CISC324: Operating Systems

Lab 1

Deadline: Monday, September 19, 2022 at 11:59PM

Instructor: Dr. Anwar Hossain
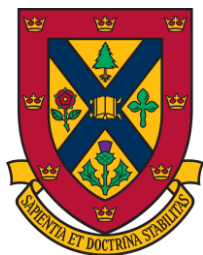
2022

Toolbox

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 (or MacOS, which is a UNIX-based OS), and a C-compiler. You can still use the Windows operating system but you will have to install additional tools such as Cygwin. An alternative would be to install Linux on a virtual machine (e.g., VirtualBox or VMWare) over Windows. You can also remotely use the CASLAB Linux computers.

Background and Outcomes

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>`.

1 Process Communication Issue

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 $\left[\frac{n}{2}\right]$, and process P_2 executes the sum from $\left[\frac{n}{2}\right] + 1$ 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). The program appears to be returning incorrect computations (e.g., for $n = 1$ it returns 0). Why is that?

2. Modify the program code to fix the issue using `wait()` and `exit()` system calls (along with macros WEXITSTATUS($\cdot$) and WIFEXITED($\cdot$)). Explain how you fixed the issue.

3. After fixing the issue, you may notice that starting from a certain value n, the returned sum becomes incorrect. What is the value of n? Explain the reason behind this limitation.

4. If we switch the function of the parent and child process (i.e., A($\cdot$) by B($\cdot$) and B($\cdot$) by A($\cdot$) in the source code), what would be the value of n?

Note. You need to create a ReadMe.txt file to type down some of your answers.

2 Race-Condition Issue

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 times (5 to 6 times) and observe the final value that is stored in the `nums.txt` file. Explain why the final value that is stored in the file never reaches `15000` (note that you should remove the file `nums.txt` each time before re-executing the program).

2. Theoretically, the final value v is bound by two values (i.e., $n \leq v \leq$ `15000`). What is the value of n (lower bound)? and provide an execution scenario that can lead to the final value being n.

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

3   What to submit

1. Place all your source codes in a folder named in the following format :
   324-1234-Lab1 where 1234 stands for the last 4 digits of your student ID,
   e.g.: For student with ID 20196072, the folder should be named 324-6072-Lab1.
2. Place a ReadMe.txt file into the same folder above.
3. Compress the above folder using Zip (the extension must be .zip).
4. Log into OnQ, locate the lab's dropbox, and upload the zipped folder.

4   What to check during submission

1. Check that you are not submitting an empty folder □.

2. Check that you are not submitting the executable files (i.e., compiled) □.

3. Check that you are not submitting the wrong files □.

4. Check that you are not submitting to the wrong dropbox □.

5. Check that you are submitting before the deadline □.