# KIT104 ICT Architecture and Operating Systems
# Tutorial Week 8

## 1. UNIX Shell Scripting/Programming Concept

In UNIX, you can **store a group of commands in a text file and then execute the file** to automate some administrative task.

All such text files are called **shell scripts**, or **shell programs**. A UNIX shell as a command interpreter is also a programming language, and has all the standard constructs like `if`, `while` and `for` (borrowed from C Programming Language but simpler to use than those in C).

Let's start with writing a simple shell script/program.

| | |
|---|---|
| A | Login to your alacritas account. Change into `kit104` directory.<br><br>Remove any existing files or subdirectories stored under `kit104` (unless you really want to keep them). |
| B | Use `joe` or `nano` to create a text file which must be named as `info.sh`, with the following file content:<br><br>`# My first shell script`<br>`# I'm a little excited …`<br><br>`echo The date today is` `` `date` ``<br>`echo The current directory is` `` `pwd` ``<br>`echo My home directory is`<br>`cd`<br>`pwd`<br>`echo The current users are`<br>`users`<br>(Note: press the `Enter/Return` key at the end of the last line)<br><br>Exit the text editor by saving the file content.<br><br>You have now created your first UNIX shell script/program. |
| C | Unlike in C or Java programming, you do NOT need to compile a shell program. Additionally, the extension (`.sh`) in the file name is optional, but having this extension helps you identify the type of this file (which is a shell script). |
| D | You can execute this program by simply typing the shell script name:<br><br>`$ ./info.sh`<br>`info.sh: permission denied` |

| | |
|---|---|
| | The reason for the above error message is that, by default, there is no `x` (execute) permission for newly created text files:<br><br>`$ ls -l info.sh`<br>`-rw-r--r-- 1 sxu staff 78 Aug 25 16:33 info.sh`<br><br>So you will need to assign `x` permission to it manually:<br><br>`$ chmod u+x info.sh`<br><br>`$ ls -l info.sh`<br>`-rwxr--r-- 1 sxu staff 78 Aug 25 16:34 info.sh`<br><br>Now you can run it. The following is a *sample output*:<br><br>`$ ./info.sh`<br>`The date today is Thu Aug 25 16:39:48 EST 2016`<br>`The current directory is /u/student/jchen/kit104`<br>`My home directory is`<br>`/u/student/jchen`<br>`The current users are`<br>`cmcgee john tgray`<br><br>Compare your output of the script against each command line contained within the script content. Can you see that each command line has been executed one after another? |
| E | Can you guess the purpose of the first two lines in this shell script?<br><br>`# My first shell script`<br>`# I'm a little excited …`<br><br>Yes these are *comment lines* which help users understand the purpose of this script. Comment lines are for humans to read. They are ignored by UNIX shells. |

## 2. UNIX Shell Variable

Like in Java or C, we can also define and use **variables** in a shell script.

In Java or C, a variable has a type (such as `int`, `char`, etc).

A shell variable is of the *string* type. However, NO type declaration is necessary. The general format is as follows:

`variable=value` (No whitespace on either side of =)

After a variable is created and assigned a value, you can use the `$` sign to evaluate it (ie, retrieving its value).

A

Create the following variables named `x` and `y` respectively, and then evaluate them. Note that, in each line, the first `$` is the shell prompt.

```
$ x=foo
$ echo $x
foo

$ y=.doc
$ echo $y
.doc
```

= 等号两边不能有空格

You can use the command `unset` to remove variables:

注意unset是没有$符号

```
$ unset x
$ echo $x

$ unset y
$ echo $y
```

You can use shell variables to construct file names.

```
$ x=foo
$ y=.doc
$ z=$x$y

$ echo $z
foo.doc

$ unset x
$ unset y
$ unset z
```

B

Let's create more shell variables:

```
$ msg1='You have mail'
$ echo $msg1
You have mail

$ msg2=You\ have\ more\ mail
$ echo $msg2
You have more mail
```

(note the single quotes)

Why is it necessary to use single quotes or \ in the above variable creation? The reason is that the whitespace on a UNIX shell is also a special character. Try the following:

```
$ msg=You have mail
have: command not found

$ echo $msg
```

