MONASH University
Information Technology

FIT9134
Computer architecture and operating systems

Week 8+9

Operating System VII:
Unix Shell Scripting

Shell Commands & Shell Scripting

- Reminder:
  - This lecture covers 2 weeks (Week 8+9), and contain materials which will be required for the 4 ***Lab Session Tasks*** on ***Shell Scripting*** in Weeks 8+10+11+12.
  - Make sure you **attend both these 2 lectures (& revise the lecture from Week 7)**, otherwise you will have great difficulties completing those coming Session Tasks.

Shell Script Basics

- A ***Unix shell script*** is just a ***text file containing Unix commands***. Any command that can be given at the normal shell prompt can also be put into a shell script. When shell scripts are executed, the commands in the script are executed, as if they are typed at the command line prompt.

- Any command that can be issued at the command prompt can be included in the shell script. There are also some additional commands which can be used in a shell script, such as flow control commands (eg. if-else, while-loop, etc).

Shell Script Basics

- Script files should generally have *read* and *execute* permissions set (*execute* permission is not needed if script is executed via the **`sh`** command).

- User script files are often stored in a **`bin`** subdirectory under the user's ***home*** directory. System script files are typically keep in standard subdirectories, such as **`/bin`**.

- To avoid confusion, do not name a script file using the name of a standard/common UNIX command, such as '**`test`**', '**`pwd`**', '**`ls`**', etc

Shell Script Basics

- There are two ways to run a script file:
  1) type its name at the prompt (the script file must first be given execute permission by '**chmod u+x** ...'), OR
  2) type **sh** ***script-name*** at the prompt

- **#** is used as a *comment* character, except:
  - if the first 2 characters of a script are **#!**, then the system expects a shell's pathname to follow, as shown below:

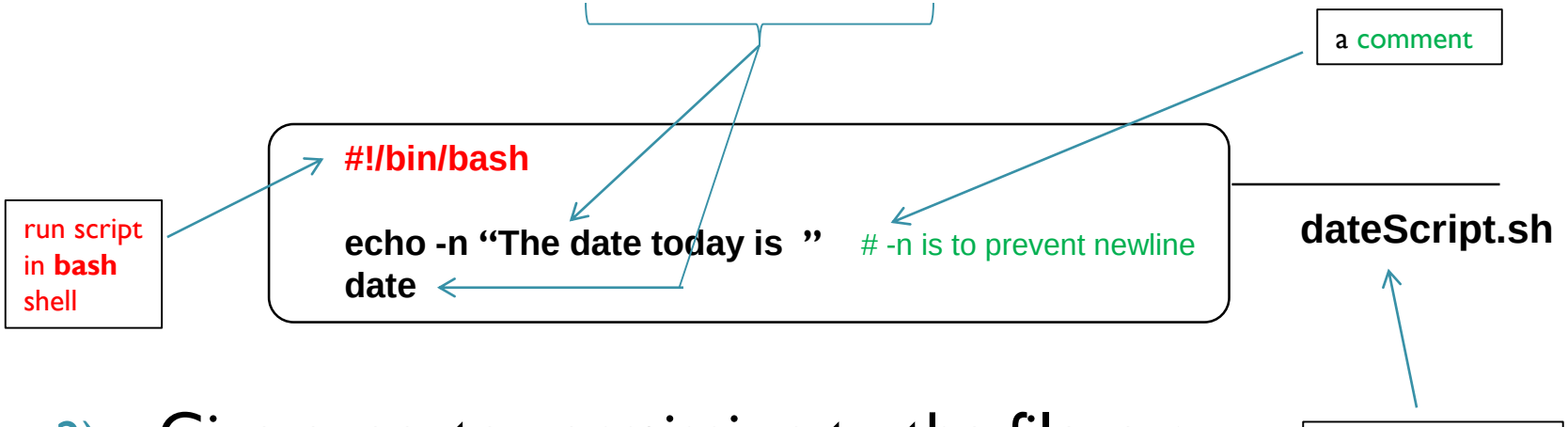
#!/bin/bash

…. (the rest of the script)

this means : "run the rest of this script using the **/bin/bash** shell"

How to create a shell script?

