Shell Scripts	**LABORATORY**	**THREE**
OBJECTIVES		
1. Unix / Linux Shell Commands and Usage 2. Basic Shell Script Programming 3. Processing Exit Status 4. Web Pages Publishing		

Unix / Linux Shell Commands and Usage

As a recap, here is a list of common Unix / Linux shell commands used with processes.

Command	Usage	Description
ps	**ps -elf**	Show the processes in the computer
bg	**bg**	Move a stopped foreground job to background
fg	**fg %2**	Move background job number 2 to foreground
jobs	**jobs**	Show the list of background jobs
time	**time a.out**	Show the total execution time of a program and its share of CPU time
uptime	**uptime**	Show the current workload of the computer, e.g., number of users
top	**top**	Show CPU and resource utilization of computer with a list of active processes
kill	**kill %2** **kill 12345** **kill -2 12345** **kill -9 12345**	Terminate a background job Terminate a process Interrupt a process as like using **<Ctrl>-C** Terminate a process more forcefully

Sometimes, users will try to customize their Linux system to look like whatever system they are familiar with, by changing the meaning of commands or create new commands. Recall that this is achieved via the use of **alias**, e.g., make **ls** behave like what you like by setting an alias for the command, **alias ls='ls -l'**, or use the more familiar command **dir** instead of the **ls** command, via **alias dir=ls**.

If you want to define many aliases at the same time, you may put them into a "batch" file and execute it (like **.bat** file in MS-DOS). Then you do not need to type in the alias commands every time you login to the system. Just place all your aliases into a file, e.g., **myalias**. You can then execute this file containing useful aliases by typing **source myalias**. You are encouraged to create your own alias file to customize your Linux. Note that you can also execute the script created in Lab 2, i.e. **allLab2**, via the **source** command, i.e., **source allLab2**. The file does not need to be changed to executable via **chmod** if **source** command is used.

In Unix / Linux, you could know more about the hardware platform that you are using with the **uname** command. Try out **uname -a** on **apollo2** and/or another machine (perhaps even a Mac). You can know the full name of your machine by **hostname**. Recall that **pwd** returns the current path where you are working. Connecting these together with the **whoami** and **pwd** commands, you can change the command prompt easily. In **bash**, the command prompt is defined within the variable **PS1**. Type **echo $PS1** and see the information about the format.

```
echo $PS1
```

On **apollo**, you would see something like **\u \h:${PWD}$**. Here **\u** means the user name (which can be retrieved by **whoami**), **\h** means the machine name (a full version can be retrieved by **hostname**), **${PWD}** evaluates the *variable* **PWD** in the current shell. This variable contains the current (present) working directory in full (which can be retrieved by the *command* **pwd**).

You may want to make the prompt like

Mickey@apollo2:

you could type

export PS1="Mickey@\h: "

To make the prompt like

(/home/12345678d) 12345678d-

you could type

export PS1="($PWD) $(whoami)- "

To make the prompt like

Linux@apollo2 [30]:

where 30 is the current command number, you could type

export PS1="$(uname)@\h [\!]:"

Many Unix / Linux functions are defined based on system variables, which are called *shell variables*. Here, **PS1** is a shell variable that directs Linux shell to display its content as the (primary) prompt. There are two types of shell variables, namely, global variables and local variables in **bash** and **csh**. A global variable is called an *environment variable*. An environment variable can be created via the **export** command in **bash**, converting a local variable into a global variable. Once created, the environment variable can be accessed and modified (using **=**) in the regular manner. You can also remove the environment variable by the **unexport** command. You would need to use the command **setenv** instead of **set** to modify an environment variable in **csh**.

The concept of environment variables is useful, since their contents can be accessed by a C or Java program to serve their purpose, e.g. knowing the context of program execution, e.g. who is running the program. Recall that we have made use of a special variable called **PATH** in **Lab 1**, by executing **export PATH="$PATH:."**. This command has the effect of inserting the current directory "." at the end of the current value stored in the variable **PATH**. Unix / Linux will then treat this as the search path for any command to be executed.

