**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 22 September 2024, 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.
* process\_jobs.sh
* lab2p2f.c

|  |  |
| --- | --- |
| **Student 1** | |
| Name: | Zachery Ng Ke Yuan |
| Student ID (AxxxxxxY): | A0269588B |
| Group (Bxx): | B04 |
| **Student 2** | |
| Name: |  |
| Student ID (AxxxxxxY): |  |
| Group (Bxx): |  |

**Part 1 – Bash Scripting**

**Question 1.1 (1 mark)**

i. Shebang. The first line specifies the path to the interpreter (bash) that the Operating System uses to execute the program.

ii.  If the shebang is omitted and the script is run directly (E.g. ./hello.sh), the OS uses the default shell for the environment to run the script. This might result in issues if the script uses specific features of a particular shell.

If the script is run by explicitly stating an interpreter (E.g. bash ./hello.sh), the specified interpretor will run the script.

If the path in the shebang is incorrectly specified, there will be an error “no such file or directory” as the OS is unable to find the specified interpreter.

**Question 1.2 (1 mark)**

i. By using arithmetic expansion $(( *expression* )), the variables x and y were treated as numeric values and subtraction is performed.

ii. Variables are treated as strings by default. The syntax $(( )) tells the shell to treat the expression as an arithmetic expression. This is needed as Bash is designed primarily to handle strings, not arithmetic expressions, so arithmetic operations in Bash require specific syntax.

#!/bin/bash

x=10

y=30

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

echo "$x - $y = $z"

**Question 1.3 (1 mark)**

#!/bin/bash

userid=$(whoami)

architecture=$(uname -m)

kernel=$(uname -r)

hostname=$(uname -n)

echo "Hello $userid, you are using a $architecture machine running kernel version $kernel on $hostname"

**Question 1.4 (1 mark)**

$# - Number of parameters passed into function

$1 – First parameter

$2 – Second parameter

$@ - All the parameters passed into function as a list

$? – Exit status of last command / function executed

**Question 1.5 (1 mark)**

i. 5. Exit(count) passes the value of count as the program’s exit status. $? Gets the exit status of the last command, which is equal to count.

ii. When exit(count) is removed, echo $? prints 0. When exit(count) is replaced by return count, echo $? Prints 5.

The program defaults to an exit status of 0 if there are no return or exit statements. When return <<value>> is used, the program exits with a status of <<value>>

**Question 1.6 (1 mark)**

i. “;” runs the programs sequentially. That is to say, after the first program finishes, the second program runs.

Output of ./slow "This is a test sentence" ; ./slow "I have increased the number of words in this sentence"

This

is

a

test

sentence

Number of words: 5

I

have

increased

the

number

of

words

in

this

sentence

Number of words: 10

On the other hand, “&” runs the programs concurrently, with interwoven outputs from both programs.

Output of ./slow "This is a test sentence" & ./slow "I have increased the number of words in this sentence"

[1] 11473

This

I

have

is

increased

a

test

the

sentence

number

Number of words: 5

[1] + exit 5 ./slow "This is a test sentence"

of

words

in

this

sentence

Number of words: 10

ii. Parallel execution for the tasks. Running the tasks in parallel allows the system to run both tasks simultaneously. This allows the log lines to be printed by Task 1 and Task 2 to send an email alert if the log line contains the specific keyword immediately without waiting for Task 1 to terminate.

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

**Part 2 – Playing with POSIX Calls**

**Question 2.1 (1 mark)**

Yes, the output is interleaved between parent and child process. Before the parent process prints sentence1 completely, the child process also prints a part of sentence2.

Parent: …

Child: …

Parent: …

When wait is removed, the parent process no longer waits for the child process to complete. Once the parent finishes printing sentence1, it exits without waiting for the child process to finish printing sentence2. This is seen from the “CHILD HAS EXITED WITH STATUS 0” line being printed before the child process finished printing.

**Question 2.2 (1 mark)**

The parent’s parent is the shell.

The process hierarchy in operating systems is a tree-like structure with init as the root. Init does not have any parent process. Every other process except the root process has a parent process.

**Question 2.3 (1 mark)**

ac – Argument count / number of command line arguments passed into program

av – Array of string pointers containing the command line arguments passed into program. av[0] holds the first argument, which is the name of the program.

vp – Array of pointers to environment variables

**Question 2.4 (1 mark)**

#include <unistd.h>

#include <stdio.h>

#include <sys/wait.h>

int main() {

if(fork() == 0) {

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

execvp("cat", args);

}

else

wait(NULL);

}

Changes: Specified the arguments in an array called args. Pass args as a parameter to execvp(“cat”, args);

**Question 2.5 (1 mark)**

i. dup2(int oldfd, int newfd) duplicates an open file descriptor (oldfd) to another file descriptor (newfd).

Dup2() is used because we want to redirect the standard input of the child process to read from file.txt instead of the default terminal input as well as redirecting the standard output of the child process to write to talk.out instead of the default terminal output.

ii. If execlp succeeds, the close() will not run, so open file descriptors remain open and are inherited by the new program (unless marked with FD\_CLOEXEC flag).

If execlp fails, close() will be executed to close the open file descriptors. This is because fp\_in and fp\_out are no longer needed.,

**Question 2.6 (1 mark)**

This prevents deadlocks, ensuring that the process would not wait indefinitely for data to be read or space to write in the pipe. If a process has the writing end open, there would not be end-of-file signaling, so the process continues to wait for more data to be read. The reading process will wait indefinitely as it assumes more data will arrive.

Furthermore, not closing the unused pipe ends can lead to resource leaks as open file descriptors may consume system resources.

**Question 2.7 (1 mark)**

Create a pipe by initailsing int p[2] and calling pipe(p). p[0] is the reading end,p[1] is the writing end.

Fork into parent and child process.

Parent process runs ./talk

Child process runs ./slow

In child process, duplicate writing end of pipe to stdout into by calling dup2(p[1], STDOUT\_FILENO);

In parent process, read output form reading end of pipe p[0].

By doing this, the output of ./slow will be written into the pipe by the child process and read by the parent process.

(For grader only)

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

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

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

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