**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 25 February 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: | Kenneth Seet |
| Student ID: | A0258173Y |
| Group: | B18 |
| **Student 2** | |
| Name: | Daniel Wang |
| Student ID: | A0255689H |
| Group: | B11 |

**Part 1 – Bash Scripting**

**Question 1.1 (1 mark)**

The first line of a Bash script, starting with #!/bin/bash, is called a shebang or hashbang. It is not a comment, and its purpose is to specify the path to the interpreter that should be used to execute the script. In this case, it indicates that the script should be interpreted and executed using the Bash shell.

The #!/bin/bash line tells the system where to find the Bash interpreter, and when you run the script, the system will use Bash to interpret and execute the commands in the script.

Without this shebang line, the system may use a default shell to execute the script, which might lead to unexpected behavior if the script relies on Bash-specific features. Including the shebang ensures that the script is executed with the intended interpreter.

**Question 1.2 (1 mark)**

We fixed the script by using double parentheses for arithmetic operation: z=$((x - y)) instead of z=$x-$y.

**Question 1.3 (1 mark)**

**#!/bin/bash**echo "Hello $(whoami), today is $(date +%A), $(date +%d) $(date +%B) $(date +%Y), and the time is $(date +%T)."

**Question 1.4 (1 mark)**

The following are special variables in Bash. What do they hold? $#, $1, $2, $@, $?

$#: Holds the number of arguments passed to a script or function.

$1: Holds the first argument passed to a script or function.

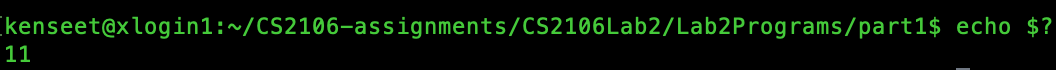
$2: Holds the second argument passed to a script or function.

$@: Stores the list of arguments passed into the function as an array.

$?: Holds the exit status of the last executed command. A value of 0 typically indicates success, while non-zero values indicate errors or other exit statuses.

**Question 1.5 (1 mark)**

I see the value 11 printed in the shell.



The final value of i in the code was 11, and when exit(i) was called, the program exited with the exit status of the value of i, which was 11.

Since $? stores the value of the exit status of the last executed command, it causes 11, which was the exit status, to be displayed on the shell.

In short, echo $? prints out the value of i from the last time exit(i) was called, which in this case, was 11.

**Question 1.6 (1 mark)**

When I ran “./slow 5 ; ./slow 10”, the output of ./slow 5 is displayed first, followed by the output of ./slow 10.

A computer screen with green text

Description automatically generated

When I ran “./slow 5 & ./slow 10”, the output of ./slow 5 and ./slow 10 both show up together, and there is no fixed order to which output appears first.

A computer screen with green text

Description automatically generated

When you run ./slow 5 ; ./slow 10, the semicolon ; is used as a command separator. This means that the second command (./slow 10) will only start executing after the first command (./slow 5) has completed.

On the other hand, when you run ./slow 5 & ./slow 10, the ampersand & is used for running the first command (./slow 5) in the background. This allows the second command (./slow 10) to start immediately without waiting for the completion of the first one.

In summary:

;: Executes commands sequentially; the second command starts only after the first one completes.

&: Executes the first command in the background, allowing the second command to start immediately without waiting for the completion of the first one.

So, the difference between the two commands lies in how they handle the execution of the commands in relation to each other (sequentially with ; or in parallel with &).

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

**Part 2 – Playing with POSIX Calls**

**Question 2.1 (1 mark)**

The parent and child processes are executing concurrently. From the output being:

A screenshot of a computer

Description automatically generated

We can observe that the output of the parent and the child are interleaved, and this shows that the parent and the child processes are executing their respective code blocks simultaneously.

**Question 2.2 (1 mark)**

The parent’s parent is 547981.

**Question 2.3 (1 mark)**

ac: Number of command-line arguments.

av: Array of strings representing command-line arguments.

vp: Array of strings representing 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);  
}

I replaced the old line of code,

execlp("cat", "cat", "file.txt", NULL);

with:

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

execvp("cat", args);

This places the arguments into a null terminated array of strings, instead of passing them one by one into execlp.

**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. The primary purpose of dup2 is to provide a way to redirect input or output streams of a process to a specific file or file descriptor.

In this context, we redirect the input from file.txt to the ./talk program, and the output from ./talk to the talk.out file.

**Question 2.6 (1 mark)**

Closing the unused ends of a pipe is good practice for proper resource management, avoiding deadlocks, signaling the end of data transmission, and preventing resource leaks in inter-process communication. It ensures that file descriptors are released appropriately and that processes can efficiently communicate through the pipe. Closing the ends of the pipe also allows the end of the file to be determined.

In this specific case, the parent should close its own read descriptor (p[0]) after forking, ensuring that when the child exits, the pipe is properly closed, and end of file tests can work correctly.

**Question 2.7 (1 mark)**

1. I created a pipe by making an int array of size 2, checking for errors in the POSIX call.

// set up pipe  
int p[2];  
// this creates a pipe. p[0] is the reading end,  
// p[1] is the writing end.  
  
// check for errors in POSIX call  
if (pipe(p) < 0) {  
 perror("lab2p2f: ");  
}

1. I forked a child process, and the parent process will execute ./slow 5, and the child will execute ./talk. The parent process will communicate with the child process through the pipe.
   1. Parent process:
   * Closes the reading end of the pipe
   * Redirects standard output of the process to the writing end of the pipe
   * Closes the writing end of the pipe
   * Executes ./slow 5
   * Waits for the child process to terminate
   1. Child process:
   * Closes the writing end of the pipe
   * Duplicates the reading end of the pipe onto the standard input
   * Closes the reading end of the pipe
   * Opens results.out to write the standard output of the process to
   * Closes results.out
   * Executes ./talk

(For grader only)

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

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

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

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