**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 10 March 2024, 1 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: Eunice Koh Shu Ning |  |
| Student ID (A0256986E): |  |
| Group (08): |  |
| **Student 2** | |
| Name: |  |
| Student ID (AxxxxxxY): |  |
| Group (Bxx): |  |

**Part 1 – Bash Scripting**

**Question 1.1 (1 mark)**

#!: The shebang symbol.

/bin/bash: The path to the Bash interpreter.

shebang line is telling the system to use the Bash shell to interpret and execute the script

**Question 1.2 (1 mark)**

z=$((x - y)) instead of z=$x-$y

This modification uses the arithmetic expansion $((...)) to perform the subtraction and store the result in the variable z

**Question 1.3 (1 mark)**

#!/bin/bash

# Get user's name

user=$(whoami)

# Get current day, date, month, year, and time

dayOfWeek=$(date +%A)

day=$(date +%d)

month=$(date +%B)

year=$(date +%Y)

time=$(date +%H:%M:%S)

# Print the greeting

echo "Hello $user, today is $dayOfWeek, $day $month $year and the time is $time"

**Question 1.4 (1 mark)**

$#

Holds the number of command-line arguments passed to the script or function.

./myscript.sh arg1 arg2 arg3, $# would be 3.

$1

Holds the value of the first command-line argument passed to the script or function.

./myscript.sh arg1 arg2 arg3, $1 would be "arg1".

$2

Holds the value of the second command-line argument passed to the script or function.

./myscript.sh arg1 arg2 arg3, $2 would be "arg2".

$@

Represents all the command-line arguments as separate words.

./myscript.sh arg1 arg2 arg3, $@ would be equivalent to "arg1 arg2 arg3".

$?

Holds the exit status of the last executed command.

**Question 1.5 (1 mark)**

11

Exit Status:

exit(i); in C explicitly terminates the program and returns an exit status i to the operating system.

echo $? in Bash retrieves and displays the exit status of the last executed command.

Exit Status Value:

In C, the value i provided to exit(i); is the exit status returned to the operating system. In Bash, $? holds the exit status of the last command executed, and echo $? prints or displays that value.

**Question 1.6 (1 mark)**

Using a Semicolon (;):

* the commands are executed sequentially.
* The first command (./slow 5) runs, and only after it completes, the second command (./slow 10) starts.
* Essentially, the semicolon acts as a separator between the two commands, ensuring that the second command waits for the first one to finish.

Using an Ampersand (&):

* When you use an ampersand (&), the commands are executed concurrently.
* The first command (./slow 5) starts running, but it doesn’t wait for it to finish. Instead, it immediately starts the second command (./slow 10).
* Both commands run simultaneously in the background.

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

**Part 2 – Playing with POSIX Calls**

**Question 2.1 (1 mark)**

parent and child processes are executing concurrently. The evidence for concurrent execution can be inferred from the interleaved sequence of "Parent: i = X" and "Child: i = Y" statements in the output.

**Question 2.2 (1 mark)**

The shell itself

**Question 2.3 (1 mark)**

ac (Argument Count):

If value of ac is reported as 5, it indicates that there are 5 command-line arguments passed into the program, including the program name itself.

av (Argument Vector):

The output lists each command-line argument separately:

Arg 0 is ./lab2p2b

Arg 1 is this

Arg 2 is is

Arg 3 is a

Arg 4 is test

These represent the individual elements of the array of command-line arguments.

vp (Environment Variables):

The output includes information about various environment variables (Env 0 to Env 57).

Examples of environment variables include HOME, PATH, SHELL, USER, PWD, etc.

Each line starting with "Env" represents a different environment variable with its name and value.

**Question 2.4 (1 mark)**

#include <unistd.h>

#include <stdio.h>

#include <sys/wait.h>

int main() {

char \*args[] = {"cat", "file.txt", NULL}; // Arguments for the cat command

if (fork() == 0) {

// Replace the process image with the cat command

execvp(args[0], args);

} else {

wait(NULL); // Wait for the child process to finish

}

return 0;

}

* I created an array of strings called representing the command cat and its arguments file.txt. The last element of the array must be NULL to indicate the end of the argument list.
* In the child process (fork() returns 0), the execvp function is called.
* execvp replaces the current process image with a new one specified by the given command and its arguments.
* args[0] is the command ("cat") and args is the array of arguments.
* If execvp is successful, the child process continues running the new program (cat in this case).
* If execvp encounters an error, it returns, and the child process exits.

**Question 2.5 (1 mark)**

dup2 is a system call in Unix-like operating systems that duplicates an open file descriptor to another specified file descriptor. In this case, dup2 is used to redirect the standard input (stdin) and standard output (stdout) of a child process.

dup2(fp\_in, STDIN\_FILENO);

dup2 duplicates the file descriptor fp\_in onto the standard input file descriptor (stdin).

After this, any attempt to read from stdin in the child process will read from file.txt.

dup2(fp\_out, STDOUT\_FILENO);

dup2 duplicates the file descriptor fp\_out onto the standard output file descriptor (stdout).

After this, any attempt to write to stdout in the child process will write to talk.out.

**Question 2.6 (1 mark)**

A pipe establishes a one-way communication channel between two processes. The parent process writes data into the pipe, and the child process reads from it. If the pipe is not closed, the parent process might accidentally retain access to the reading end, leading to potential issues such as deadlock situations, inefficient resource usage, and confusion in communication channels. Closing the unused end of the pipe is crucial for proper synchronization and resource management in interprocess communication.

**Question 2.7 (1 mark)**

1. Create a pipe

int pipe\_fd[2];

// Create a pipe

if (pipe(pipe\_fd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

1. Fork a Child Process for ./slow and ./talk

pid\_t slow\_pid = fork();

pid\_t pid = fork();

1. Child Process of ./slow:

Redirect standard output to write to the pipe

Close unused file descriptors

Execute the "./slow 5" command

1. Child Process of ./talk:

Redirect standard input to read from the pipe

Close unused file descriptors

Redirect standard output to write to "results.out"

Close unused file descriptors

Execute the "./talk" command

1. Parent process

Close the write end of the pipe for slow

Close the Read End of the Pipe for talk

Wait for the child processes to finish

(For grader only)

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

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

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

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