1) Create a text file (eg. using the ***vi*** editor), eg:
   - **$ vi dateScript.sh**
   - add a sequence of shell commands in the file, eg:



2) Give execute permission to the file, eg:

   $ chmod u+x dateScript.sh

name of the script (commonly ends with **.sh**)

3) Use the file name as command, eg:

   $ dateScript.sh

or, you may need to type **./dateScript.sh**

The date today is Mon Aug 24 08:46:45 EST 1997

command prompt

Different Shells

- Shell scripting commands/syntax are generally not compatible between different shells
  - ie. a script written for the Bourne Shell may not run properly in the C Shell, and vice-versa.

- For the rest of the semester, we will write all scripts with the **bash** shell – so make sure you default login shell is `/bin/bash`

Variables - again

- Variables are set as follows:

  name=value

  NO spaces around the '=' operator

- Setting shell variables can be done at the command line, but is usually done in a special "startup" file. In the *Bourne Shell*, this file is **.profile**
  - for individual users, this file is typically stored in their home directories, and are executed automatically whenever the users log in
  - system-wide startup files are also often present, and stored in system directories, eg: /etc/profile

this file is automatically executed at login time – its actual name depends on your login shell

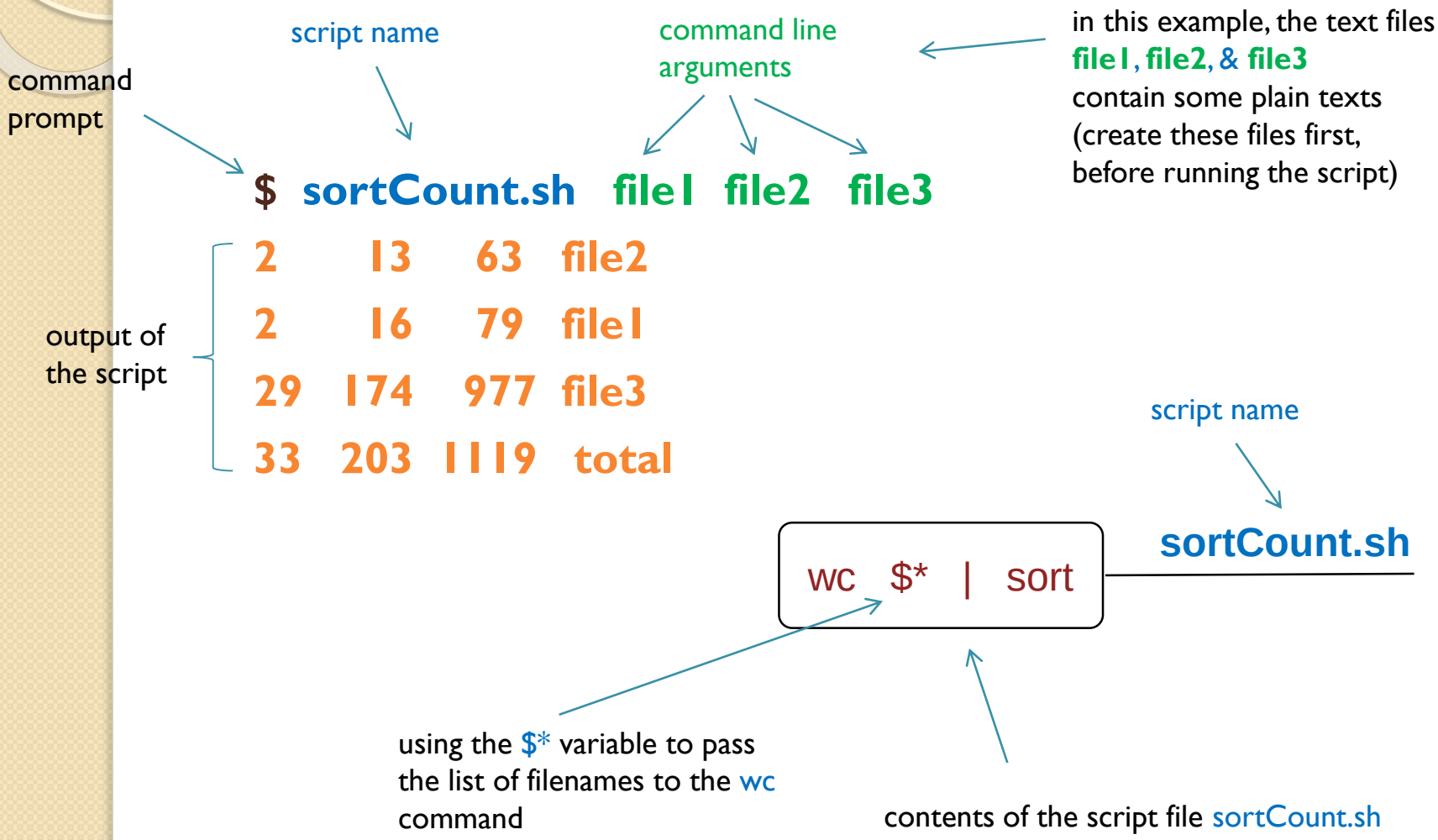
Some useful built-in special variables

- most of the special variables listed below are meant to be used only within shell scripts:

Name of variable	Content of variable
$0	the ***name*** of the shell script (if applicable)
$1 … $9	$n refers to the nth ***command line argument***
$#	the ***number of arguments***
$*	a ***list of all*** the *command line arguments* *(not counting the command name itself)*
$?	***Exit status*** of a script (more on that shortly)

- it is generally considered good practice to quote variables when using them in scripts, eg. "**$#**", "**$number**", etc (there are some exceptions)

Example : using special variables inside scripts



Note : no spaces around the = sign

Double quotes

- ***Double quotes ("…. ")*** generally turn the contents into a string. They could also be used to preserve spacing in a variable's value. However, some special characters will retain their meanings.

Eg:

command prompt

```
$ person="alex    and    jenny"

$ echo  $person
  alex and jenny

$ echo  "$person"
  alex    and    jenny
```

The original string contains multiple spaces between the words

The variable is ***expanded***
Note how the multiple spaces within the string are lost when they are passed to the **echo** command

Note how the variable still gets ***expanded*** within the double-quotes (as the $ character retains its meaning), but the multiple spaces are preserved.

Single quotes

- ***Single quotes ('….')*** are similar to Double quotes, but are "stronger" – ie. allows the user to ***quote special characters without expansion.***

Eg:

```
$ person="alex     and     jenny"

$ echo  $person
  alex and jenny

$ echo  '$person'
  $person
```

The variable is ***expanded*** and the actual content passed to **echo**

Note how the variable is no longer ***expanded*** within the single-quotes, as everything within the single quotes becomes just a string.

Example – Command Arguments

contents of the script file **prog1.sh**

- $ **cat prog1.sh**

```
echo You are running program: "$0"
echo Argument \#1 is: "$1"
echo Argument \#2 is: "$2"
```

- $ **chmod u+x prog1.sh** (give execute permission)
- $ **prog1.sh apple banana**

```
You are running program: prog1.sh
Argument #1 is: apple
Argument #2 is: banana
```

- note : a **#** on the command line normally represents a comment – hence if we want to print the **#** character itself as a normal character, it must be ***quoted*** (or "***escaped***") with a backslash as shown above.

output of **prog1.sh**

shift - promote arguments

- The **shift** command moves all command line arguments to the left one position: the original **$1** is lost, **$1** now takes on the value **$2** had, **$2** takes on the value **$3** had,...,**$9** takes on the value of the tenth command line argument. *So more than 9 arguments can now be retrieved by using the **shift** command.*

Eg:

$ **cat prog2.sh**

script

```
orig_args="$*"
echo There are "$#" args
echo They are: "$*"
shift
echo There are "$#" args
echo They are "$*"
echo Original args are: "$orig_args"
```

$ **prog2.sh red yellow blue**

```
There are 3 args
They are: red yellow blue
There are 2 args
They are yellow blue
Original args are: red yellow blue
```

output

Sample Script (with user interaction)

script

```
$ cat myinstall.sh
echo "$0" will move files to your bin directory
echo –n "Enter the filenames to move:"
read filenames
chmod u+x $filenames
mv $filenames "$HOME"/bin
echo Installation is complete
```

this command reads a user input (via keyboard), and stores it into the variable called "**filenames**"

output

```
$ myinstall.sh
myinstall.sh will move files to your bin directory
Enter the filenames to move: myvi.sh myrm.sh
Installation is complete
```

The user typed in some inputs. The inputs will be stored in variable "**filenames**"

```
$ ls $HOME/bin
myvi.sh   myrm.sh
```

files are now in new location, with execute permissions set

The **expr** command (evaluating *Expressions*) – some examples

note the use of the *back-quotes* to assign the result of **expr** into a variable

```
$   y=1
$   x=`expr "$y" + 1`
```

performing simple calculation

```
$   echo "$x"
2

$   echo `expr length "maths"`
5
```

finding length of a string

The **expr** command (evaluating *Expressions*) – some more examples

```
$   echo `expr substr "donkey" 4 3`
key
```

extracting a substring from a given string

```
$   add.sh 10 12
22
```

add.sh

```
#!/bin/bash

echo `expr "$1" + "$2"`
```

a simple "addition calculator" script

exit

- The **exit** command can be used to terminate a shell script, and optionally return an ***exit status*** value (for example to indicate error condition)

