standard output	1
Standard error	2

- Later we see how to refer to their *file descriptors*
- These are normally connected to the keyboard and your command-line terminal



2.5 Plumbing (continued)

- Data can be redirected by the shell
  - Transparently to the process concerned
  - Any or all streams can be redirected
  - You can redirect to/from a file or to/from another process
- Redirection to a process is known as ‘piping’

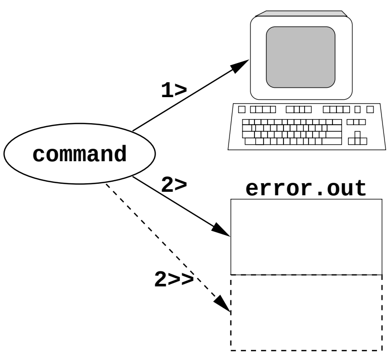
2.6 Output Redirection

- Redirection of output is done using ’>’
- For example:

  `$ command > output`

  Creates the file `output` (or overwrites it if it already exists) and places the standard output from `command` into it
- We can append to a file rather than overwriting it by using >>
- > and >> are actually shorthands for `1>` and `1>>`
- Error output can be redirected using `2>` or `2>>`

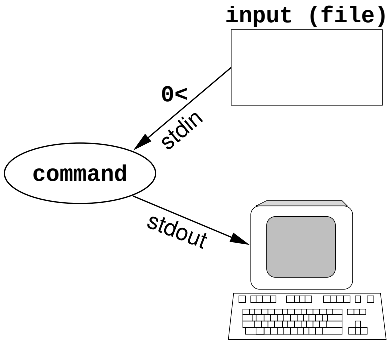
  `$ command 2> error.out`



2.7 Input Redirection

- < redirects standard input from a file, e.g.

  `$ command < input`
- `command` will now take the contents of the file `input` as its input
- This could also be written as `0<`
- Consistent with > and `1>`



2.8 Combining Redirection

- Redirect more than one descriptor by giving more than one redirection, e.g.

```
$ command 1> output 2> error
```

- Group redirections using the `>&` operator, e.g.

```
$ command 1> output 2>&1
```

  - Output to the file called `output` (`> output`)
  - Send errors to the same place as the standard output (`2>&1`)
- The order of these is *very* important
- The redirections are evaluated left-to-right, e.g. the following differs from the previous example

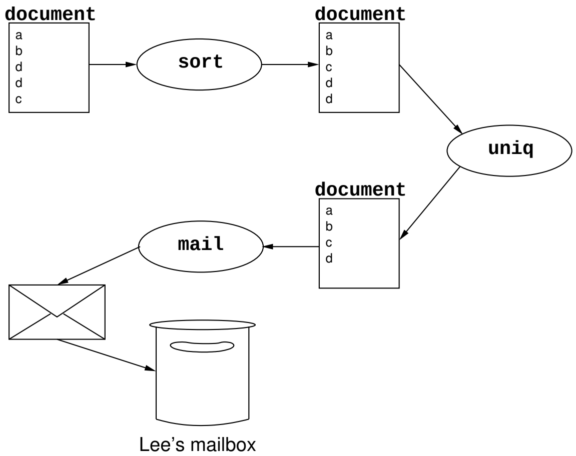
```
$ command 2>&1 > output
```

  - It sends error to the normal output and normal output to the file called `output`

2.9 Pipelines

- You can output to another process with '`|`'
  - Known as the *pipe* symbol
- A *pipe* connects the output of one process to the input of another
- The data waiting to be transferred is buffered
- The processes run concurrently
- Linux ensures that the processes keep in step
- For example:

```
$ sort document | uniq | mail lee
```



2.10 Background Processes

- Most commands run to completion before you get your shell prompt back
- A 'background' process continues while you get your prompt back immediately
- To launch a process in the background place `&` at the end of the line, e.g.

```
$ sort /var/log/maillog > output &
```

- Unless you use redirection (plumbing), output and error continue to appear on your terminal
- Input is disconnected, so typing goes to the shell, not to the background process
- If a process needs user input, and can't take it from a file, it is 'stopped'
  - It won't resume until brought to the foreground to receive input
- You should normally start background processes with their output and error redirected to a file, e.g.

```
$ sort big_file > output 2> error_output &
```


2.11 Background Processes (continued)

- Running processes can be put in the background
  - Suspend the process by typing `^Z` in the terminal that the process is running in
  - Put the process in the background using `bg`
- Bring a process back to the foreground using `fg`
- `fg` and `bg` operate on the most recent process by default
  - Change to a process, by job number or name
- `jobs` displays current shell processes:

```
$ jobs
[1]+  Stopped (tty output)    top
$ fg %1
```

