Programming Assignment 2 – Semaphores

Brianna Patten
Student ID #113632860
University of Oklahoma
brianna.patten@ou.edu

I. Introduction/Project Overview

This project's primary objective was to further familiarize students with shared memory. As multiple processes originally accessed shared memory (programming project 1), it was critical to learn about synchronization and semaphores to protect the shared memory. By using semaphores, the shared memory was only accessed by one process at a time, allowing a more secure and efficient program.

II. Finding a Solution to the Problem

The main problem that arose in the programming project was figuring out how to correctly protect the shared memory while allowing the processes to run efficiently. By implementing the POP() and VOP() functions within the incremental function I had created in project 1, this issue was resolved.

```
//Function to increment "total" based on individual process limits
void processInc(int numToIncrement, int num) {
    for (int i = 1; i <= numToIncrement; i++) {
        // Acquire the semaphore to protect the critical section
        if (POP() == -1) {
            perror("Error in acquiring semaphore");
            exit(1);
        }
        total->value++; // Critical section

        // Release the semaphore after the critical section
        if (VOP() == -1) {
            perror("Error in releasing semaphore");
            exit(1);
        }
    }

}
```

Figure 1

My program still required the for-loop and the exit statements so the program would run correctly, so the two crucial functions were placed accordingly (before and after the crucial point, which was the incrementation of the shared memory) within the existing code.

Before this main issue arose, I had to include an additional library in order to even utilize semaphores. This was the '#include <sys/sem.h>' line that was added. I also consistently faced an error with SETVAL, which was resolved by altering 'value' and 'value1' to let the second parameter (NSEMS) in the 'semctl' arguments to be 0. This is because, at that moment, there were no active semaphores yet.

Lastly, I implemented sufficient error checking to be sure that I would thoroughly understand what was happening if my program had an error. This allowed me to work quickly and gain a better understanding of my program.

```
//Creating semaphore
sem_id = semget(SEMKEY, NSEMS, IPC_CREAT | 0666);
if(sem_id < 0){
    printf("Error in creating the semaphore./n");
}else{
    printf("Successfully created semaphore.\n");
}
//Initializing semaphore
value1 = semctl(sem_id, 0, SETVAL, semctl_arg);
value = semctl(sem_id, 0, GETVAL, semctl_arg);
if (value < 0){
    printf("Error detected in SETVAL.\n");
} else{
    printf("Successfully initialized semaphore.\n");
}
```

Figure 3.1

III. Post-Execution Discussion

Similar to programming project 1, we can recognize that the outputs, excluding the process that finishes last, vary during each run. This was thoroughly explained both in lecture and in the

programming project 1 report, which examined why the totals of processes changed during each separate run. This programming project is different, as the last-running process always ends at 1,100,000. This is because each process has a "target" number to increment to. Even though the total may print as over or under this number during the process' execution, the total that the last process prints will be 1,100,000 because all processes will have contributed their target number to the shared memory variable.

Figure 3

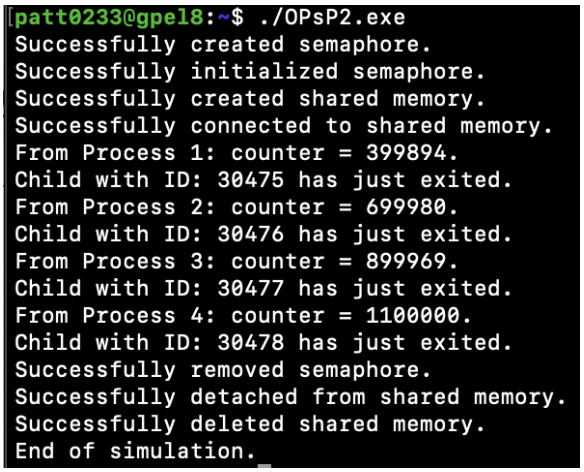

Figure 4

We can further understand the shared memory variable by understanding semaphores, which protects our shared memory. In the program, four child processes were created with the intention to increment 'total' (our shared memory). In order to protect this, semaphores were used. These ensured that only one process could access 'total' at a time, while the others waited in queue. This is why we had consistent results for our ending value of 'total', but different results for the printed values beforehand.

IV. Conclusion

Overall, this project further helped me understand shared memory and how it can be protected. I intend to move forward and continue to use what I learned about process synchronization mechanisms.