# FIT9134
# Computer architecture and operating systems

MONASH University
Information Technology

**Week 7**
**Operating System VI:**
**More Unix Shell basics**

# Shell Commands & Shell Scripting

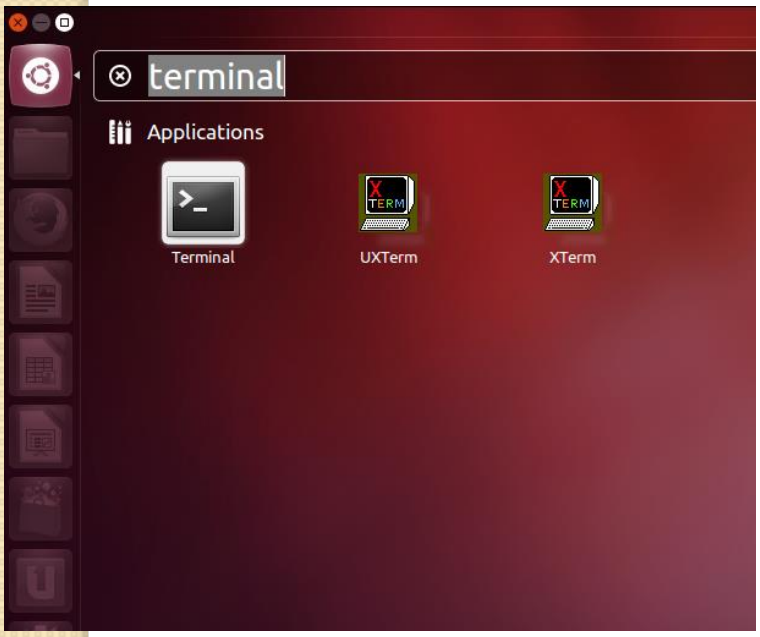
- Important:
  - This lecture, plus the lectures for Week 8 and Week 9, contain materials which will be essential for the 4 ***Lab Sessions*** on ***Shell Scripting*** in Weeks ~~9-12~~ 8/10/11/12.
  - Make sure you **attend all these 3 lectures**, otherwise you will have great difficulties completing those coming Session Tasks.

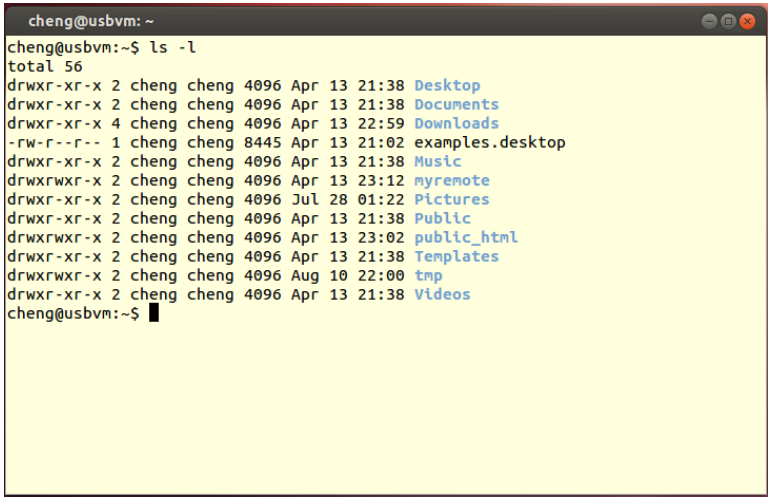
# What is a Unix "shell"?

- a Unix "shell" is a ***command-line interpreter***:
  - accepts & executes text commands from user
  - runs shell scripts (programs written in a shell scripting language) to perform tasks
- provides the traditional user interface for a Unix operating system; each user is given a "default shell" when the user account is created.

# What is a Unix "shell"?

- modern Unix systems often provide a semi-graphical version of the shell, eg. the ***Ubuntu Terminal*** window:



# 2 Major Unix Shells

- ***Bourne shell***
  - developed by Stephen Bourne at Bell Labs in the late 1970s
  - supports shell scripting
  - typically in **/bin/sh**

- ***C shell***
  - developed by Bill Joy at the University of California in the late 1970s
  - designed to be more interactive & user-friendly
  - scripting language resembles C language
  - typically in **/bin/csh**

# Which shell versions?

- major ***Bourne shell*** variants:
  - **Korn** shell (David Korn, Bell Labs)
  - **bash** (Bourne again shell – GNU – does include some **csh** features)
  - **ash** (Kenneth Almquist – another open source version of Bourne shell) – dash is Debian ash for the Debian Linux distribution
  - **zsh** (Z shell – Paul Falstad, Princeton student – close to **ksh**)

- major ***C shell*** variant:
  - **tcsh** (Ken Greer – enhanced **csh**)

- which shell to use is often a matter of personal choice, but remember that typically **Bourne** shell variant scripts do not run in **C** shell variants, and vice-versa. There may also be commands which are specific to each variant.

