CS2106 Introduction to Operating Systems
Lab 2 – Shell Scripting and Process Programming
Week of 7 February 2022

Introduction

In this lab we will look at shells and at creating and managing processes. A shell is a program that runs on top of the operating system, and its main task is to interface the user with the operating system (OS). The familiar graphical environment in Windows and MacOS are both examples of graphical shells; they allow a user to interact with the OS using graphical elements like menu bars, context menus, text boxes, clickable links and images, etc.

Bash – Bourne Again Shell – is an example of a command-line shell, just like zsh in MacOS. Users interact with the OS by typing in commands. In this lab we will explore two aspects of programming command-line shells. In the first section we will look at how to do shell script programming on Bash, while in the second section we will look at how to create processes, run programs, do input and output redirection and pipes in C.

Instructions

Some points to note about this lab:

a. This lab should be completed on xcne nodes. If you choose to complete this lab on your laptop, etc., ensure that the codes run on xcne or they will be marked wrong.
b. You may complete this lab on your own, or with ONE partner from any lab group. You do not need to partner with the same person with whom you completed Lab 1.
c. There are two deliverables for this lab; a zip file that you can submit with your partner or individually IF you are doing the lab alone, and a practical demo that must be done individually.
d. Only ONE copy of the submission zip file is to be submitted. Therefore, if you do the report with a partner, decide who should submit the report. DO NOT submit two copies.
e. Use the enclosed AxxxxxxY.docx answer book for your report, renaming it to the student ID of the submitter.
f. Please indicate the student number, name, and group number of each person in the lab report.
g. Create a ZIP file called AxxxxxY.zip with the following files (rename AxxxxxY to the student ID of the submitter):
   - The AxxxxxY.docx, appropriately renamed.
   - The grade.sh file from part b of Part 1.
   - The lab2p2f.c file from part b of Part 2.
h. Upload your ZIP file to the Files->Lab Submissions->Lab 2 Submissions folder on LumiNUS by **SUNDAY**, **20 February 2022, 11.59 pm.**
i. Some extra time is given for submission, but once the folder closes, **NO SUBMISSIONS WILL BE ENTERTAINED AND YOU WILL RECEIVE 0 MARKS FOR THE LAB.**

Part 1 – Bash Scripting

Bash scripting is an essential skill for anyone who works on servers running *nix operating systems like Linux. It can automate many tasks. For example you can write a Bash script to automatically compile your code, run unit tests if the compilation succeeds, and then push the code to a Github repository if the unit tests pass, while capturing the outputs of each stage to a file for later review.

Switch to the "part1" directory. This section introduces just the basics of shell scripting. For more details please see https://devhints.io/bash

a. Bash Script Basics

Before we start building a more interesting script, let's go through some basics.

i. Creating a Hello World Script

Log on to one of the xcne nodes, and use your favorite editor and create a file called "hello.sh". (the extension "sh" is conventional but unnecessary – you could have equally called it "hello.myhighfalutinshellscript" if you wanted. But please don't.). Enter the following into "hello.sh":

```
#!/bin/bash
echo "Hello world!" # Echo is similar to printf in C.
```

Question 1.1 (1 mark)

Ordinarily, comments in Bash scripts start with #. So for example:

```
# This is a comment
ls -l      # This is also a comment
```

However the first line of the Bash script is NOT a comment:

```
#!/bin/bash
```

What is this line for?

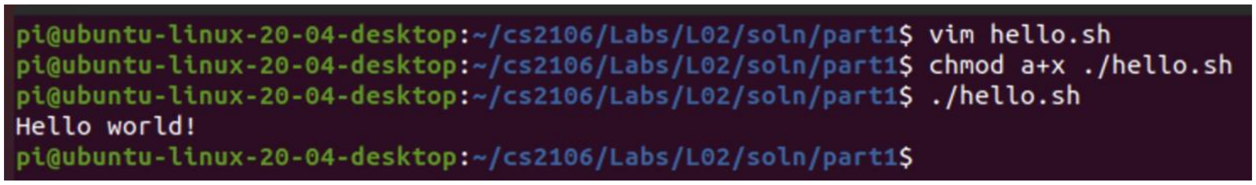
Now exit your editor, and on the Bash command line, we convert it to an executable file by executing the following command:

```
chmod a+x ./hello.sh
```

This command sets the "executable" flag on hello.sh for "all" using the parameter "a+x", thus everyone can execute lab2p1a.sh. To do so:

```
        ./hello.sh
```

You will see the output below:

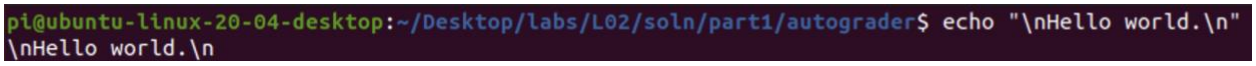


```
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ vim hello.sh
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ chmod a+x ./hello.sh
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ ./hello.sh
Hello world!
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$
```