When you type **ls -l**, you will see the *permission mode* for your files, the first one indicates whether it is a directory (where **d** means a directory and **-** means a file, and it is also used to indicate special system-related privileges), the first part for yourself, the second part for users in the same group, and the third part for all other users. For each part, the privileges refer to *read*, *write* and *execute* respectively.

Note that the content of this paragraph is ***valid for standard*** Unix / Linux systems in many other institutes and for your personal web page (**/webhome/12345678d/public_html**), but not necessarily valid for the departmental file system. This standard access permission arrangement is also valid under **/tmp**, but it is more dangerous to try at **/tmp**, since everybody logging onto the same machine (**apollo** or **apollo2**) will be accessing and sharing the SAME **/tmp**. You might ignore this part if you just use the departmental file system. However, it is still useful to learn about the general Unix / Linux file system settings. In Unix / Linux systems, you better change the permission mode for your home directory from the default **drwx--x--x** mode (everyone can "execute" your directory, i.e., can access its content) or even worse **drwxr-xr-x** mode (everyone can see the files and access the content) to **drwx------** mode (no one else could access), with the change mode command **chmod 700 /webhome/12345678d**. Here the parameter for **chmod** is expressed in **Octal** (**base 8**), one octet for each part, so that **7** means **rwx** and **5** means **r-x**.

If you use the departmental file system, you will almost always see **drwx------** for any directory and **-rwx------** for any file. Only the write privilege can be set currently for the owner. Thus, performing **chmod 644** or **chmod 600** or **chmod 777** will **not** have created any change. If you perform **chmod 000**, you could get **-r-x------** for a file. Changes in local desktop files would not affect the file system.

The First Shell Script

Try to run the following first shell script program in **bash**, call it **first.sh**. Just type the script name to execute it, after setting it to be executable:

```
chmod 700 first.sh
```

Note that you may need to make some changes to the program text before it can be executed, if you merely copy it from the pdf file. Watch out for difference in *quotation symbols* and for **^M** characters when you make the copy, before you try to run it. This normally happens when files are moved across from Windows system to Unix / Linux systems. Each text line in a Windows file ends with **^M** and **^L** (called **<CR><LF>** or **Carriage-Return** and **LineFeed**), but the line in a Unix / Linux file ends with just **^L** (i.e. **<LF>**). So there is an extra **^M** in the view of Unix / Linux when a file created under Windows is moved or copied over. This is one major reason that a seemingly correct shell script program cannot run properly on the actual machine from our past experience. You can use the command **dos2unix** available on most Unix / Linux systems to perform the removal of those **^M** characters for file **first.sh**:

```
dos2unix first.sh
```

The **dos2unix** command is not available in Mac OS/X, but you can still do the same trick using the following approach, by creating a new file called **first2.sh**:

```
tr –d '\r' < first.sh > first2.sh
```

```
#!/bin/bash
echo Hello $USER, time is $(date)
# $(date) allows the COMMAND/PROGRAM date to be executed and replacing the output
echo -n "You must be "
# -n keeps output on the same line and we need a space at the end
whoami
# whoami is a command/program
echo Your home is $HOME
# HOME is a variable
echo Your calendar for this month is:
cal # cal is a program
echo Friends on this computer $(hostname) are:
who # who and hostname are programs
echo "Thanks for coming. See you soon!!"
```