2.12 Background Processes and `nohup`

- Sometimes it is necessary to start a process and leave it running when you log out
- If your shell is killed, any background processes will also be lost
- `nohup` gets round this by detaching the process from the terminal
- Always redirect output and error with `nohup`, e.g.

```
$ nohup sort bigfile > out 2>&1 err.out
```

- If you don't redirect them then they will end up in `./nohup.out` and `./nohup.err`

2.13 Command Grouping and Sub-shells

- `bash` can execute multiple commands on a line
- Sequential commands are separated by ';'

```
$ sort data ; mail lee < sorted_data
```

- It's possible to launch a sub-shell to execute a command or group of commands
  - Put commands in parentheses, e.g.

    ```
    $ (command1 ; command2)
    ```

- Can also put a subshell in background, e.g.

```
$ (command1 ; command2) &
```


2.14 Process Management

- `ps` (process status) prints info about a users' processes:

```
PID   TTY STAT TIME COMMAND
22074 p0  S    0:02 Eterm -t trans
22075 p0  S    2:13 emacs -bg black
22081 p0  S    0:00 asclock
22590 p5  R    0:00 ps
```

- `jobs` only prints info about processes belonging to the current shell
- `wait` postpones shell until process is finished
  - Usually given a process id as an argument
  - If no argument is given it waits until all the shell's processes have terminated
- `kill` is used to send *signals* to processes
  - Can terminate background processes
- Some processes use signals to trigger tasks, e.g. log rotation, re-reading config files, etc

2.15 Signals

- `kill` can be given a signal name or number
- There are a variety of signals:

SIGHUP	1	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Interrupt from keyboard
SIGQUIT	3	Quit from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal
SIGUSR1	30 10 16	User-defined signal 1
SIGUSR2	31 12 17	User-defined signal 2

2.16 Signals (continued)

- Unless specified, `kill` sends a SIGTERM which causes most processes to terminate
- If a process is unresponsive, it can be forcibly killed by sending it SIGKILL

```
$ kill -9 1512
1512: Terminated
```

or

```
$ kill -KILL 1512
1512: Terminated
```

- Can only signal your own processes
  - Superuser can signal all

2.17 Background Processes: `top`

- `top` displays the processes running on a machine
- Results can be sorted in various ways
- Options:
  - See `man top` for full details, including command-line options
  - Inside `top` use `h` for help on interactive options
- Typical output:

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM   TIME COMMAND
22594 user      10   0   736  736   556 R      0  8.2  0.5   0:00 top
    1 root       0   0   144   96    76 S      0  0.0  0.0   0:03 init
    2 root       0   0     0    0     0 SW     0  0.0  0.0   0:19 kflushd
    3 root     -12 -12     0    0     0 SW<    0  0.0  0.0   2:42 kswapd
  486 root       5   5  3160 2356   804 S N    0  0.0  1.8   0:03 mysqld
  869 root       0   0    68   12    12 S      0  0.0  0.0   0:00 mingetty
  838 www        0   0 11468 6280   488 S      0  0.0  4.9   4:14 squid
   48 root       0   0   100   80    48 S      0  0.0  0.0   0:01 kerneld
  230 root       0   0   384  372   272 S      0  0.0  0.2   0:12 syslogd
  239 root       0   0   164  120    72 S      0  0.0  0.0   0:00 klogd
  250 daemon     0   0   164  132    88 S      0  0.0  0.1   0:00 atd
  261 root       0   0   192  160   112 S      0  0.0  0.1   0:01 crond
  272 bin        0   0   244  224   168 S      0  0.0  0.1   0:00 portmap
  283 root       0   0   572  296   248 S      0  0.0  0.2   0:17 snmpd
  295 root       1   0   136   88    60 S      0  0.0  0.0   0:00 inetd
  306 root       0   0   516  488   224 S      0  0.0  0.3   0:06 named
  317 root       0   0   124   56    48 S      0  0.0  0.0   0:00 lpd
```

- N.B. `top` is not available on all unices

2.18 Filename Generation

- Some characters are ‘special’ to the shell

Chars	Meaning
`*`	Matches any string, including the null string
`?`	Matches any single character
`[...]`	Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a range. Any character lexically between those two characters, inclusive, is matched. If the first character following the ‘`[`’ is a ‘`!`’ or a ‘`^`’ then any character not enclosed is matched. A ‘–’ or ‘`]`’ may be matched by including it as the first or last character in the set.

Table 2.1: Special characters under `bash`

- Special characters can be used to match filenames, e.g. to show files beginning with `f`

