Project 2 Semaphores Report

Connor Finley

*Computer Science, University of South Florida*

finleyc@mail.usf.edu

***Abstract*— This document outlines the findings of the second project in Operating Systems, which dealt with semaphores: creating, initializing, waiting, and signaling, and deallocating them.**

***Keywords***— **semaphores, signal, wait, processes, fork, exit, wait, shared memory**

1. Introduction

This project added an additional layer to the first project by implementing semaphores on top of shared memory. By doing so, the shared memory variable could be protected from the other processes while being manipulated.

1. Code Structure
2. *Semaphores*

More blocks of code were added to handle semaphores properly. These code blocks, provided by the project instructions, setup the implement of semaphores using the Unix System V library, rather than the POSIX standard. The semaphore and its operations had to be initialized through utilization of the library, buffers, and unions. To use the semaphore the way it was intended was fairly simple. The critical section that manipulated the ‘total’ shared memory variable had to be wrapped in wait and signal function calls.

1. *Shared Memory Alterations*

The shared memory variable, ‘total,’ was incremented in a different fashion than in the first project. This time, instead of limiting the increments using the total variable, the total was incremented using constant targets: 100,000, 200,000, 300,000 and 500,000.

1. Findings

The performance was ostensibly the same as the first project in that results were given instantly. This time, since semaphores were used, each process incrementor function had to wait until the previous incrementor function had finished. This gave a more procedural look to the results.

Some readouts can be seen below.

From process 1: total = 100000

Child with ID: 64867 has just exited.

From process 2: total = 300000

Child with ID: 64868 has just exited.

From process 3: total = 600000

Child with ID: 64869 has just exited.

From process 4: total = 1100000

Child with ID: 64870 has just exited.

End of program.

---

From process 1: total = 100000

Child with ID: 65072 has just exited.

From process 2: total = 300000

Child with ID: 65074 has just exited.

From process 4: total = 800000

From process 3: total = 1100000

Child with ID: 65075 has just exited.

Child with ID: 65076 has just exited.

End of program.

Most of the time, readouts like the first two were seen.

1. Conclusions

Shared memory can be temperamental and semaphores help with this fact. The use of wait() and signal() aid in making sure critical sections are protected when it comes to multiple processes and prevents unintended effects. Semaphores in general, though trivial in implementation, provide a useful fix to what can be a big problem in multiprocessing and multithreaded environments. I believe the expectation of this project was to gain an understanding of how the addition of semaphores impacts our previous code and its results. Just like the last project, this specific application would probably be useless in reality due to its uncertainty, but it allowed us to gain a quick and simple understanding of semaphores and how to implement them in more meaningful systems.