Project 2: Shared Memory & Semaphores

Daniel Sawyer U3363-7705
COP4600 Operating Systems Fall 2017
University of South Florida, Tampa

***Abstract*— Using shared memory across multiple processes using semaphores to lock it.**

I. INTRODUCTION

In this project we use shared memory across four separate child processes forked from one parent process. We use a semaphore to lock the resource then we wait for child processes to complete and print that it has ended. The semaphore prevents memory read/write collisions.

II. FUNCTIONS USED & OUTPUT

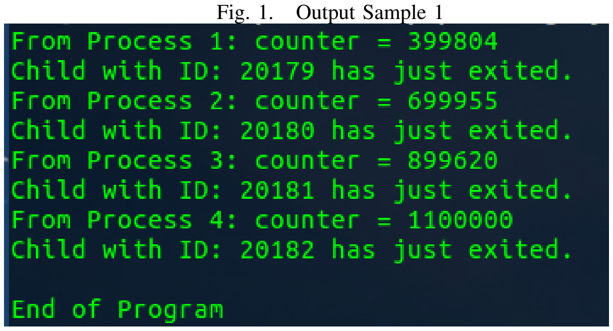
A. Functions Used

For this project, we are required to fork a parent process four times and have each child run a single function then exit. We do this by checking if the PID (Process Identification) returned by fork() is equal to zero. If the PID is equal to zero, we know we are in that child process and execute its corresponding function. Besides the fork() function, we also use shmget() for getting shared memory ID, shmat() for attaching the shared memory to the process, shmdt() for detaching the shared memory once the process is done with it, and shmctl() for removing the shared memory once the parent process recognizes all the children have finished. We also use a semaphore and functions POP(), which waits and locks the shared memory once available, and VOP() which releases the shared memory once we are done with it. This prevents memory access collisions and allows the final number to be the correct 1.1 million unlike the first project.

B. Output

The output for this project is all over the place. This is due to its parallel nature. When processes run in parallel, there is not always uniformity in the output. The programmer has limited control over when and how each process can execute. The end result is 1.1 million which is correct though and this is due to using semaphores to lock the shared memory resource which removes the memory collisions we had in Project 1.

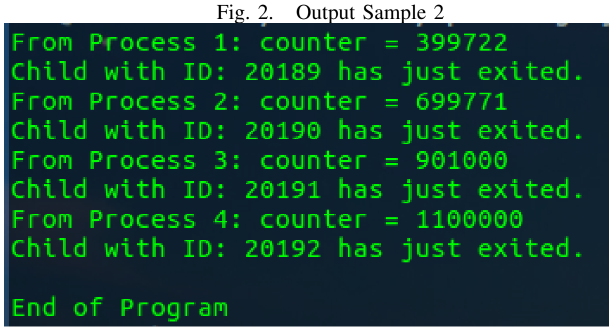
Fig. 1. Output Sample 1



```
From Process 1: counter = 399804
Child with ID: 20179 has just exited.
From Process 2: counter = 699955
Child with ID: 20180 has just exited.
From Process 3: counter = 899620
Child with ID: 20181 has just exited.
From Process 4: counter = 1100000
Child with ID: 20182 has just exited.

End of Program
```

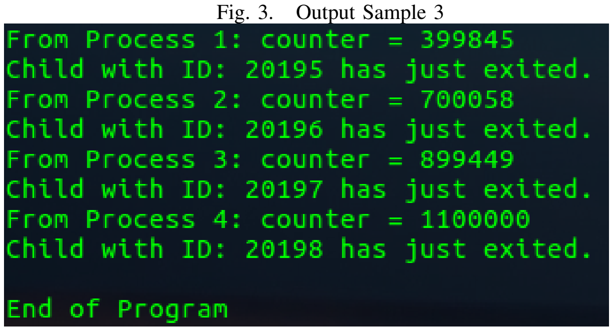
Fig. 2. Output Sample 2



```
From Process 1: counter = 399722
Child with ID: 20189 has just exited.
From Process 2: counter = 699771
Child with ID: 20190 has just exited.
From Process 3: counter = 901000
Child with ID: 20191 has just exited.
From Process 4: counter = 1100000
Child with ID: 20192 has just exited.

End of Program
```

Fig. 3. Output Sample 3



```
From Process 1: counter = 399845
Child with ID: 20195 has just exited.
From Process 2: counter = 700058
Child with ID: 20196 has just exited.
From Process 3: counter = 899449
Child with ID: 20197 has just exited.
From Process 4: counter = 1100000
Child with ID: 20198 has just exited.

End of Program
```

III. CONCLUSIONS

As you can see from the figures above, the count value is way off for the first three but the final value is correct. This is due to the semaphores protecting the shared memory resource.