Note: echo does not normally process '\n' or other slash escape sequences. For example, if you did:

```
        echo "\nHello world.\n"
```

You would get:

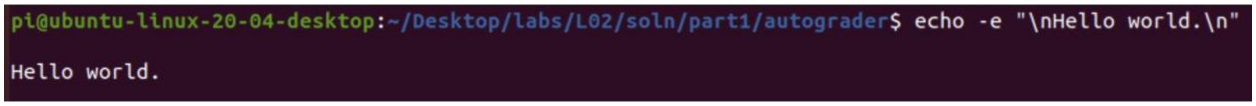


```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1/autograder$ echo "\nHello world.\n"
\nHello world.\n
```

This is probably not what we want. To process escape sequences, specify the "-e" option when calling echo. E.g.

```
        echo -e "\nHello world.\n"
```

You will now see that \n is properly processed:



```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1/autograder$ echo -e "\nHello world.\n"

Hello world.
```

ii. Variables

Variables are very useful in Bash scripts, and can be used to store strings, numeric values, and even the outputs of programs. Use your favorite editor and create a file called "diff.sh", with the following lines to subtract 20 from 15 giving -5.

```
        #!/bin/bash
        x=15
        y=20
        z=$x-$y
        echo "$x - $y= $z"
```

NOTE: In your assignment statements (e.g. x=5), it is VERY IMPORTANT that there is NO SPACE between the variable, the '=' and the value. Likewise in the line z=$x-$y, it is VERY IMPORTANT that there THERE ARE NO SPACES in the statement. If you have spaces you will get an error message like "x: command not found".

Use chmod to make this script executable, execute it and answer the following question:

Question 1.2 (1 mark)
The script produces the wrong result "15 – 20 = 15-20", instead of "15 – 20 = -5". Using Google or otherwise, fix the script so that it says "15 – 20 = -5".

Summarize in one line how you fixed the script.

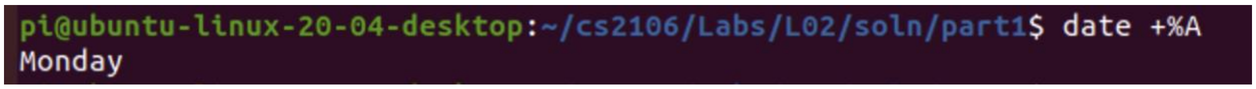
Notice some things about the script above:

(a) You assign to a variable using =, which is expected. However as mentioned earlier, there must not be any spaces in your assignment statement. For example, x=5 is correct, but x = 5 is wrong and will result in an error like "x: command not found"
(b) Use $<var name> to access the value stored in <var name>. For example, we used $x and $y to access the values stored in x and y.
(c) Notice that we can similarly access the values of the variables in the echo statement by using $.

Now let's look at how to capture the output of a program to a variable. On your xcne or other Linux shell session, type:

```
date +%A
```

This should print out the current day. For example:

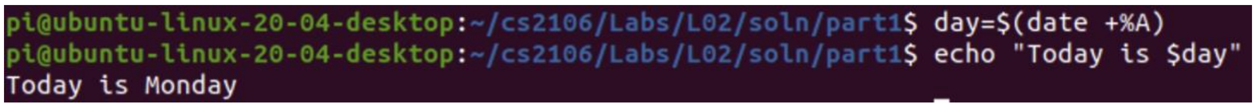


```
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ date +%A
Monday
```

To store this in a variable, we use the $(.) operator. Enter the following into your Bash shell (you do not need to write a script for this part):

```
day=$(date +%A)
echo "Today is %day."
```

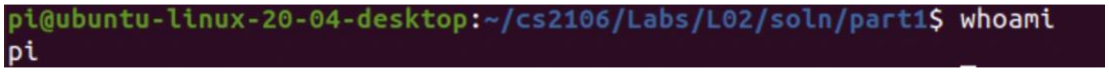
You will see (this lab sheet was written on a Monday):



```
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ day=$(date +%A)
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ echo "Today is $day"
Today is Monday
```

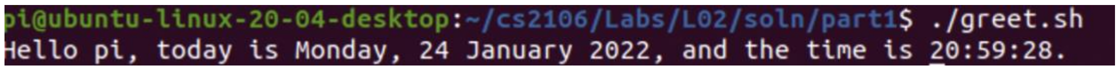
Question 1.3 (1 mark)

The "whoami" command returns the user ID of the current user:



```
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ whoami
pi
```

Use your favorite editor and create a shell script called "greet.sh" that greets the user, stating the current day of week, day, month, year and time in the following way:



```
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ ./greet.sh
Hello pi, today is Monday, 24 January 2022, and the time is 20:59:28.
```

Hint: You can type "man date" (without the quotes) in Bash to learn more about the date command, and how to extract the full name of the month, etc.

Cut and paste your code to your answer book.

