**Instructions** This lab assignment explores the data shared problem and process synchronization using Peterson’s solution.

Objectives of this assignment:

* to work on a Unix based system
* to “*dust off*” your programming skills in C
* to understand the fork() function to create a”child” process
* to understand the relationship (or lack of) between parent and child process
* to experience the ***data shared*** problem
* to deploy the **Peterson’s solution** to address the data shared problem

**IMPORTANT:**

1. *Your code will be tested and graded* ***REMOTELY*** *on the Engineering Unix (Tux) machines. If the code does not work on those machines, you will not get any credit even if your code works on any other machine.*
2. *A late submission will get a 50% penalty if submitted right after the deadline. The next day, you cannot submit the lab.*
3. *One submission per group.*
4. *Writing and presentation of your report are considered to grade your lab (30%). Your conclusions* ***must be supported*** *by the data/measurements you collect.*
5. *The quality of your code will be evaluated (****80%****).*
6. ***Questions about this lab must be posted on Piazza if you need a timely answer benefiting all students****.*

**Use this file to answer the questions. Highlight your answers and do NOT remove anything from this file. Just Insert your answers.**

**Part I: Programming on Tux machines**

**(10 points) Program Exercise 1**:

# Exercise 1: Download the program *lab2-1.c*. Compile it and execute it. Observe the code and observe the output. This program has a parent and child processes *sharing* a variable. This program is *intended* to increment the shared (common) variable counter *\*countptr*. The parent process is *supposed* to increment *\*countptr* by increments of 20 while the child increments by 2s. A satisfactory execution of this program may be: the child increments the counter *\*countptr* twice (reaching 4), then the parent increments the counter *\*countptr* thrice to reach finally 64. Answer the following questions:

1) Does the program really execute as supposed (or intended)? Describe/Justify/Explain

2) Is the variable \****countptr*** really a shared (common) variable? In other words, are the changes made to \**countptr* by the child visible by the parent, and *vice versa*?  Describe/Justify/Explain.

**(90 points) Program Exercise 2**:

The program ***lab2****-****2.c*** creates a genuine **shared** variable \**countptr*. Download, compile, and execute this program.

1)       Based on the execution, show that \**countptr* is now a genuine shared variable (*countptr* points to a zone shared by the parent and the child). Now, are the changes to \**countptr* made by the child visible by the parent?

2)       Does the program really execute as supposed (or intended), i.e, the counter increases exclusively in increments of 2 or 20? Explain what is happening.

3)       **Without modifying** the routine *add\_n()*, use the *Peterson’s* *solution* to correct the program ***lab2-2.c***. to execute as intended: the variable should increase by 2’s or twenty’s

***Hint***: Besides the pointer ***countptr*** used to point to the shared memory zone, you need to map three other integers Interested[2] and Turn (Peterson’s variables); These variables may be shared exactly the way that the zone pointed by *countptr* is shared.

**What to turn in?**

**Electronic copy**

Turn in separate files:

1. THIS file with INSERTED answers
2. Program ***lab2-2.***.c (corrected)

**A penalty of 10 points will be applied if these instructions are not followed.**

1. Your report must:
   1. state whether your code works. If is does work, state any issues you are aware of.
   2. Good writing and presentation are expected.

Lab 2 Assignment Report

Exercise 1:

(1) Lab2-1.c does not execute as it is supposed/intended to. It "is intended to increment the shared(common) variable counter \*countptr" (lab2.docx), but the variable is not shared between the child and parent processes. This is the output we found when running the program (figure 1.1):

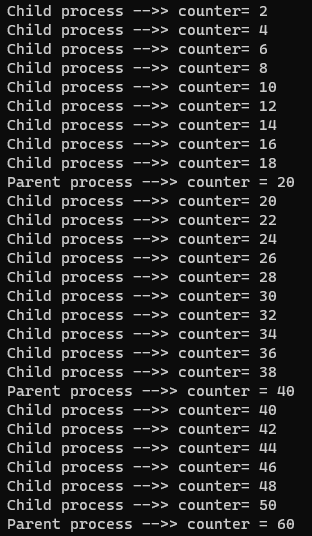


Figure 1.1

Evidence (child process):

* The first time the child process calls the add\_n function, the result is "counter = 2". Supposing that the add\_n function is written correctly, that must mean that counter was incremented two times, and that it was initially equal to 0 upon the first increment.

Evidence (parent process):

* The first time the parent process calls the add\_n function, the result is "counter = 20". Supposing that the add\_n function is written correctly, that must mean that counter was incremented twenty times, and that it was initially equal to 0 upon the first increment.

Both processes started at counter = 0 (at the same time), when they should have been sharing the variable. That is, when the parent process first started incrementing, it should have read the counter variable from the child process, and as such counter would not have been equal to 0 on the parent process's first incrementation.

(2) "The parent process is supposed to increment \*countptr by increments of 20 while the child increments by 2s." (lab2.docx) While the increments are technically correct, the two processes should NOT be outputting at the same time. The parent process should wait until the child process finishes its increments. Since it does not wait, and the count variable is not shared between them, then neither process can see the other's count variable, and neither is waiting for the other to finish.

Exercise 2:

(1) In this program, the variable \*countptr is mapped to a shared space between both of the processes using the mmap() function. Therefore, both processes can see the changes that the other makes to the variable.

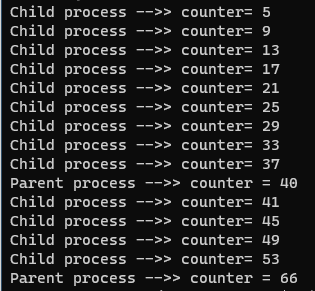
(2) The program does not, however, run as intended even though the variable is now appropriately shared. This is the output we found when running the program (figure 2.1): 

Figure 2.1

Instead of increments of two and twenty, the parent and child processes are incrementing in varying and incorrect amounts. This is happening because while both processes have access, mutual exclusion is not enforced, and therefore modifications are happening simultaneously in both processes.

To make the program work as it is supposed to, we would need something that makes the processes wait to start their critical section (calling the add\_n function) until the other process has finished.

(3) After implementation of Peterson’s Algorithm, the file runs as expected.

This is the output of lab2-2.c we found when running the program (figure 2.2):

Text

Description automatically generated

Figure 2.2

The parent and child processes are correctly incrementing in correct amounts of two and twenty. The Peterson’s Algorithm is properly handling the multiple processes by adding the variables turn and flag[2]. Now, when a process requests/wants to execute in the critical section, it sets its flag to true and turn as the index of the other process. The result of this is the process wanting to execute allows the current process to run fist performing busy waiting until it has finished its own critical section. After the current process completes, it sets its own flag to false as indication. The file is not free of synchronization issues, however. The changes to the file are being made in real time, and the processes are waiting for each other to execute, but there is nothing to check whether modifications have actually been made. The code would run the way it is written whether those changes happened or not, ending up with an incorrect output.