Eg:

exit 0 Exits and signifies no error on termination

exit 1 Exits and signifies an error on termination

By convention, any number other than 0 indicates an error condition

test – Evaluate & Test an Expression

- **test** evaluates an expression, returning either ***true*** *(0)* or ***false*** *(1)*

** Important **

- Syntax: **test *expression***
  OR, **[*expression*]**

must have spaces around [ ]

- *expression* contains one or more criteria:
  - **-a** between criteria is a logical ***AND*** operator
  - **-o** between criteria is a logical ***OR*** operator
  - any criterion can be negated by preceding it with **!**
  - criteria can be grouped with parentheses
  - special characters within an expression must be quoted to prevent interpretation by the shell

test - Numeric Tests

Syntax: **[*number relation number*]**

relation:

-lt	less than
-le	less than or equal to
-gt	greater than
-ge	greater than or equal to
-eq	equal to
-ne	not equal to

Note : ***spaces*** between the **test** expression & the [ ]'s are ***NOT optional***

```
$ [ "$count" -lt 9 ]      (assume count contains 7)
$ echo $?                 (quoting prevents errors
  0                        when count is null)
```

test - String Tests

Syntax:	**[*string1* = *string2*]**	(equal?)
	[*string1* != *string2*]	(not equal?)
	[-n *string*]	(not zero-length?)
	[-z *string*]	(zero-length?)

Note : spaces around = sign

```
$ x=abc
$ [ "$x" = "abc" ]
$ echo $?
0
(no error)
```

```
$ [ -z "$x" ]
$ echo $?
1
(error)
```

Error status is stored in the special variable **$?**

test – common File Tests

Syntax:

[-f *file*]	true if *file* exists and is an ordinary file
[-d *file*]	true if *file* exists and is a directory
[-r *file*]	true if *file* exists and you have r access
[-w *file*]	true if *file* exists and you have w access
[-x *file*]	true if *file* exists and you have x access
[-s *file*]	true if *file* exists and its size is non-zero
etc…	(see **man test**)

Common scripting errors involving ***spaces***

- A common error is the incorrect use of ***spaces*** around operators, eg:

[-z "$x"] error: missing spaces near the [ ]'s

["$x"="abc"] error: missing spaces around =

y = 20 error: extra spaces around =

Common scripting errors involving ***variables, quotes***

- Some other common errors:

```
if [ number -lt 5 ]
```

error: forgetting to put the '$' (ie. $number) for variables

```
result='expr "$a" + "$b"'
```

error: used the wrong type of quotes (ie. should have used backquotes (`…`) instead of single quotes)

if-then Control Structure

Syntax:

```
if test-command
then
    commands
fi
```

- eg. checking for correct number of command line arguments:

```
if [ "$#"  -eq  0 ]
then
    echo  Error :  at least one argument must be specified
    exit 1
fi
```

note how the code returns an error status of 1

if-then-else Control Structure

Syntax:

```
if test-command
then
    commands
else
    commands
fi
```

Eg :

```
if [ "$var1" = "$var2" ]
then
   echo var1 and var2 are the same
else
   echo var1 and var2 are different
fi
```

Example : using if-then-else Control Structure in a script

countUsers.sh

```
#!/bin/bash
# Print the count of users

count=`who | wc -l`

if [ "$count" -eq 1 ]
then
  echo There is one user
else
  echo There are "$count" users
fi
```

if-then-elif Control Structure

Syntax:

```
if test-command
then
    commands
elif test-command
then
    commands
        .
        .
else
    commands
fi
```

if-then-elif Example

```
if [ "$word1" = "$word2" –a "$word2" = "$word3"
then
    echo "Match: words 1, 2, & 3"
elif [ "$word1" = "$word2" ]
then
    echo "Match: words 1 & 2"
elif [ "$word1" = "$word3" ]
then
    echo "Match: words 1 & 3"
elif [ "$word2" = "$word3" ]
then
    echo "Match: words 2 & 3"
else
    echo No match
fi
```

-a means "***AND***"

"for-in" Control Structure

Syntax:

```
for loop-index in argument-list
do
        commands
done
```

- The arguments in the list are assigned, one by one, to the *loop-index* and the *commands* between the 'do' and 'done' are executed.