Shortcut:

Try the following in Bash and see if it helps make your code shorter. ☺

```
echo "Hello $(whoami), today is $(date +%A)."
```

iii. Test Statements

Bash can test for certain conditions using the [[.]] operator. Some things you can do:

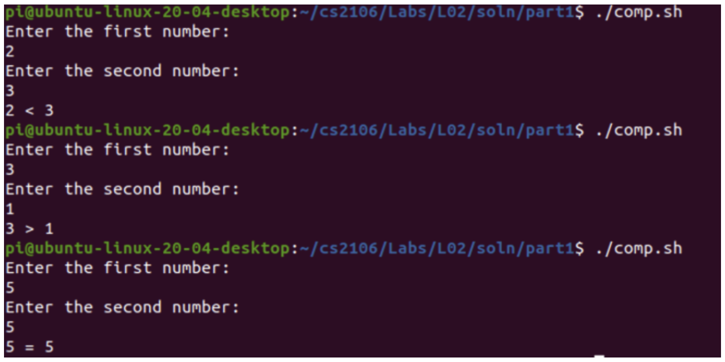
Test	**Result**
`[[ -z <string> ]]`	Tests if <string> is empty (i.e. "").
`[[ -n <string> ]]`	Tests if <string> is not empty.
`[[ <string1> == <string2> ]]`	Tests if <string1> is equal to <string 2>
`[[ <string1> != <string 2> ]]`	Tests if <string1> is not equal to <string 2>
`[[ num1 -eq num2 ]]`	(Numeric) Tests if num1 == num2
`[[ num1 -ne num2 ]]`	(Numeric) Tests if num1 != num2
`[[ num1 -lt num2 ]]`	(Numeric) Tests if num1 < num2
`[[ num1 -le num2 ]]`	(Numeric) Tests if num1 <= num2
`[[ num1 -gt num2 ]]`	(Numeric) Tests if num1 > num2
`[[ num1 -ge num2 ]]`	(Numeric) Tests if num1 >= num2
`[[ -e FILE ]]`	Tests if file exists
`[[ -d FILE ]]`	Tests if file is a directory
`[[ -s FILE ]]`	Tests if file size > 0
`[[ -x FILE ]]`	Tests if file is executable
`[[ FILE1 -nt FILE2 ]]`	Tests if FILE1 is new than FILE2
`[[ FILE1 -ot FILE2 ]]`	Tests if FILE1 is older than FILE2
`[[ FILE1 -ef FILE2 ]]`	Tests if FILE1 is the same as FILE2

Note the spaces after [[and before]]. They ARE important!

These tests can be use within if..elif..else..fi statements. Create a shell script called “comp.sh” and type in the following:

```
#!/bin/bash

echo "Enter the first number: "
read NUM1
echo "Enter the second number: "
read NUM2

if [[ NUM1 -eq NUM2 ]]; then
    echo "$NUM1 = $NUM2"
elif [[ NUM1 -gt NUM2 ]]; then
    echo "$NUM1 > $NUM2"
else
    echo "$NUM1 < $NUM2"
fi
```

Make comp.sh executable, and execute it. You can enter various numbers to play with it:



```
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ ./comp.sh
Enter the first number:
2
Enter the second number:
3
2 < 3
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ ./comp.sh
Enter the first number:
3
Enter the second number:
1
3 > 1
pi@ubuntu-linux-20-04-desktop:~/cs2106/Labs/L02/soln/part1$ ./comp.sh
Enter the first number:
5
Enter the second number:
5
5 = 5
```

Some things to note:

- You can read from the keyboard using “read”. The syntax is “read <varname>”, where <varname> is the variable that we want to store the read data to.

- The “if” statement has an odd syntax; you need a semi-colon after the [[..]]:

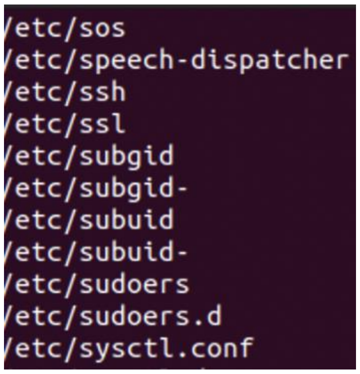
```
if [[ NUM1 -eq NUM2 ]]; then
      echo "$NUM1 = $NUM2"
else
      echo "$NUM1 != NUM2"
fi
```

iv. Loops

Bash supports both for-loops and while-loops. The for-loop is similar to Python’s. For example, to list all the files in the /etc directory, we could do:

```
for i in /etc/*; do
      echo $i
done
```

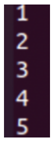
We get an output like this:



```
/etc/sos
/etc/speech-dispatcher
/etc/ssh
/etc/ssl
/etc/subgid
/etc/subgid-
/etc/subuid
/etc/subuid-
/etc/sudoers
/etc/sudoers.d
/etc/sysctl.conf
```

You can also iterate over a range:

```
for i in {1..5}; do
      echo $i
done
```

