**CS2106 Introduction to Operating Systems**

**Lab 2 - Shell Scripting and Process Programming**

**Answer Book**

Please read the instructions in the main lab sheet before completing this document. Submission deadline is **Sunday 16 March 2025, 11.59 pm**. The folder will stay open slightly after this, but once the folder closes, **absolutely no submissions will be allowed.**

**Submission checklist:** A ZIP file called AxxxxxxY.zip, where AxxxxxxY is the student ID of the student submitting. The ZIP file should contain:

* This file, appropriately renamed to the submitter’s student ID.
* grade.sh
* lab2p2f.c

|  |  |
| --- | --- |
| **Student 1** | |
| Name: | Ling Guan Yu |
| Student ID (AxxxxxxY): | A0308292A |
| Group (Bxx): | B20 |
| **Student 2** | |
| Name: | NIL |
| Student ID (AxxxxxxY): | NIL |
| Group (Bxx): | NIL |

**Part 1 – Bash Scripting**

**Question 1.1 (1 mark)**

It is a special construct called Shebang that is the character sequence #! used at the beginning of a script. It indicates to the system what program (specifically command-line shell) is to be used to interpret the script, /bin/bash in this case.

**Question 1.2 (1 mark)**

I changed the line z=$x\*$y to z=$((x\*y)) to make the value of z an arithmetic expansion of x \* y instead of the string “15\*20”.

**Question 1.3 (1 mark)**

#!/bin/bash

echo "Hello $(whoami) from the future. \

Today is $(date --date="+3 days" +%A), \

$(date --date="+3 days" +%d) $(date --date="+3 days" +%B) $(date --date="+3 days" +%Y), \

and the time is $(date +%T)." # No need \"+3 days\" for time as same time

**Question 1.4 (1 mark)**

$# 🡺 Number of arguments (that are passed to the function in this case)

$1 🡺 Value of first function parameter

$2 🡺 Value of second function parameter

$@ 🡺 All function parameters as separate words

$? 🡺 Exit status of last executed command (the return value of func in this case)

**Question 1.5 (1 mark)**

I see the final value of i being printed on the screen. When exit(i) is called, the final value of i is the exit status returned to the operating system. Since $? gives the exit status of the last executed command as mentioned in Question 1.4, calling echo $? will send the final value of i as output to the screen (stdout).

**Question 1.6 (1 mark)**

Running “./slow 5 ; ./slow 10” gives the output equivalent to running “./slow 5” then “./slow 10”, one after the other.

Running “./slow 5 & ./slow 10” gives the following output (<> stands for a val):

[1st line] [<job-number>] <PID>

[Subsequent lines] *“./slow 5” and “./slow 10” concurrently executing in an interleaving fashion*

[Last line] [<job-number>]+ Exit 11 ./slow 5

“;” denotes sequential execution (or more specifically synchronous execution) while “&” denotes parallel execution (virtual parallelism with 1 foreground job and 1 background job).

(For grader only) Part 1 total: \_\_\_\_\_\_\_\_\_\_\_ / 6

**Part 2 – Playing with POSIX Calls**

**Question 2.1 (1 mark)**

Yes, the parent and child processes are executing concurrently (virtual parallelism). This can be seen from the interleaved output produced by the parent and child processes.

**Question 2.2 (1 mark)**

The parent’s parent is bash (the shell).

**Question 2.3 (1 mark)**

“ac” – contains the number of command-line arguments

“av” – contains the command-line arguments themselves stored in an array of char\* strings (1st is the program name followed by other arguments passed through the command line)

“vp” – contains the environment variables available to the program stored in an array of char\* strings (all elements have the format of VARIABLE=<value> except for the last element which is NULL)

**Question 2.4 (1 mark)**

int main() {

if(fork() == 0) {

char\* const argv[] = {"cat", "file.txt", NULL};

execvp("cat", argv);

}

else

wait(NULL);

}

/\* Explanation

To use execvp instead of execlp, just have to ensure that arguments are passed in a way that matches the calling convention of execvp instead of execlp (switched from a variadic argument list to a char\* const array in this case).

//\*/

**Question 2.5 (1 mark)**

“dup2” duplicates the file descriptor passed as the first argument (aka oldFd) and replaces the file descriptor passed as the second argument (aka newFd) with the duplicate (newFd is closed before this happens if it is already open). “dup2” was used here to make it such that ./talk receives input from file.txt through a duplicate of fp\_in and sends output to talk.out through a duplicate of fp\_out. “dup2” was used instead of “dup” so an existing file descriptor can be explicitly specified for replacement.

**Question 2.6 (1 mark)**

The reading end in the parent process has to be closed for its writing end to receive a SIGPIPE signal (so it can stop writing when the child process stops reading [reading end in the child process is closed]). The writing end in the child process has to be closed for its reading end to receive an EOF signal (so it can stop reading when the parent process stops writing [writing end in the parent process is closed]).

**Question 2.7 (1 mark)**

Setting up the pipe between ./slow and ./talk

I used fork() in int main() to create a child process. In the parent process, I closed the writing end (not used by parent), used dup2 to make input come from a duplicate reading end instead of stdin then I closed the original reading end. In the child process, I closed the reading end (not used by child), used dup2 to make output go to a duplicate writing end instead of stdout then I closed the original writing end.

(For grader only)

Part 2 total: \_\_\_\_\_\_\_\_\_\_\_ / 7

**REPORT TOTAL: \_\_\_\_\_\_\_\_\_\_\_\_ / 13**

**Demo: \_\_\_\_\_\_\_\_\_\_\_\_\_ /4**

**Total: \_\_\_\_\_\_\_\_\_\_\_\_\_/17**