# How to check/change login shells

Check:

- *echo $SHELL*
- *ps*
- examine */etc/passwd*

Change:

- *chsh*

# Standard Input, Output and Error

- Remember, in Unix, *everything is a file…*
- Every time a shell is started, 3 files are opened automatically: ***stdin***, ***stdout***, ***stderr***

| *File* | *Default Device* | *File Descriptor* |
|---|---|---|
| ***stdin*** | terminal input | 0 |
| ***stdout*** | terminal output | 1 |
| ***stderr*** | terminal output | 2 |

# Output redirection (> symbol)

- Unix commands generally send output to ***stdout***
- The output can be "*redirected*" to a file instead, using the symbol **>**, eg:

```
$ ls -l /etc/passwd > tempfile
$ cat tempfile
-rw-r--r-- 1 root root 29757 Jul 23 9:05 /etc/passwd
```

- The symbol **>** *overwrites* the existing contents of the file. The symbol **>>** *appends* to the end of the file, eg:

```
$ ls -l /usr/bin/passwd >> tempfile
$ cat tempfile
-rw-r--r-- 1 root root 29757 Jul 23 9:05 /etc/passwd
-r-s--x--x 1 root root 13044 Jan 6 2001 /usr/bin/passwd
```

# Input redirection (< symbol)

- The symbol **<** means "*take the input for a program from a file instead of from the terminal*" (***stdin***), eg:

  **$ wc –l** (reads and counts the number of lines typed at the keyboard)

  **$ wc –l < mynotes** (reads and counts the number of lines in the file "mynotes")

- Another example - display the list of logged-in users in sorted order:

  **$ who > temp** (writes output to "temp")

  **$ sort < temp** (reads input from "temp")

  - this could also be achieved with a single command, using a ***pipe***:

    **$ who | sort** (the 2 commands are connected by a pipe, hence no temporary file is needed)

# Error Redirection ( 2> symbol)

- Any command that produces error messages on ***stderr*** can have the messages redirected to another file. Some examples are:

```
$ cat  y
This is y
$ cat  x  y
cat: cannot open x
This is y
```

```
$ cat  y  >  file1

$ cat  x  y  >  file2
cat: cannot open x
```

What are the contents of **file1** & **file2** after these commands?

```
$ cat  x  y  1>  file1  2>  file2

$ cat  file1
This is y
$ cat  file2
cat: cannot open x
```

In this example,
the file **y** contains "**This is y**",
the file **x** does <u>not</u> exist

<u>Abbreviations :</u>
**<0** is the same as **<**
**1>** is the same as **>**

# Be Careful with Output Redirection

- Do not use the same file name as an argument to a command ***AND*** as the output file destination at the same time. You will not get the correct output file, and you may destroy the original file, eg:

$ **cat f1 f2 > f1**

(**f1** may end up with the contents of just **f2**)

The safe way is:

$ **cat f1 f2 > temp**

$ **mv temp f1**

Result will be highly unpredictable. Some **shell** will be smart enough to reject such a command.

# Unix Pipes ( | symbol )

- *Unix Pipes* allow you to use the output of one command directly as input of another command, ie:

**cmd1 | cmd2 | cmd3 | .....**

Eg:

**who | sort | less**

# Pattern matching (wildcards)

- Special characters *, ? and [ ] can be used to abbreviate filenames. The shell then generates the complete filenames.
- ? matches any single character
- * matches any group of characters (except a leading period)
- [ ] surrounding a group of characters causes the shell to match filenames containing the individual characters

# Pattern matching: Examples

- **ls**
  *amemo mem memo memo.0612 memoa memorandum memosally sallymemo user.memo*

listing all files in current directory

- **ls memo?**
  *memoa*

"**filtering**" the results using wildcards

- **ls memo***
  *memo memo.0612 memoa memorandum memosally*

- **ls memo[ar]***
  *memoa memorandum*

# Command Substitution

- When a command is enclosed between two grave accent marks (` or sometimes called backquotes), the shell *substitutes it with the output of the command*. For instance, the following command:

**$ *echo* there are *`ls | wc -l`* files in my directory**

would output something like:

the command enclosed in the backquotes is substituted by its result (52 in this example)

**there are 52 files in my directory**

the **echo** command simply prints its arguments on the terminal

Do not confuse the ***backquote*** (`) with the ***single quote*** (')!