Basic Shell Script Programming

Try out the shell script programs covered in the lecture and follow through the statements / commands to understand how they work. Try the programs using different inputs. Note that the symbol ***** is a *wildcard character* that leads to the matching of filenames, etc.

```
#!/bin/bash
# There is a bug here, could you debug it?
read -p "What was your score? " score
echo You get a score of $score
# [ ... ] is for testing condition on an expression
# use gt > / ge >= / lt < / le <= / eq = / ne != for comparison, use –a and –o for and/or
if [ $score -ge 85 -a $score -le 100 ]; then
   echo You got an A grade!
else if [ $score -ge 70 ]; then
      echo You got a B grade
   else
      echo You better study!!
   fi
fi
# a second way to express this
# use && and || to connect multiple testing expressions
if [ $score -ge 85 ] && [ $score -le 100 ]; then
   echo You got an A grade!
elif [ $score -ge 70 ]; then
# elif is like else if, but using else if needs one more matching fi at the end
   echo You got a B grade
```

```
else
   echo You better study!!
fi
```

```
#!/bin/bash
# Try also using the notion of ;;& instead of ;; in some of them (works on apollo/apollo2)
read -p "Which subject have you performed worst? " code
case $code in
   comp1*)     echo "$code is just a level 1 subject." ;;
   comp2*)     echo "$code is an intermediate level 2 subject." ;;
   comp[34]*) echo "$code is a challenging senior level subject." ;;
   comp[56]*) echo "$code is too advanced for you." ;;
   comp*)      echo "$code is unrecognized in comp." ;;
   *)          echo "$code is not a valid code." ;;
esac
```

```
#!/bin/bash
# Is there any potential bug in the first for loop?
courselist=(1011 2021 2432)
for course in ${courselist[*]}; do  # list from an array
   if [ $course -lt 2000 ]; then
      echo level 1 course $course
   else echo level 2 course $course
   fi
done
for file in `ls`; do          # list from command ls
   echo file $file exists
done
```

```
#!/bin/bash
# try list expanded from wildcard match (all C programs)
for f in *.c; do              # simple list
   echo $f is a C file
done
# try list expanded from a range
for i in {1..10}; do          # list of numbers
   echo i is $i
done
# try list expanded from a range of non-consecutive numbers
for i in {1..20..3}; do       # list of nonconsecutive numbers
   echo i is $i
done
# try a C-style loop, need to use the syntax of (( ... ))
for ((i=1; i<=10; i++)); do # like C for loop
   echo number is $i
done
# try handling input argument list
for item; do                  # absence of list implies the input argument list
   echo one of the arguments is $item
done
```

```
#!/bin/bash
# Could you convert it to compute the factorial?
read -p "Please enter a number: " num
ctr=1 sum=0
while [ $ctr -le $num ]; do
   let sum=sum+ctr            # same as sum=$((sum + ctr))
   echo $ctr
   let ctr++                  # same as ctr=$((ctr+1))
done
echo "The sum of $num numbers is $sum."
```

```
#!/bin/bash
# Compare with the program based on while loop
read -p "Please enter a number: " num
ctr=1 sum=0
until [ $ctr -gt $num ]; do
   let sum=sum+ctr           # same as sum=$((sum + ctr))
   echo $ctr
   let ctr++                 # same as ctr=$((ctr+1))
done
echo "The sum of $num numbers is $sum."
```

```
#!/bin/bash
# Note that this does not work for some of you due to the current file system in PolyU.
# You may do this under your own /webhome/12345678d,
# or make use of /tmp, which would still work under apollo or apollo2.
# Note that /tmp is shared between everyone using the same machine,
# so please create your own subdirectory for the experiment:
# $ cd /tmp
# $ mkdir myid12345678d
# $ cd myid12345678d
# Copy some files to here and then try
for f in *; do               # for each file in the current directory, i.e. "*"
   if [ -d $f ]; then
      echo $f is a directory
   fi
   if [ -f $f -a -x $f ]; then
      echo $f is a plain executable file
   fi
   if [ ! -w $f ]; then
      echo $f could not be modified
   fi
done
```

```
#!/bin/bash
# Call this script testarg.sh and try out different situations
echo "This script is called $0"
if [ $# -eq 0 ]; then
   echo "No argument"
else
   echo "The number of arguments : $#"
   echo "The first argument : $1"
   echo "The last argument : ${!#}" # this returns the last argument
   echo "The full list : $@"
   echo "The fifth argument : $5"   # how would you check that this fifth argument exists?
fi
```

Processing Exit Status (Optional)

After a command or program terminates, it would return an *exit status* back to the shell or the parent process. The exit status is a number between **0** and **255**. By convention, an exit status of **0** means that the process execution is successful. If the exit status is non-zero, it means that the execution has failed in some way with an error, in much the same way that **404** means **Page Not Found** in **http** access.

