Assignment 2 – Unix Processes (5%)

Introduction

The assignment total is 10 marks. You should begin Assignment 2 after reviewing the materials in Module three; it is due by the end of Module four. Check your Course Schedule for the precise due date. Directions for submitting Assignment 2 to your Open Learning Faculty Member for grading can be found in the Assignments Overview tab. An assignment marking criteria follows at the end of this document.

Instructions

Through this assignment, you will learn how to use multiple threads in you UNIX application programs. We will work on process creation and termination and the way a process can be sent to the background, using a Small C program.

> You must submit C files and screen shots that show how your program works. Refer to the marking criteria at the end of this document.

Problem:

1. `exec()` - The **exec**() family of functions replaces the current process image with a new process image.

2. `fork()` - **fork** is an operation whereby a process creates a copy of itself

3. `wait()`-The **wait**() system call suspends execution of the current process until one of its children terminates.

4. `exit()` - The function **_exit**() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed;

5. `kill()` - The **kill**() system call can be used to send any signal to any process group or process.

To do:

Q1. Get onto the TRU Linux machine and run the system calls, fork(), wait() and exec()

Write a program named as "testOS.c" that executes the "*cat –b –v –t filename*" command in the child process

Code for "testOS.c" to compile and run:

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

int main (int args, char *argv[])
 {
   pid_t fork_return;
              pid_t pid;
              pid=getpid();
              fork_return = fork();

   if (fork_return==-1) {
     // When fork() returns -1, an error has happened.
     printf("\nError creating process " );
               return 0;
   }
    if (fork_return==0) {
      // When fork() returns 0, we are in the child process.
      printf("\n\nThe values are Child ID = %d, Parent ID=%d\n", getpid(), getppid());
      execl("/bin/cat", "cat", "-b", "-v", "-t", argv[1], NULL);
    }
          else {
     // When fork() returns a positive number, we are in the parent process
     // and the return value is the PID of the newly created child process.
     wait(NULL);
     printf("\nChild Completes " );
              printf("\nIn the Parent Process\n");
     printf("Child Id = %d, Parent ID = %d\n", getpid(), getppid());
              }
              return 0;
}
```

Details:

a. Type the following program in Linux

b. Compile it using gcc as **gcc testOS.c**

c. Run the program as the following command: ./a.out **filename** where **filename** is the name of a text file that you have created. The text file should contain multiple lines of text. For example:

> This is line one in my text file.
> This is line two in my text file.
> This is line three in my text file.
> Linux provides many commands for manipulating text!

What is happening:

1. Your code will *fork()*
2. The child will use the *execl* to call *cat* and use the filename passed as an argument on the command line
3. The parent will wait for the child to finish
4. Your program will print from both processes:
   - The process id
   - The parent id

Q2. Once the first program is typed, and executed, answer the following questions

a. If you try to print a message after the *exec** call, does it print it? Why? Why not?

b. Who is the parent of your executable program?

c. How would you change the code so that the child and parent run concurrently?

Running a Process

When you start a process (run a command), there are two ways you can run it:

- Foreground Processes
- Background Processes

Foreground Processes:

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

Background Processes:

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

Stopping Processes:

If a process is running in background mode then first you would need to get its Job ID using **ps** command and after that you can use **kill** command to kill the process as follows:

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD amrood
6738  3662  0  10:23:03  pts/6  0:00  first_one  amrood
6739  3662  0  10:22:54  pts/6  0:00  second_one  amrood
3662 3657 0 08:10:53 pts/6 0:00 -ksh amrood   6892
3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here **kill** command would terminate first_one process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows:

```
$kill -9 6738
Terminated
```

Q3. Execute and document the commands to:

a. Run a process in background
b. Run a process in foreground
c. Call the process from background to foreground
d. Kill the process
e. Show the process status

Results:

Hand in:

- A print out of the programs;

- A printout of the output of commands (screen capture)

- Answers to questions listed

Assignment Marking Criteria	Weighting
No syntax error: All requirements are fully implemented without syntax errors. Submitted screen shots will be reviewed with source code.	/5
Correct implementation: All requirements are correctly implemented and produce correct results Submitted screen shots will be reviewed with source code. Questions are answered correctly.	/5
Total	/10