不加''单引号或\ 空格前就会被当成命令处理
赋值为空

| | |
|---|---|
| | Here the shell understands that you try to assign `You` as value to a variable named `msg`, and then run the `have` command. Given that there is no such command named `have`, the command line has failed. |
| C | Let's create the shell variable named `msg` again:<br><br>`$ msg='You have mail'`<br>`$ echo $msg`<br>`You have mail`<br><br>`$ echo "$msg"` 双引号会输出变量内容<br>`You have mail` 单引号会把变量当字符串直接输出<br><br>`$ echo '$msg'`<br>`$msg`<br><br>*Here you see the other difference between the use of single quotes and double quotes.*<br><br>*Double quotes allow the use of $ to evaluate shell variables. Single quotes do not.*<br><br>Summary about the difference between single quotes and double quotes on UNIX:<br><br>*Enclosing a group of characters in single or double quotes can remove the special meaning of the characters. Though the shell ignores all special characters enclosed in single quotes, double quotes permit using $ to evaluate a variable and back-quote ` for command substitution.*<br><br>The following exercises reinforces the above summary. |
| D | `$ echo The newline character is \n` n<br>`$ echo 'The newline character is \n'` \n<br>`$ echo "The newline character is \n"` \n<br><br>`$ echo There are `who \| wc -l` users on the system` 26<br>`$ echo "There are `who \| wc -l` users on the system"` 26<br>`$ echo 'There are `who \| wc -l` users on the system'` echo the `who…<br><br>`$ msg='We will depart at 10AM'`<br>`$ echo The message is $msg` the msg is we will depart at 10am<br>`$ echo The message is "$msg"` the msg is we will depart at 10am<br>`$ echo The message is '$msg'` the msg is $msg |

## 3. Specifying a Shell, and Interactive Shell Scripts

| | |
|---|---|
| A | You can specify a UNIX shell for a script to run on.<br><br>Add the following line to the beginning of your `info.sh` script:<br><br>`#! /bin/sh`<br><br>Save and run this script again. From now on, this script will also run on the Bourne shell, regardless of what shell you are currently using. This is useful because there are differences between UNIX shells. A script written to run on one shell may not run as expected on a different shell. |
| B | You can make shell scripts interactive, using the `read` command.<br><br>Create the following shell script. Save and name it as `emp1.sh`:<br><br>`#! /bin/sh`<br><br>`# This is an interactive script`<br><br>`echo -e "Enter the pattern to be searched: \c"`<br>`read pname`<br>`echo -e "Enter the file to be used: \c"`<br>`read flname`<br>`echo -e "Searching for $pname from $flname\n"`<br>`grep "$pname" $flname`<br>`echo -e "\nSelected lines shown above"`<br><br>–e "Enter the pattern to be searched<br>hi<br>–e "Enter the file to be used: c"<br>i<br>–e "Searching for hi from in"<br>grep: i: No such file or directory<br>–e "nSelected lines shown above"<br><br>Assign execute (`x`) permission to this file.<br><br>The script prompts the user to enter a keyword, and then searches through a record file to find all the entries which contain the keyword. The first `echo` command prompts the user to enter a keyword. The first `read` command stores the user input into the variable `pname`. The second `echo` command prompts the user to enter a record filename. The second `read` command stores the user input into the variable `flname`. The `grep` command does the job and displays the result.<br><br>Note the use of the option `-e` and `\c` and `\n` for the `echo` command. The option `-e` with `\c` place the cursor at the end of the message line for accepting user input. The `\n` is used to generate a new blank line which improves the layout of an output. |
| C | To correctly run this script, you need a support file.<br><br>Create a new text file which must be named as `emp2.lst`, with the following content:<br><br>`2000, John Warren, NSW`<br>`2001, Adam Davis, NSW`<br>`3000, John Smith, ACT` |

| | |
|---|---|
| | 3001, Jenny Gray, ACT<br>7000, Elton John, TAS<br>7001, Sam Jones, TAS |
| D | Run the `emp1.sh` script as follows. Ensure that you understand each line in the output.<br><br>`$ ./emp1.sh`<br>`Enter the pattern to be searched:` **`John`** (user input)<br>`Enter the file to be used:` **`emp2.lst`** (user input)<br>`Searching for John from emp2.lst`<br><br>`2000, John Warren, NSW`<br>`3000, John Smith, ACT`<br>`7000, Elton John, TAS`<br><br>`Selected lines shown above` |
| E | Please note that the script allows you to search for a pattern with multiple words (such as `John Smith`).<br><br>`$ ./emp1.sh`<br>`Enter the pattern to be searched:` **`John Smith`** (user input)<br>`Enter the file to be used:` **`emp2.lst`** (user input)<br>`Searching for John Smith from emp2.lst`<br><br>`3000, John Smith, ACT`<br><br>`Selected lines shown above`<br><br>The reason is in the following statement of this script:<br><br>`grep "$pname" $flname`<br><br>The use of double quotes here allow you to enter any string of characters as value for the `pname` variable. If you remove the double quotes in this statement, and run the script as above again, you would get unexpected output. |

