Project 2 Report

Daniel Yowell
113558774
CS3113
Dr. Song Fang
4/5/2023

***Abstract*—This project demonstrated how to coordinate access to shared memory in C by using semaphores, preventing race conditions or other undesirable outcomes.**

Keywords—C, process, shared memory, semaphore

I. Overview

Our goal for this project was to build upon the ideas established in Project 1, which introduced shared memory and inter-process communication. Namely, we wanted to protect the "critical sections" of our code—areas where memory is shared between processes—to mitigate the random outputs we observed in our previous project.

II. Concepts

A. Semaphores

When several processes share an address in memory, we run the risk of having a process access this memory at an inconvenient time (in other words, the critical sections of two processes overlap). This is called a "race condition," and it can generate unexpected or unintended results unless kept in check.

Semaphores are variables used to designate which process has permission to access shared memory. By using semaphores, we can ensure that only one process is accessing or modifying shared memory at a time, with any other processes placed in a waiting state until the current process exits its critical section.

B. Semaphore Libraries

While working on this project, I found two different libraries for implementing semaphores in C: <sys/sem.h> and <semaphore.h>. While I somewhat preferred the clarity of <semaphore.h>, I was only able to successfully implement the <sys/sem.h> format on OU's Linux servers. Fortunately, the lecture slides for this class provided a template for semaphore implementation using the <sys/sem.h> library, which I frequently referred to over the course of development.

III. Analysis

After correcting the errors I made in Project 1, I reworked much of the design to meet the requirements of Project 2. This code also ran successfully on OU's servers and demonstrated the concepts outlined in the project description.

Here are two different outputs I received when running this program:

```
From Process 1: counter = 100000.
Child with ID: 17050 has just exited.
From Process 2: counter = 300000.
Child with ID: 17056 has just exited.
From Process 3: counter = 600000.
Child with ID: 17057 has just exited.
From Process 4: counter = 1100000.
Child with ID: 17058 has just exited.

End of simulation.
```

```
From Process 4: counter = 500000.
Child with ID: 17097 has just exited.
From Process 1: counter = 600000.
Child with ID: 17094 has just exited.
From Process 3: counter = 900000.
Child with ID: 17096 has just exited.
From Process 2: counter = 1100000.
Child with ID: 17095 has just exited.

End of simulation.
```

It can be observed that the process functions still ran in a seemingly random order (1-2-3-4, 4-1-3-2, etc). This is likely because I forked all of my processes prior to making calls to the process functions. As such, the scheduler in my computer decided which process should enter the critical section first. Thanks to the semaphore variable, all other processes were locked out of the critical sections of their code until the first process exited its own critical section, after which the semaphore signaled for another process to continue, and so on.

Note that the final value of the shared memory variable is always the sum 1100000. By coordinating our processes properly, we can guarantee that this expected output is always what we get, no matter how these processes were scheduled beforehand.

IV. Conclusion

This program successfully demonstrated the importance of process management using semaphores, providing an example of how the usage of semaphores can achieve a desired result and prevent potential bugs. My code successfully achieved the project requirements, while also serving as a valuable learning experience to assist in building future C programs.