```
$ echo f*
```

- To show files starting with ‘`f`’, followed by a vowel:

```
$ echo f[aeiou]*
```


2.19 Quoting Mechanisms

- Sometimes it’s necessary to ignore a character’s special meaning
- Use a backslash (\) to quote a special character, e.g. to list a file called `f*`

```
$ ls f
```

- To quote a longer string, enclose it in quotes :

'	disable all interpretation
"	disable filename generation and blank space interpretation

2.20 Shell built-in commands

- Some commands must be built in to the shell, because they can't be executed independently
  - `cd`, if executed independently would change its own directory, *not* that of your shell
  - `umask`, would change its umask, *not* the shell's
  - `logout`
  - `history`
- Other commands are built in for speed e.g.
  - `pwd`
  - `echo`

2.21 Basic Shell Exercises

What should I focus on?

You should:

- clearly understand what redirection is, and what a pipe is
- understand that it is the *shell* that expands the filename generation symbols (`*`, `?` and `[...]`), and not `echo`, `ls`,...
- understand the differences between these three ways of quoting: `'...'`, `"..."`, and quoting a single character with `\`
- understand how to put a process into the background, bring it to the foreground, and how to interrupt it (*not* with `Control-z`!)
- understand the difference between using `Control-c` and `Control-z`
- understand that subshells have their own working directories.

The exercises

1. *Redirection*

(a) Try typing the following commands exactly as they appear here. Refer to sections 2.6 on page 37 to § 2.9 on page 40 first. Also refer to section 2.13 on page 44 before doing the last exercise.

```
$ cat > newfile 2> newfile.error
$ car > newfile 2> newfile.error
$ car
$ Space Shuttle
$ echo
$ echo foo
$ ech
$ ech foo
$ cat > newfile 2>&1
$ car > newfile 2>&1
$ cat < newfile
$ echo foo | cat > newfile 2>&1
$ ech foo | cat > newfile 2>&1
$ echo foo | car > newfile 2>&1
$ (ech foo | cat) > newfile 2>&1
```

After each step, examine the contents of `newfile` and `newfile.error` by typing:

```
$ cat newfile
$ cat newfile.error
```

Make sure you understand what happens in each case, ask the tutor if you are not sure.

2. *Filename expansion and Quoting*

(a) Do the following in the `/bin` directory; refer to section 2.18 on page 49 first.

- **i.** List all filenames with exactly three characters.
- **ii.** List all filenames with exactly three characters in which the second character is a vowel.
- **iii.** List all filenames with a, b, c, or d as the last character.
- **iv.** Construct a command to print the number of filenames consisting of exactly three characters. (You may combine commands together in a pipeline. You may find the wc utility useful here; check `man wc` for more information.)
- **v.** Construct a command to print the total number of files with exactly two, three or four characters in their name. (Again, you may find the wc utility useful.)

(b) Compare the effect of the following commands. Refer to section 2.19 on page 50 first.

```
echo $HOME
echo "$HOME"
echo '$HOME'
echo *
echo "*"
echo '*'
echo $HOME/*
echo "$HOME/*"
echo '$HOME/*'
```

(c) Change back to your home directory and try to create a file with the name `*`. Was this a sensible thing to do? How would you delete it? (Be *very* careful!)

(d) Create a file called `--file`. Try to remove this. Use the `rm man` page to help you.

3. *Background processes and* `nohup`

Refer to section § 2.10 on page 41 to $sec:basic-shell-and-tools-approach-background-and-nohup first.

(a) Start the command `sort /dev/random` in the background in your current shell

(b) Bring it back to the foreground and terminate it by typing ˆC

(c) Start it again, and once more so that you have two copies running in the background

(d) Bring them to the foreground and terminate them in the order you started them

(e) Start the same command in the background, and terminate it using `kill`

4. *Grouped commands*

Compare the following command sequences, and make sure you understand the differences. Refer to section § 2.13 on page 44 first.

(a)
```
$ cd /tmp
$ cd /usr; ls
$ pwd
```

(b)
```
$ cd /tmp
$ (cd /usr; ls)
$ pwd
```

(c) `$ sleep 5; sleep 5 &`

(d) `$ (sleep 5; sleep 5) &`

Check you can use your history to get at and repeat any of the commands you have typed.