There is a special shell variable **$?** in **bash** that will store the exit status of the command or process just executed. After a command is executed, this special status variable **$?** is set to the exit status of the command. Try the following command execution, and observe the outputs. However, note that the exit status for a chain of commands with I/O redirection or pipe could be hard to interpret.

```
$ ps
$ echo $?
0
$ abc
bash: abc: Command not found...
$ echo $?
127
$ echo $?
0
```

It is clear that the first command **ps** is normally executed successfully, returning a status of **0**. The second command is normally not a command in Linux, unless you have created an executable file with the name **abc**. The error is **Command not found** and the exit status is **127**. It is interesting to see that the third execution of **echo** to check the status variable **$?** will give you **0** for the unknown command **abc**. This is because the second checking on the status variable **$?** is not checking for the status of execution of the command **abc**, but the execution of the previous command, i.e. the second **echo** in the example above. Therefore, the status is **0**, since the second **echo** command is executed correctly, outputting **127**.

Note that if you do have a program with the name **abc** and if the program executes correctly, the exit status would be **0** for your case above. However, if your program **abc** does not execute correctly, there would be a non-zero status, depending on the nature of the error and the arrangement of your program. For segmentation fault error, the exit status is **139**. An arithmetic error gives an exit status of **136**. Try the following program **lab3A.c** with inputs of 2 and 0. There will be errors in both executions. Try to check the exit status with **echo $?** to verify the error code resulted due to bad inputs.

```
// lab 3A
#include <stdio.h>

int main() {
  int n, m, ans, *ans2;

  m = 5;
  printf("Please enter a number\n");
  scanf("%d",&n); // try non-zero and zero
  ans = m / n;
  printf("Dividing %d by %d yields %d\n",m,n,ans);
  *ans2 = m / n; // why is there a problem here?
  printf("Dividing %d by %d yields %d\n",m,n,*ans2);
  printf("Program terminates\n");
}
```

You have the freedom to decide on the *exit status* of a program. If you consider that your program does not terminate successfully, you could return some specific number. The exit status of a process is determined by the argument inside the **exit**() system call. It is an integer between **0** and **255**, but some programmers also choose to return -1 inside the **exit**() system call. A return value of **-1** from the **exit**() system call is equivalent to a value of **255**. This is because the exit status is just a byte (unsigned). Thus, an exit status of **256** is the same as **0**, i.e. correct execution. Try **lab3B.c** below with different inputs and check the exit status.

```
// lab 3B
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) { // isprime.c implementation
  int n, prime, i;

  n = atoi(argv[1]);
  printf("Your number is %d\n",n); // should comment out for usage in key.sh
  prime = 1;
  for (i = 2; i <= n/2; i++)
     if (n % i == 0) prime = 0;
  if (prime) {
     printf("Your number %d is prime\n",n); // should comment out for usage in key.sh
     exit(0);
  } else {
     printf("Your number %d is not prime\n",n); // should comment out for usage in key.sh
     exit(1);
  }
}
```

The status variable **$?** is often used in a script program to test for the status of the previous command. It could perform some recovery or remedial actions in case of problems. Here is an example to check for the successful execution of a *file copy* command **cp**. This is quite common in a file backup procedure. If the file copy action is not successful, perhaps due to the lack of storage, the files would need to be copied to somewhere else for backup, or the storage needs to be cleaned up before retrying with the copying. Try out the following program under the four different scenarios and observe the output. Distinguish the error messages produced by the shell and those produced by the script program.

```
#!/bin/bash
# Call this script filecp and try out different situations
# 1. filecp dummy lab3Z.c          # assume that you do not have files dummy and lab3Z.c
# 2. filecp lab2A.c dummy          # assume that your lab2A.c is still there
# 3. filecp lab2A.c dummy          # copy again
# 4. filecp lab2A.c dummy          # copy with  dummy not modifiable (chmod 400 dummy)
cp $1 $2
if [ $? -eq 0 ]; then
   echo "File copied successfully"
else
   echo "Fail to copy file"
fi
```

