School of Computing
Queen's University

CISC324: Operating Systems

Lab 1: Marking Scheme (20/20)

Deadline: Monday, Oct 4, 2021, at 23:59 PM

Instructor: Dr. Samir Mohammad

2021

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 $[\frac{n}{2}]$, 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). The program appears to be returning incorrect computations (e.g., for $n = 1$ it returns 0). Why is that?

   There is no actual communication between the parent process and its child process so that they can cooperate (2.0 points).

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.

   The parent process has to execute `wait(&status)`, where `status` is an integer variable. This will force the parent to block and wait for a signal from the child process when the latter terminates (2.0 points).

   The child process has to execute `exit(total)`, where `total` is an integer variable that contains the sum that the child computed (2.0 points).

   When the child process terminates and executes `exit(total)`, the content of the variable `status` gets affected (specifically, the two less significant bytes of `status`). The parent uses `WIFEXITED(status)` to retrieve the cause of the termination of the child, and it uses `WEXITSTATUS(status)` to retrieve the lowest byte of the child's variable `total`. Hence, the parent process can obtain the sum that the child computed and added to the sum that it has locally computed (2.0 points).

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.

   The value is $n = 26$. In fact, for $n = 26$, the child compute a summation that results in 260, and then sends it through `exit()`. Then, the parent uses `WEXITSTATUS(status)` to retrieve the lowest byte of 260. Hence, the parent process obtain the incomplete value from the child (2.0 points).

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

   The value is $n = 46$. In fact, for $n = 46$, the child compute a summation that results in 270, which cannot be expressed in 8 bites (1.0 points).

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).

   By executing the program multiple times (and removing the file nums after each execution), we can observe that the final value that is stored in the nums file is less than 15000. Normally, if each process writes for 5000 times, then three processes would write a total of 15000. However, this is not the case. In fact, in a time-sharing system (adopting RR or variants of RR), processes are concurrently executed (i.e., the execution bursts are either overlapping, e.g., in multicore/multiprocessor systems, or interleaving, e.g., single-processor system). They are allocated the CPU for a certain amount time (quantum) before being switched by another process. Therefore, there is no guarantee that a given set of instructions, belonging to a given process Px, will be executed atomically without being interrupted by the system so that another set of instructions, belonging to another process Py, are executed.

Concretely, if a process is programmed to read a value val from a file F.txt and write a different value val+1 to that same file, then there is no guarantee that this process will not be interrupted for some time before getting the CPU back to write the value val+1 to the file. During the interruption time, other processes could have had access to the same file and have read the stored value, which is val. Then all these processes will have the chance to execute again an write their modified value, which is val+1. So, assuming the existence of three processes, and assuming that the initial value is val=0, instead of having the final value val=3, we may obtain val=1.

This situation (paradigm) is called race-condition. It occurs when multiple processes are trying to access in read and write mode a shared resource. The final status of that shared resource will depend on the order of execution of the individual instructions of these involved processes. From a top viewpoint, the instructions (of all these processes) will be mixed up. Theoretically, there will be different combinations (execution scenarios), but only one scenario will occur at a given execution.

In this question students should show through their explanation that they understood the reason why 15000 cannot be obtained. By providing enough arguments and explanations, a student gets full mark. The TA should estimate the mark for this question based on the student explanation and w.r.t. the above explanations (3.0 points).

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.

   The value is $n = 2$ (1.0 point).

   One of the scenarios can be as follows (3.0 points):

   (1) All processes read the value of 0 from the file.

   Content of the file: 0

   (2) Process P1 executes 5000 times, writes the value 5000 ,and terminates.

   Content of the file: 5000

   (3) Process P2 executes 4999 times and gets interrupted.

   Content of the file: $x < 5000$

   (4) Process P3 completes its first round and writes the value 1 to the file.

   Content of the file: 1

(5) Process P2 executes its last round by reading the value 1.

Content of the file: 1

(6) Process P3 reads the value 1 and executes till completion.

Content of the file: $x < 5000$

(7) Process P2 gets the CPU and writes 2 into the file and terminates..

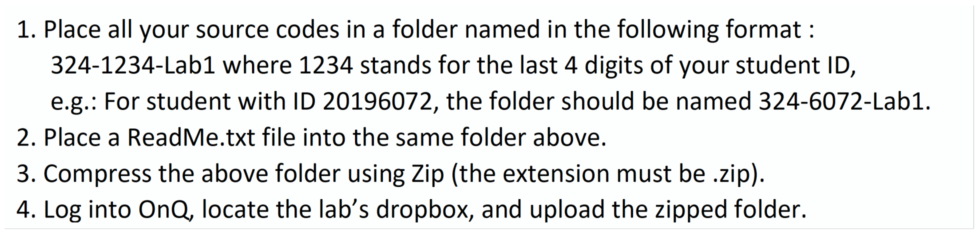
Content of the file: 2

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`.

   A typical solution is to make the three processes execute sequentially. For example, the parent process create the first child and then waits for its termination. Once, terminated, it creates the second process, waits for its termination, then create the third process and wait for its termination. The final vale would be 15000 (2.0 points).

   A full mark is given to students who code their solution in a different way that executes correctly.

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 ☐.