## 4. Positional Parameters

| | |
|---|---|
| A | A Shell script can read in command-line arguments. The first argument is referred to as `$1`, the second argument is referred to as `$2`, and so on. `$0` refers to the script file name. `$#` refers to the number of arguments in the command line. `$*` refers to the complete set of arguments as a single string. Command-line arguments are referred to as *positional parameters*.<br><br>What are command line arguments? For example, for the following command line:<br><br>`$ grep Andrew /etc/passwd`<br><br>`Andrew` is the first argument (`$1`). `/etc/passwd` is the second argument (`$2`). |
| B | Write the following shell script and save it as `pospar.sh`<br><br>`#! /bin/sh`<br><br>`echo "name of the shell script: $0"`<br>`echo "the number of arguments: $#"`<br>`echo "the first argument is: $1"`<br>`echo "the second argument is: $2"`<br>`echo "the third argument is: $3"`<br>`echo "the complete set of arguments is: $*"`<br><br>$0 是自身命令<br>$# 是参数的个数<br>$1 是第一个参数<br>$2 第二个 $3 第三个 如此类推<br>$* 就是把所有的参数输出出去 |
| C | Assign `x` permission to the script, and then run the script using each of the following command lines. Do you understand the output of each command line?<br><br>`$ ./pospar.sh`<br><br>`$ ./pospar.sh ab`<br><br>`$ ./pospar.sh ab cd`<br><br>`$ ./pospar.sh ab cd ef`<br><br>`$ ./pospar.sh ab cd ef gh`<br><br>`$ ./pospar.sh ab "cd ef" gh`<br><br>`$ ./pospar.sh "ab cd" "ef gh"` |

## 5. Exit Status of a Command

| | |
|---|---|
| A | Every command returns a value after execution, which is the *exit status* or *return value* of the command. The `$?` is used to store the exit status of the last command which has been run on a shell.<br><br>The `$?` has the value of 0 if the command succeeds, and a non-zero (normally 1 or 2) value if it fails.<br><br>To find out whether a command has executed successfully, run the following command line:<br><br>`$ echo $?`<br><br>$? 表示命令执行的结果,成功返回0,否则返回非零,一般是1或2 |
| B | Following the previous exercises you should still have the file `info.sh` stored under your current directory.<br><br>`$ cat info.sh`<br>`$ echo $?`<br><br>Ensure that you do not have a file named `fool` stored under the current directory. Do the following:<br><br>`$ cat fool`<br>`$ echo $?` |
| C | Following the previous exercises you should still have the file `emp2.lst` stored under your current directory. Ensure that you understand the outputs of the following command lines:<br><br>`$ grep John emp2.lst >/dev/null; echo $?`<br><br>`$ grep Jack emp2.lst >/dev/null; echo $?`<br><br>返回0<br>返回1<br>/dev/null 扔走输出<br><br>What is the purpose of the `>/dev/null` in these command lines? |

## 6. Logical Operators

| | |
|---|---|
| A | We can use a command's *exit status* to control the flow of a shell program. This involves using the following logical operators:<br>`&&`<br>`\|\|`<br><br>`cmd1` `&&` `cmd2` - `cmd2` is executed *only* when `cmd1` succeeds<br>`cmd1` `\|\|` `cmd2` - `cmd2` is executed *only* when `cmd1` fails |
| B | Run the following command lines and think about their outputs:<br><br>`$ grep John emp2.lst && echo "Pattern found"`<br>`2000, John Warren, NSW`<br>`3000, John Smith, ACT`<br>`7000, Elton John, TAS`<br>`Pattern found`<br><br>&& 前执行,后面才执行<br>II 前面失败,后面才执行<br><br>`$ grep John emp2.lst \|\| echo "Pattern not found"`<br>`2000, John Warren, NSW`<br>`3000, John Smith, ACT`<br>`7000, Elton John, TAS`<br><br>`$ grep Jack emp2.lst && echo "Pattern found"`<br><br>`$ grep Jack emp2.lst \|\| echo "Pattern not found"`<br>`Pattern not found`<br><br>Why is it that there is no output for the third command line here? |
| C | **In the `emp1.sh` script, modify the `grep` statement so that it becomes:**<br><br>**`grep "$pname" $flname \|\| exit`**<br><br>**Save the change. Run the script two times by entering `John` and `Jack` as the pattern, respectively. Explain to your tutor the purpose of the `\|\| exit` in the `grep` statement. (10 marks)**<br><br>if grep fail then exit |

[chaofanz@alacritas kit501]$ ./emp1.sh
–e Enter the pattern to be searched: \c
John
–e Enter the file to be used: \c
emp2.lst
–e Searching for John from emp2.lst\n
2000, John Warren, NSW
3000, John Smith, ACT
7000, Elton John, TAS
–e \nSelected lines shown above
[chaofanz@alacritas kit501]$ ./emp1.sh
–e Enter the pattern to be searched: \c
Jack
–e Enter the file to be used: \c
emp2.lst
–e Searching for Jack from emp2.lst\n

(The End)

emp1.sh
```
#! /bin/sh
# This is an interactive script
echo  ?~@~Se  "Enter the pattern to be searched: \c"
read  pname
echo  ?~@~Se  "Enter the file to be used: \c"
read  flname
echo  ?~@~Se  "Searching for $pname from $flname\n"
grep  "$pname"  $flname II exit
echo  ?~@~Se  "\nSelected lines shown above"
```