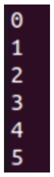


```
1
2
3
4
5
```

The while loop works pretty much the way you'd expect it to. For example:

```
i=0
while [[ $i -le 5 ]]; do
      echo $i
      let i=i+1
done;
```

We get:

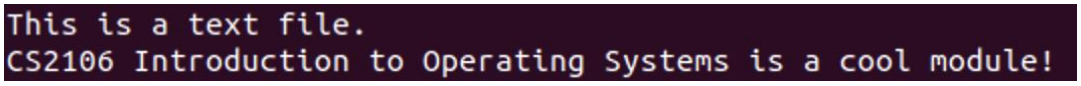


```
0
1
2
3
4
5
```

You can read text files using the while loop! To print file.txt (provided in your zip file in the part1 directory), you can do:

```
cat file.txt | while read line; do
      echo $line
done
```

The "cat" command prints the contents of file.txt to the screen, but the "|" hijacks this output and sends it to the "read" command using a mechanism known as "pipe". Recall that read line will read whatever is being piped in to the "line" variable.



```
This is a text file.
CS2106 Introduction to Operating Systems is a cool module!
```

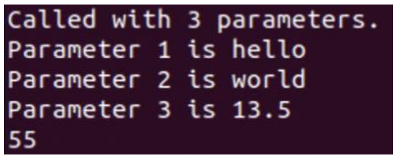
v. Functions

You can declare a function called "func" this way. Parameters are accessed using $1, $2, etc. for the first parameter, second parameter, etc. Create a file called "func.sh" and type in the following:

```
#!/bin/bash
function func {
      count=1
      echo Called with $# parameters.
      for i in $@; do
            echo Parameter $count is $i
            let count=count+1
      done
      return 55;
```

```
    }

    func hello world 13.5
    echo $?
```

Make this file executable, then execute it. You will see:



```
Called with 3 parameters.
Parameter 1 is hello
Parameter 2 is world
Parameter 3 is 13.5
55
```

Question 1.4 (1 mark)

The following are special variables in Bash. What do they hold?

$#, $1, $2, $@, $?

vi. Miscellaneous Topics

Let's look at a few additional features in Bash, which you may find useful.

a) Redirecting Output

You can redirect output to the screen (stdout) to a file. For example, if you wanted to capture the output of "ls" to a file:

```
ls > ls.out
```

Note that you can *append* to an output file by using >> instead of >. For example:

```
ls -l >> ls.out
```

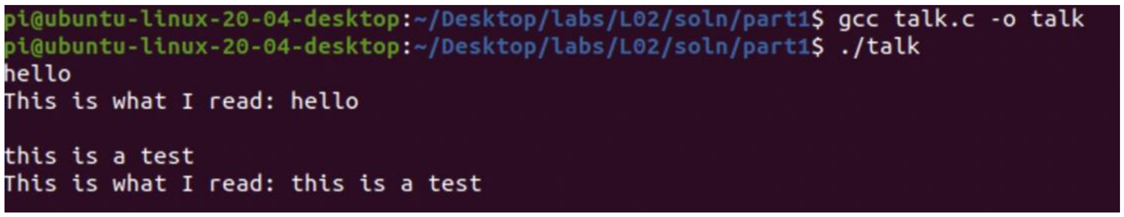
Would append output to ls.out without overwriting previous content.

b) Redirecting Input

You can also redirect input from the keyboard to a file. In the "part1" directory you have a file called "talk.c". Compile and run it using:

```
gcc talk.c -o talk
./talk
```

This program echoes back whatever you type on the keyboard, prefacing it with "This is what I read:"

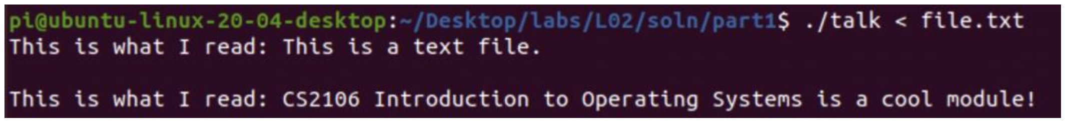


```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1$ gcc talk.c -o talk
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1$ ./talk
hello
This is what I read: hello

this is a test
This is what I read: this is a test
```

There is also a file called file.txt. Enter the following command:

```
./talk < file.txt
```

As you can see in the output, the contents of file.txt are printed out by "talk" as though they were typed in on the keyboard:



```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1$ ./talk < file.txt
This is what I read: This is a text file.

This is what I read: CS2106 Introduction to Operating Systems is a cool module!
```

c) Getting the Result Returned by a Program

You have a file called "slow.c" in the "part1" directory. It counts from some value *n* that you specify on the command line, to *n + 5*, once per second. The source code is simple and shown below:

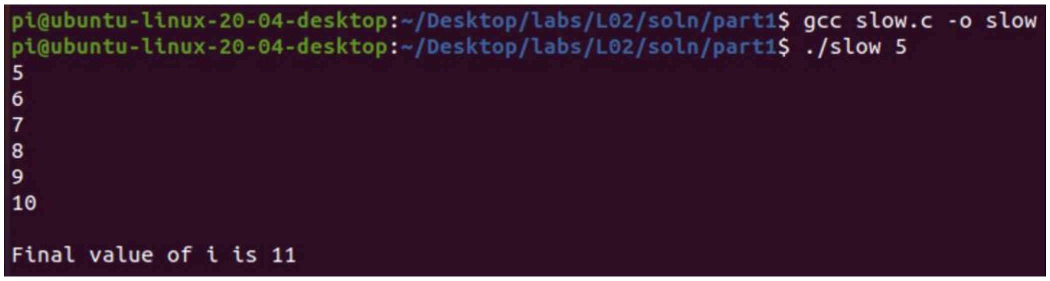
```
#include <stdio.h>

// For the sleep function
#include <unistd.h>

// For the atoi function that converts
// strings to integers
#include <stdlib.h>

int main(int ac, char **av) {
    if(ac != 2) {
        printf("\nCounts from specified integer n to n + 5\n");
        printf("Usage: %s <integer>\n\n", av[0]);
        exit(-1);
    }

    int n = atoi(av[1]);
    int i;
    for(i=n; i<=n+5; i++) {
        printf("%d\n", i);
        sleep(1);
    }

    printf("\nFinal value of i is %d\n\n", i);

    exit(i);
}
```

Notice in particular the final statement "exit(i);". This allows slow.c to return the final value of "i" to the operating system. In this section we will see how to retrieve this value.

Compile and run it using the following instructions:

```
gcc slow.c -o slow
./slow 5
```

You will see the following output:



```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1$ gcc slow.c -o slow
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1$ ./slow 5
5
6
7
8
9
10

Final value of i is 11
```

Now enter the following command:

```
echo $?
```

Question 1.5 (1 mark)

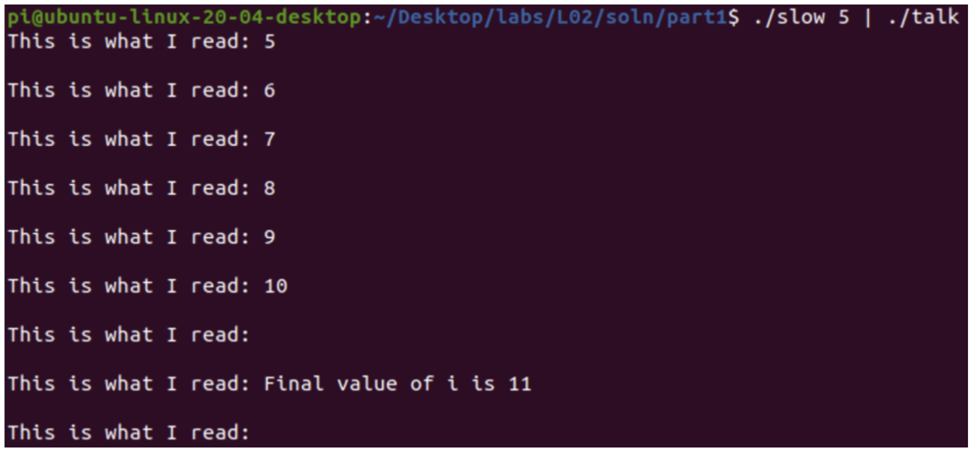
Earlier we mentioned the "exit(i);" statement in slow.c. What do you see when you do "echo $?" ? How is "echo $?" related to "exit(i);"?

d) Pipes

Assuming one program prints to the screen and another reads from the keyboard, you can channel the output of the first program to the input of the second using a mechanism called a "pipe". Earlier we saw "slow.c" and "talk.c". Enter the following command:

```
./slow 5 | ./talk
```

Your screen will appear to hang; wait for around 6-7 seconds and you will see:



```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1$ ./slow 5 | ./talk
This is what I read: 5

This is what I read: 6

This is what I read: 7

This is what I read: 8

This is what I read: 9

This is what I read: 10

This is what I read:

This is what I read: Final value of i is 11

This is what I read:
```

As you can see, the output of "slow" was channeled to the input of "talk". Pipes are very useful mechanisms and we will see more of it later.

e) Running Programs Sequentially and In Parallel

Enter the following command:

```
./slow 5 ; ./slow 10
```

If a command is terminated by the control operator &, the shell executes the command in the background in a subshell. The shell does not wait for the command to finish, and the return status is 0.

Now enter the following command:

```
./slow 5 & ./slow 10
```

Commands separated by a ; are executed sequentially; the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.

Question 1.6 (1 mark)

What happens when you run "./slow 5 ; ./slow 10", and when you run "./slow 5 & ./slow 10"? What is the difference between ";" and "&"? **man bash | grep -C2 '$@'**

b. Writing a Cool Script

We will now pull together everything you've learnt in this section to write a script to write a simple autograder.