A script program is also often used to *connect* several programs together to achieve something more serious or complicated. For example, we could have a program to test for whether a number is a prime number, and if we get two prime numbers and a chosen public key, use them to generate the private key and modulo for encryption. A typical script program to take in two possibly prime numbers, a public key, an existing input file and an output file name to generate the private key and encrypt might look like this. However, you need to create your own programs to perform **isprime** testing, **genkey** for key generation, and **toencrypt** for encrypting the content from the input file to the output file in order to actually run the script.

```
#!/bin/bash
# Call this key.sh, rename lab3B executable as isprime
# Develop your genkey and toencrypt programs (in C or script or others)
# Usage: key.sh 131 251 7 input.txt output.txt
./isprime $1 # derived from lab3B.c
if [ $? -ne 0 ]; then
   echo "First number is not prime"
   exit 1
fi
./isprime $2
if [ $? -ne 0 ]; then
   echo "Second number is not prime"
   exit 2
fi
ans=(`./genkey $1 $2 $3`) # genkey computes N, D from 2 primes and E, D=0 if E is bad
N=${ans[0]} D=${ans[1]}
if [ $D -eq 0 ]; then
```

```
   echo "Your chosen public key $3 is bad"
   exit 3
fi
echo "Your private key is $D with modulo $N"
if [ ! -f $4 ]; then
   echo "File $4 does not exist, will stop"
   exit 4
fi
if [ -f $5 ]; then
   echo "File $5 exist, will stop"
   exit 5
fi
./toencrypt $N $3 < $4 > $5 # toencrypt encrypts file using key E with modulo N
echo Encrypted $4 into $5
```

In shell scripts, the exit status from commands could be exploited when executing a chain of commands to make the script more efficient. Commands in a command list in a script are executed in a *short-circuit* manner. This brings in efficiency in programming and execution, but also brings in confusion and could be error-prone for ordinary programmers. Short-circuit execution means that the execution of a combined conditional statement will not be executed to the end, but will stop once the truth value, i.e. **TRUE** or **FALSE** can be determined upon partial execution.

- **`first-command && second-command`**
  Here, the second command is not executed if the first command does not execute correctly (i.e., the return status is non-zero). The successful execution of the second command does not change the fact that this conditional evaluates to **FALSE**.
- **`first-command || second-command`**
  Here, the second command is not executed if the first command executes correctly (i.e., the return status is zero). The potential unsuccessful execution of the second command does not change the fact that this conditional evaluates to **TRUE**.

You must be careful and *beware* of the difference of a success in exit status (0 = **success**, thus 0 || 1 = 0 meaning **success**) and a **TRUE** in normal expressions in if-conditionals (non-zero = **true**, thus 1 || 0 = 1 meaning **true**) when evaluated. This is opposite to normal programming logic, where 0 means **FALSE**.

Similar execution strategies can be embedded into the if-conditionals, with the **`[[ ... ]]`** construct for testing partial condition execution (beware that 0 means success for command exit status).