# Command Sequences

- We can execute several commands in sequence using the "**;**" operator:

***date ; who ; ps -ael***

(display today's date; *then* displays who is on the system; *then* find what processes are running. *The 3 commands are not connected*.)

This is an example of ***unconditional* sequencing**.

# Metacharacters

- The shell has many ***metacharacters*** with special meanings – some we have already seen or used:

| Metacharacter | Meaning |
|---|---|
| **>** | output redirection |
| **<** | input redirection |
| ***** | wild card : matches zero or more characters |
| **?** | wild card : matches any ***single*** character |
| **[...]** | wild card : matches any ***single*** character (from the set of characters within the brackets) |
| **\|** | Pipe symbol |
| **$** | access the ***value*** (ie. content) of a variable |
| **** | remove the special meaning of the character immediately following the back-slash character |
| **`command`** (backquotes) | Command Substitution; the ***backquotes*** will be replaced by the output from command (e.g: **result=`who -l`**) |
| **'  '** | remove the special meaning of all characters enclosed by the single quotes |
| **"  "** | remove the special meaning of all characters enclosed by the double quotes, except the '**$**' and '****' characters |

# Shell Variables

- There are two types of shell variables:
  - **local and environment**

- Examples of some predefined environment variables:

| | |
|---|---|
| **$HOME** | the full pathname of your home directory |
| **$PATH** | a list of directories to search for commands |
| **$USER** | your user name |
| **$SHELL** | the full pathname of your login shell |
| **$TERM** | the type of your terminal |

……..

- To access the value of a variable, the variable name is preceded by the $ sign, eg :

  **$ echo Current username is $USER**
  **Current username is andy**

# Shell Variables

- Some shell variables are inherited from the environment; others are created by the shell when it starts up. To view those variables in the **bash** shell, use the **set** command:

**$ set**

| | |
|---|---|
| HOME=/home/cwilson | your home directory |
| HOSTNAME=myvm | your machine name |
| PATH=/usr/bin:/usr/sbin: | command search path |
| TERM=vt100 | terminal type |
| PS1="$ " | prompt string 1 |

……

- Depending of the shell used, some other related commands are :

  **printenv**, **setenv**

# User-defined Variables

- Users can define their own "local" variables.
- A variable name starts with a letter and contains only alphanumeric characters.
- The shell substitutes the value of a variable only when it is preceded by a **$**, unless there are other special characters.

```
$ person=alex
$ echo person
person
$ echo $person
alex
$ echo '$person'
$person
```

← note : do ***NOT*** put spaces around the '=' sign

```
$ echo "$person"
alex
$ echo \$person
$person
```

# How does the Shell find commands?

- When a command is entered, the shell
  - first checks whether it is a built-in command
  - else, checks if the command starts with a / (ie. an absolute path)
    - if an absolute path is given, shell executes the file as a command
  - else, searches the directories specified in the environment variable **PATH**, from left to right, for an executable file that matches the command name
    - to see the current ***command search path***, type:

```
echo $PATH
.:/usr/ucb:/bin:/usr/bin        (a typical search path)
```

- Note : *by default, the path typically does not include the current directory*. This often creates slight confusions when the user tries to run a command in the current directory. How would you fix this?

# Process *Exit Value*

- Every UNIX process terminates with an ***exit value*:**
  - **0** means success
  - **non-zero** (usually 1) means failure

- The variable that contains the last exit value is **?**. Hence **$?** will contain the exit value of the ***last command executed***.

# Conditional Sequencing

- ***Conditional*** sequencing examples :

  **$ prog1 && prog2**

  - execute prog2 only if prog1 was successful

  **$ prog1 || prog2**

  - execute prog2 only if prog1 returns error (ie. fails)

# Filters

- A "***filter***" is typically a command that reads the output from one program (or standard input or a file) and produces some modified output
- It ***filters*** (based on the program) the contents of its input stream
- It sends its results to standard output, but never modifies the input stream or file
- Output results may be further processed by another filter

- Examples of some common filters:
  - **sort** - sort on specified fields
  - **wc** - count lines, words and characters
  - **grep** - search files for a pattern (see also lecture on regular expressions)
  - **tr** - translate specified characters

# Examples

- **ls /etc | grep conf | wc –l**

  list all the files in **/etc**, then search for the string **conf**, then count lines

- **ls –l /etc | grep ^d | sort –r > /tmp/directories**

  list the files in **/etc** in long format,
  then search for strings beginning with **d**,
  then sort in reverse alphabetic order,
  then saves the output to **/tmp/directories**