In the part1/autograder directory you will find two sub-directories:

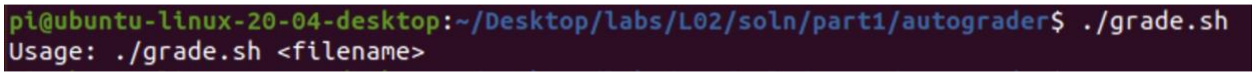
Sub-Directory	**Contents**
ref/	Contains "model answer" program in a set of C files, and a set of test files with a ".in" extension.
subs/	Student submissions, divided into directories.

A template shell script called "grade.sh" has already been created for you. This script is called with a single argument, which is the name of the program to be compiled and tested. E.g. to compile a program called "sum", you'd do:

```
./grade.sh sum
```

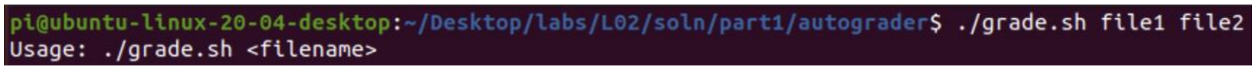
Complete the template file. This is what your completed shell script should do:

1) If no argument is supplied to the script, it should output:



```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1/autograder$ ./grade.sh
Usage: ./grade.sh <filename>
```

Also if more than one argument is supplied to the script, it should output:



```
pi@ubuntu-linux-20-04-desktop:~/Desktop/labs/L02/soln/part1/autograder$ ./grade.sh file1 file2
Usage: ./grade.sh <filename>
```

2) If exactly one argument has been supplied, then use gcc to compile the C source files in ref/ to the filename specified when invoking the script. So for example if we did:

```
./grade.sh fun
```

Your script should do:

```
gcc <C filenames here> -o fun
```

3) Delete all output reference files in ref/
4) Generate new output reference files, by running the program on each ".in" file. So for example if you have "f1.in" and "f1.in", your script should produce "f1.in.out" and "f2.in.out"   by running the equivalent of:

```
fun < f1.in > f1.in.out
fun < f2.in > f2.in.out
```

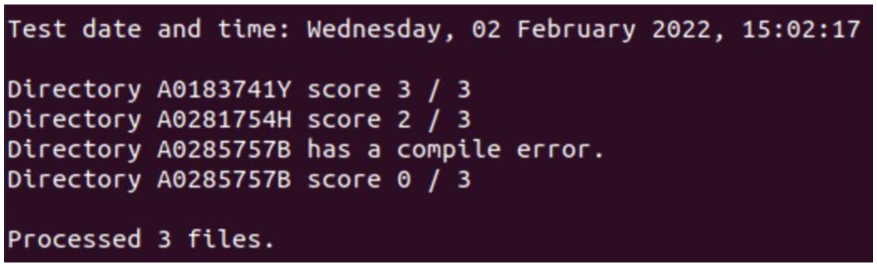
Your shell script SHOULD NOT hardcode the generation of the ".out" files, but must use a loop to iterate over every ".in" file to generate the corresponding ".out" file. While naming the output "f1.out" looks better than "f1.in.out", it's also harder so that's not a requirement. ;) You can also give your output reference files other names, e.g. f1.in.ref, f1.ref, etc.

You may assume that the program you are testing always takes input from the keyboard and outputs to the screen, so you can use ">", "<" and "|" if you wish.

5) Now iterate over every student directory in subs/:
   a. Compile the code in the student's directory.
   b. Output an error message to the "results.out" file (see later) if there is a compile error.
   c. Generate output files for each of the ".in" files in ref/.
   d. Compare the output files from the student's program with the output reference files generated in step 4.
   e. Award one point to the student for each output file is *exactly* identical to the corresponding output reference file.

f. Print out the points awarded to the student to “results.out”, upon the maximum points possible. (Hint: Since we award 1 point for each output file that matches the reference output file, if there are *n* such output reference files, than the maximum mark possible is *n.* This value should NOT be hard-coded.)
g. Print out the total number of student files marked in “results.out”.

6) Your “results.out” file should look like this, with the day, date, time scores and number of student files marked (“processed”):



```
Test date and time: Wednesday, 02 February 2022, 15:02:17

Directory A0183741Y score 3 / 3
Directory A0281754H score 2 / 3
Directory A0285757B has a compile error.
Directory A0285757B score 0 / 3

Processed 3 files.
```

DEMO (4 marks)

Run your shell script and show the output of results.out to your TA. Points will be deducted for incorrect scores or missing output.

Some hints:

1) You should assume that grade.sh will be run in the directory that contains both ref/ and subs/.
2) You can detect differences between the reference output file and the student’s output file by using “diff”. In particular, there is a way of using the return value of “diff” to check if the two files are identical. You can do “man diff” for details.
3) Similarly, there is a way to detect compile errors based on the value that gcc returns to the shell.