- **`if [[ first-command && second-command ]]; then`**
- **`if [[ first-command || second-command ]]; then`**

Try the following program with different scenarios. Observe the outputs.

```
#!/bin/bash
# Call this script chain.sh and try out different situations
# Example: chain.sh whoami pwd
# Example: chain.sh whoami abc
# Example: chain.sh abc whoami
# Example: chain.sh abc def
# where abc and def are not commands nor your executable files
echo first command is $1 and second command is $2
echo "executing $1 && $2"
($1) && ($2)
echo "executing $1 || $2"
($1) || ($2)
# Executing command chain inside a testing condition
if [[ `$1` && `$2` ]]; then
   echo "both are successful"
else
   echo "some are not successful"
fi
if [[ `$1` || `$2` ]]; then
   echo "some are successful"
else
   echo "both are not successful"
fi
```

Publishing Web Pages via Unix / Linux

[Optional]
In Department of Computing, personal web pages accessible to the public are stored in Unix/Linux file system. They must be placed under a special directory called **`public_html`** under your web home directory (under your home directory in other normal Unix file systems). Directories must be at least *executable* and files must be at least *readable* to other users. The default file accessed by web browsers is **`index.html`**. Once you have created a set of web pages using Microsoft Expression Web, Dreamweaver or any other appropriate tool, you may try to publish them to the public. You have to upload those related files from PC/Mac to Unix/Linux, perhaps using sftp / WinSCP or move them via J: drive. The web page files must be stored under **`/webhome/12345678d/public_html`**. We must do some preparation work to set the appropriate protection mode. ***Note***: The preparation work only needs to be done ***once*** in Unix/Linux. You may or may not need to set protection mode for your own directory in PolyU system, but that is the default approach in most other Unix/Linux systems.

- Change the protection mode of your own directory to **`drwx--x--x`** (i.e., 711) by typing
  `chmod 711 /webhome/12345678d`
- Create a sub-directory called **`public_html`** by typing
  `mkdir /webhome/12345678d/public_html`
- Change the protection mode of the directory to **`drwx--x--x`** (i.e., 711) by typing
  `chmod 711 /webhome/12345678d/public_html`

Now, invoke sftp or WinSCP to *upload* to Unix/Linux or *copy* the files to J: for your home. You can then copy or move to **`/webhome/12345678d/public_html`** yourself on apollo or apollo2. Alternatively, you can use WinSCP to directly copy all the set of files related to your web to **`/webhome/12345678d/public_html`**

The files are now placed in Unix/Linux. There is one major subtle difference between PC and Unix/Linux. In Unix/Linux, all web pages have file names ending with **`.html`** but in PC, all web pages have file names ending with **`.htm`**. Depending on the web server, you may need to rename the file(s), e.g..

```
mv index.htm index.html
```

The protection mode for the first default web page should be set *readable* to the public, i.e..

```
chmod 644 /webhome/12345678d/public_html/index.html
```

Now, you can explore your web pages using any browser. Your own homepage can be viewed at

`https://web.comp.polyu.edu.hk/12345678d`

where **`12345678d`** should be replaced by your own username. You may need to make some adjustments to the contents of those web page files if you discover some minor problems, such as path names. Also, you need to make sure that files for web pages are *readable* (**`chmod 644`**) and directories are at least *executable* (**`chmod 711`**). It is now that the pico/nano editor becomes useful when you need to touch up in the file contents. However, you need to work directly with the "terrible" HTML formats. To simplify, you may use **`zip`** or similar programs to archive all the necessary files created with web tools into a *zip file* and place it under Unix/Linux. For your information, Unix/Linux people often use the **`tar`** command to archive files. Finally, you can extract the files from the zip file and *set the appropriate protection mode* for files and directories.

Laboratory Exercise

Script programs are useful for text processing involving many files. For instance, each subject may be associated with a file containing the students taking the subject and perhaps with assessment grades. Given a collection of such subject files, it is not difficult to find out the academic performance of a particular student and generate a transcript for that student.

*Write a **bash** shell script* called **transcript** to search through a list of subject files and generate the transcript for one or more given students in the argument list. The first part of the argument list indicates the subject files to be used. This is followed by a special marker "**`student`**". Then a list of student ID follows. Each subject file starts with a line of 4 entries: the word "**`Subject`**", the subject code, the academic year and the semester that the subject is/was offered. Each of the subsequent lines contains the ID of a student and the grade. For subjects in the current semester, there is no grade for the students. To facilitate the checking of students, there is a special file called **`student.dat`**. Each line of this file contains the ID of a student and the name. For simplicity, there are only two semesters in each academic year. You can assume that there is no error in the information contained in all the files.

