SCHOOL OF COMPUTING, QUEEN'S UNIVERSITY

CISC324: Operating Systems

Lab 1

December 20, 2019

This lab has been prepared to be conducted on UNIX-like operating systems (e.g., Linux Ubuntu). Please make sure to have a well set up environment. You will need a computer, a Linux operating system, and a C-compiler. You can still use the Windows operating system but you will have to install additional tools such as Cygwin.

Objective

During this lab, you will learn how to use the most important process management system calls. You will be writing programs, using C-programming language, to create new processes, terminate running processes, wait for processes (synchronization), change the code segment of processes, and get the attributes of processes. The following POSIX API system call primitives will be used:

1. `fork()`: By executing the primitive `fork()` in a C-program, the involved process makes a system call to the kernel through the POSIX API asking the it to create a new process (child process). The new process starts its execution from the instruction statement that is just after the `fork()` system call that created it. At the same time, the primitive `fork()` returns a value n of type `pid_t`. The parent process will have the value $n > 0$ (equal to the PID of the created child process), whereas the created child process will have $n = 0$. If $n = -1$, then `fork()` has failed its execution.

2. `getpid()`: This primitive returns the value of the PID of the process that executed it. Recall that each process has a unique identification number called PID (Process IDentification number) that is used by the system to identify the process.

3. `getppid()`: This primitive returns the value of the PID of the parent of the process that executed it. Recall that each process has a parent process that created it (Except init which has PID=1).

4. `exit(v)`: Terminates the process which executes the primitive and returns to the parent process a one-byte integer value contained in the value of v.

5. `wait(&status)`: When executed by a parent process, the later waits for its child process termination notification. When a child process terminates, the primitive `wait()` returns the PID of the child process which terminated. Also, it returns from the address status (`&status`), the value sent by the exit primitive (see the value of v in `exit()`) and the cause of the termination of its child process (all in one-byte). Basically, the most significant byte contains the value of v, and the less significant byte carries the cause of the termination. You can use the macro `WIFEXITED(status)` to retrieve the cause (1 normally terminated, otherwise abnormally terminated) and `WEXITSTATUS(status)` to retrieve the value of v. Finally, if there is no child to wait for, `wait()` returns `-1`.

To use these primitives, make sure that your C-program includes the following header files: `<unistd.h>`, `<wait.h>`, `<sys/types.h>`, and `<stdlib.h>`.

Exercise 1

Assume that we possess a multiprocessing computer and that we would like to compute, using a computer program, the sum of a sequence from 0 to n (see equation below), where $n > 0$. To speed up the computations, we would like to implement our program in such a way so that we make use of multiprocessing. A simple intuition consists of dividing the sum into two parts that will be run by two different processes. Let us say process P_1 executes the sum from 0 to $[\frac{n}{2}]$, and process P_2 executes the sum from $[\frac{n}{2}]+1$ to n. Process P_1 is set the task to display the final result. `Exer_1.c` is a typical implementation of this scenario using the C-programming language under POSIX environment.

$$\sum_{i=0}^{n} i = 0+1+2+\cdots+n$$

1. *Compile and execute the program for different values (you may have to use the option `-lm` to include the math.h library before compiling). Does the program perform the correct computations? If yes, why?, if no, why?*

2. *The program appeared to be returning wrong computations (e.g., for $n = 1$ it returns 0). Modify the program code to fix the issue using `wait()` and `exit()` system calls. Explain your modifications and explain the limitation of this solution.*

Exercise 2

The program in `eXer_2.c` consists of one parent process that creates three other child processes. Then, each child process, tries to execute the program `count.c`. By executing the latter program, each process opens a shared file named `nums.txt`, reads the stored value, increments it, then rewrites the new value back to the file, for 5000 times. By compiling the program `count.c` using commands such as (`$cc count.c -o count.out`) and placing the output in the same directory (folder) as program `eXer_2.c`, then if each of the three processes reads the value from the file `nums.txt` then increments that value before writing it back to the file, in the normal circumstances, we should find at the end of the execution that the file contains the value `15000 (5000x3)`.

1. *Execute the program `eXer_2.c` multiple time (5 to 6 times) and observe the final value that is stored in the `nums.txt` file. Explain your observation (note that you should remove the file `nums.txt` each time before re-executing the program).*

2. *By keeping the three child processes and using `wait()` primitive to communicate with their parent process, modify the program `eXer_2.c` in such a way so that the final value that is stored in `nums.txt` is always `15000`.*