Part 2 – Playing with POSIX Process Calls

Change to the “part2” directory. In this section we will learn how to create processes, how to parallelize processes, and how to redirect input and output, and how to set up pipes between processes.

a. <u>Introduction to Process Management in C</u>

i. Creating a new Process

You can spawn a new process by using “fork”. Open the “lab2p2a.c” file with your favorite editor. Examine the code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int slow(char *name, int time, int n) {
    for(int i=n; i<=n+4; i++) {
        printf("%s: i = %d\n", name, i);
        sleep(time);
    }
}

int main() {
    int id;

    if((id = fork()) != 0) {
        int stat;
        int my_id = getpid();
        int parent_id = getppid();
        printf("\nI am the parent.\n");
        printf("My ID is %d\n", my_id);
        printf("My parent's ID is %d\n", parent_id);
        printf("My child's ID is %d\n\n", id);
        slow("Parent", 1, 5);
        printf("\nWaiting for child to exit.\n");
        wait(&stat);
        printf("CHILD HAS EXITED WITH STATUS %d\n", WEXITSTATUS(stat));
    }
    else
    {
        id = getpid();
        int parent_id = getppid();
        printf("\nI am the child.\n");
        printf("My ID is %d\n", id);
        printf("My parent's ID is %d\n\n", parent_id);
        slow("Child", 2, 10);
        exit(25);
    }
}
```

Some explanation of this code:

- We #include <stdlib.h> to bring in “exit”, which the child will use to return a value.
- We #include <unistd.h> to bring in the prototype for “fork”, used to spawn a new process, and #include <sys/wait.h> to bring in “wait”.

- The “slow” function counts from *n* to *n+4* inclusive, similar to the slow.c program we saw earlier, except that that one counted to *n+5*. The parent process prints a number every second and the child process prints a number every two seconds.
- The “fork” function call creates a new process:
  - “fork” returns 0 to the child process, and the child’s process ID (PID) – a non-zero value – to the parent.
  - A process can get its own PID using getpid(), and its parent’s PID using getppid().
- The child returns a value of 25 using “exit”.
- The parent calls “wait” to wait for the child to finish, stores the child’s return value into “stat”, then calls the WEXITSTATUS macro to extract the returned value from “stat”. You can find out more about the various exit status macros here: https://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html

Compile and execute the above code using:

```
gcc lab2p2a.c -o lab2p2a
./lab2p2a
```

Observe the results and answer the question below:

Question 2.1 (1 mark)

Are the parent and child processes executing in concurrently? How do you know this?

Question 2.2 (1 mark)

You can see that that the parent process also has a parent. Who is the parent’s parent? **Hint:** The “ps” command shows the processes running in the current shell together with their PIDs. Execute the following two commands

```
./lab2p2a
ps
```

ii. Accessing Arguments and Environment Variables in C

Use your favorite editor to open “lab2p2b.c”. Examine the code:

```
#include <stdio.h>

int main(int ac, char **av, char **vp) {
    printf("ac = %d\n", ac);
    printf("Arguments:\n");

    int i;

    for(i=0; i<ac; i++)
        printf("Arg %d is %s\n", i, av[i]);

    i=0;
    while(vp[i] != NULL) {
        printf("Env %d is %s\n", i, vp[i]);
        i++;
    }
}
```

Compile and execute the program using the following commands, observe the outputs and answer the questions that follow:

```
gcc lab2p2b.c -o lab2p2b
./lab2p2b
./lab2p2b hello world
./lab2p2b this is a test
```

Question 2.3 (1 mark)

What do “ac”, “av” and “vp” contain?

iii. <u>Loading and Executing A Program</u>

In C we can load and execute a program using the “exec*” family of system calls. The table below shows the different versions. You must #include <unistd.h> to use these functions.

Version	**What it Does**
`execl(const char *path, const char *arg1, char *arg2, const char arg3, …)`	Executes command shown in “path”. Arguments to command are listed individually, terminated with a NULL. E.g. `execl(“/bin/ls”, “ls”, “-l”, NULL);` Note you must specify the full path to “ls”. Also conventionally the first argument is always the name of the program you are running.
`execlp(const char *file, const char *arg1, char *arg2, const char arg3, …)`	Like execl, except that if you do not specify the full path to the command to run, the OS will search for the command in all directories specified in the PATH environment variable. E.g. `execlp(“ls”, “ls”, “-l”, NULL);`

`execv(const char *path, char *const argv[]);`	Like execl, except that the arguments are specified in an array instead of individually. The last element of the array must be NULL:
	
	E.g.
	
	`char *args[] = {“ls”, “-l”, NULL};`
	`execv(“/bin/ls”, args);`

`execvp(const char *file, char *const argv[])`	Like execv, except that if you do not specify the full path to the command ot run, the OS will search for the command in all directories specified in the PATH environment variable.
	
	E.g.
	
	`char *args[] = {“ls”, “-l”, NULL};`
	`execvp(“ls”, args);`

There are also execle and execve function calls which pass in environment variables, and we will ignore these here.

Since all the exec* functions replace the current process image with the process image of the program being run, and is conventionally run within a fork(). Again use your favorite editor to open the "lab2p2c.c" file, and examine the code:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    if(fork() ==  0) {
        execlp("cat", "cat", "file.txt", NULL);
    }
    else
        wait(NULL);
}
```