- If "in *argument-list*" is omitted, the *loop-index* automatically takes on the value of each of the command line parameters one at a time - this is equivalent to:

```
for loop-index in $@
```

Example: for-in Construct

```
$ cat users
alice
bob
campbell
```

Note: to run this example, you need to create the data file first, ie : create the file called "**users**", and put in these 3 lines of data as shown.

Eg :

```
for name in `cat users`
do
    echo Hello "$name", how are you today?
done
```

script

what is the output when we run this script?

while Construct

Syntax: **while** ***test-command***
do
commands
done

Eg:

```
answer=y

while [ "$answer" = y ]
do
      echo Enter a name
      read name
      echo "$name" >> names.txt
      echo –n "Continue [y/n]:"
      read answer
done
```

This script will keep asking for a name – as long as the user keeps entering 'y' to the question. The names are all saved to the file called "**names.txt**"

reads an input from keyboard, and store it in a variable

script

Another while example

```
$ cat count.sh
```

```
number=1
while [ "$number" -lt 10 ]
do
    echo –n "$number"
    number=`expr "$number" + 1`
done
echo
```

```
$ count.sh
123456789
```

until Construct

Syntax:

```
until test-command
do
    commands
done
```

Example : until

```
until [ "$#" -eq 0 ]      # number of command-line arguments
do
   if [ -d "$1" ]         # if first argument is a directory
   then
        echo Contents of "$1":
        ls -F "$1"        # list files in directory
   fi
   shift
   echo There are "$#" items left on the command line
done
```

break and continue

- a **`for`**, **`while`** or **`until`** loop can be interrupted with a **`break`** or **`continue`** command.

- **`break`** terminates execution of the loop completely by transferring control to the statement just after the **`done`** statement.

- **`continue`** transfers control to the **`done`** statement and hence terminates just one iteration of the loop.

case Construct

Syntax:

```
case test-string in
        pattern-1)
                commands-1
                ;;
        pattern-2)
                commands-2
                ;;
        pattern-3)
                commands-3
                ;;
        .........
esac
```

pattern is like an ambiguous file reference (***?[ ]** can be used). Can also use the **|** character to separate choices.

Example : case Construct

```
echo –n "Enter A, B or C: "
read character
case  "$character"  in
   a|A)
         echo You entered a/A
         ;;
   b|B)
         echo You entered b/B
         ;;
   c|C)
         echo You entered c/C
         ;;
   *)
         echo You did not enter A, B, or C
esac
```

this means: if character has the value of '**a**' OR '**A**'

this means: if character does not match any of the above

Example (a menu-displaying script)

```
#!/bin/bash
echo menu test program
stop=0
while [ "$stop" -eq 0 ]
do
   cat MENUOPTIONS
   echo
   echo -n 'your choice: '        #prompt for input
   read reply
   echo
```

Contents of the file MENUOPTIONS

```
1 :   display the date.
2, 3: display logged-on users
4 :   exit
```

print a blank line.

```
# script continued on next page…
```

Example Continued

```
  case "$reply" in
  "1")
     date          #display date
     ;;
  "2"|"3")
     who           #display logged-on users
     ;;
  "4")
     stop=1        #finish (exit script)
     ;;
  *)
     echo illegal choice
     ;;
  esac
  echo
done
```

Debugging Script Files

- To view each command in a script file as it executes:
  - put '**set -x**' at the start of the script, *OR*
  - run the script using the command '**sh -x *scriptname***'
- Each command of the script will be printed on standard output, after evaluation (and substitution) by the shell.

like this

```
$ cat check.sh
echo Enter your name:
read name
if [ -z "$name" ]
then
  echo Nothing entered
fi
```

script

```
$ sh -x check.sh
+ echo Enter your name:
Enter your name:
+ read name

+ [ -z ]
+ echo Nothing entered
Nothing entered
```

output

user entered nothing here
(ie. simply pressed <**enter**>)

Some Useful References

- ***The UNIX Programming Environment,*** B.Kernighan, R.Pike
- ***The Unix System V Environment***, S.Bourne
- http://linuxcommand.org/tlcl.php
- http://www.freeos.com/guides/lsst/
- http://www.topbits.com/unix-shell-scripting-tutorials.html