For instance, the program can be run like:

```
transcript COMP101121S2.dat COMP241122S1.dat student 1234
```

Here, **`COMP101121S2.dat`** and **`COMP241122S1.dat`** are the names of subject files, and **`1234`** is the ID of the student whose transcript is to be generated. The two lists are separated by the special marker "**`student`**". The use of wildcard character "**`*`**" can also be used in the list of subject files. However, it would ***not*** be used in the list of student ID. Typical file contents are shown below:

Filename	Content
`student.dat`	1223 bob 1224 kevin 1225 stuart 1226 otto 1234 john 1235 mary 1236 peter 1237 david 1238 alice
`COMP101121S2.dat`	Subject COMP1011 2021 2 1223 F 1234 B 1235 B+ 1236 A
`COMP101122S1.dat`	Subject COMP1011 2022 1 1223 B 1224 B+ 1225 B- 1238 C+
`COMP241122S1.dat`	Subject COMP2411 2022 1 1223 C+ 1234 B- 1235 B 1236 A
`COMP243222S2.dat`	Subject COMP2432 2022 2 1223 1235 1236 1237

Sample outputs (*formatting is not important*):

```
transcript COMP101121S2.dat COMP241122S1.dat student 1234
```

```
Transcript for 1234 john
COMP1011 2021 Sem 2 B
COMP2411 2022 Sem 1 B-
GPA for 2 subjects 2.85
```

```
transcript COMP* student 1236 1234 1223
```

```
Transcript for 1236 peter
COMP1011 2021 Sem 2 A
COMP2411 2022 Sem 1 A
COMP2432 2022 Sem 2
GPA for 2 subjects 4.00

Transcript for 1234 john
COMP1011 2021 Sem 2 B
COMP2411 2022 Sem 1 B-
GPA for 2 subjects 2.85

Transcript for 1223 bob
COMP1011 2021 Sem 2 F
COMP1011 2022 Sem 1 B
COMP2411 2022 Sem 1 C+
COMP2432 2022 Sem 2
GPA for 2 subjects 2.65
```

Remember that your program is supposed to work on *any correct datasets*. As a result, you ***should not*** hardcode the subject file names in your script program. It is ***not required*** that you use the real data set for program development and testing. It would be easier if you use a small dataset for development and initial program testing.

Requirements:

1. Elementary level: your script can read in the subject file data and echo the relevant information correctly.

2. Basic level: your script can produce the correct set of subjects and the associated information for one student. Subject ordering is unimportant. GPA calculation is not required.

3. Required level: your script can produce the correct set of subjects and the associated information for all students, with correct GPA calculation. If a student takes the same subject more than once, only the grade for the latest taking will be included in GPA calculation. If all subjects have no grade (i.e., current subjects), the GPA would be *blank* or *unknown*. You may round the GPA to two decimal places or truncate it at two decimal places, i.e., GPA for 7.7/3=2.566 can be displayed as either 2.57 or 2.56.

4. Bonus level: your script can accept the name of students besides the student ID and can check for potential students with the same name but different ID. For example, peter is a very common name among the students and you should display the transcripts of all peters.

Please provide ***appropriate comments*** and check to *ensure* that your program can be ***executed*** properly under the Linux (**apollo** or **apollo2**) environment before submission.

Name your program **transcript** (note that you cannot submit a file with extension **.sh** in BlackBoard for security reason) and **submit it via BlackBoard on or before 24 February 2023**.

Hint: You can get the digits of the GPA with integer arithmetic through multiplying the numerator by 100. For example, GPA for john is 5.7/2 = 2.85 with floating point arithmetic, and the actual digits would be 570 / 2 = 285 using only integer arithmetic. The GPA can be displayed by printing the decimal point in the proper position, upon getting the first digit and the remaining digits via integer arithmetic.