Question 2.4 (1 mark)

Change the code to use execvp instead of execlp. Cut and paste your new code here and explain what you've done.

iv. Redirecting Input and Output

In the previous section we saw how we can redirect input and output using "<" and ">" respectively. Now we will see how to do this programmatically:

(a) You will see a "talk.c" program in the "part2" directory. This is exactly the same as the "talk.c" program in part 1. Compile it, naming the output executable "talk".

(b) Open lab2p2d.c, and examine the code:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int fp_in = open("./file.txt", O_RDONLY);
    int fp_out = open("./talk.out", O_CREAT | O_WRONLY);

    if(fork() == 0) {
        dup2(fp_in, STDIN_FILENO);
        dup2(fp_out, STDOUT_FILENO);
        execlp("./talk", "talk", (char *) 0);
        close(fp_in);
        close(fp_out);
    }
    else
        wait(NULL);

}
```

A few things to note:

- We are going to use library calls to execute the equivalent of:

  ```
  ./talk < file.txt > talk.out
  ```

- We are using the more primitive "open" and "close" operations to operations to open file.txt for reading (O_RDONLY). We also create a new file called talk.out for writing (O_CREAT | O_WRONLY). We use these instead of fopen and fclose and open and close will create file descriptors in the form that we need here.

Compile and run the program using:

```
gcc lab2p2d.c -o lab2p2d
./lab2p2d
cat talk.out
```

Notice that our program has done exactly ./talk < file.txt > talk.out as mentioned earlier.

Question 2.5 (1 mark)

What does “dup2” do? Why do we use “dup2” here?

v. Pipes

A “pipe” is a byte-oriented communication mechanism between two processes using two file handles; the first is for reading, and the second is for writing.

Open the lab2p2e.c file and you will see the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

int main() {
    int p[2];
    char str[] = "Hello this is the parent.";

    // This creates a pipe. p[0] is the reading end,
    // p[1] is the writing end.

    if(pipe(p) < 0)
        perror("lab2p2e: ");

    // We will send a message from father to child
    if(fork() != 0) {
        close(p[0]); // The the end we are not using.
        write(p[1], str, strlen(str));
        close(p[1]);
        wait(NULL);
    }
    else
    {
        char buffer[128];

        close(p[1]); // Close the writing end
        read(p[0], buffer, 127);
        printf("Child got the message \"%s\"\n", buffer);
        close(p[0]);
    }
}
```

We note some points of this code:

- Most POSIX calls will return a value of less than 0 if there's been an error. In this code we check for errors for the first time in these lines:

```
if(pipe(p) < 0)
        perror("lab2p2e: ");
```

  The "perror" call prints the cause of the error to the screen, preceding it with the string "lab2p2e:".

- The "pipe" call takes in an integer array p of two elements. The p[0] element is the reading end of the pipe, and the p[1] element is the writing end of the pipe.

- Just as we used "open" and "close" instead of "fopen" and "fclose" in the previous example, here we again use the more primitive "read" and "write" operations. You can see from the code how these are used.

- We always close the end we are not using. For example, the parent is not using the reading end and thus closes it, and likewise with the child.

Run the code by doing:

```
gcc lab2p2e.c -o lab2p2e
./lab2p2e
```

Observe that the child has receive the parent's message.

Question 2.6 (1 mark)

Why do we always close the end of the pipe we are not using?

b. Piping Between Commands

We will now attempt something more challenging; we will write a program that pipes the output of one program to the input of another.

(a) Compile the talk.c program if it's not already compiled. Name the executable file "talk".
(b) Compile the slow.c program, also in the part2 directory. Name the executable file "slow".
(c) Open the file "lab2p2f.c", and write the code that does the equivalent of the following using C pipes, redirection, fork and exec* as you've learnt before.

```
./slow 5 | ./talk > results.out
```

Some hints:

- ./slow will take about 7 seconds to run, so your computer will appear frozen for that time even when it is correctly done.
- You can execute "`./slow 5 | ./talk`", then do:

  ```
  gcc lab2p2f.c -o lab2p2f
  ./lab2p2f
  cat results.out
  ```

  You will see an output similar what what you saw above.

- The essential thing is that you need to run ./slow and ./talk as two processes, then somehow redirect the output of one program to the writing end of the pipe, and the input of the other program to the reading end of the pipe.

- Don't forget to redirect the final result to "results.out".

Question 2.7 (1 mark)

Explain briefly how you set up the pipe between ./slow and ./talk