Project 1: Shared Memory

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

***Abstract*— Using shared memory across multiple processes without atomic operations.**

I. INTRODUCTION

In this project we use shared memory across four separate child processes forked from one parent process. Then we wait for child processes to complete and print that it has ended. We do this without any atomic operations which leads to memory access collisions.

II. FUNCTIONS USED & OUTPUT

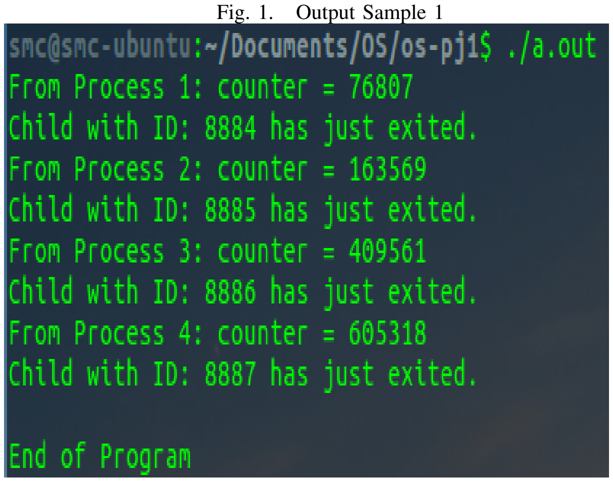
A. Functions Used

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 (process1-4). 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.

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 programmer also cannot prevent memory access collisions without the use of atomic operations. An example is a location in memory contains a value of zero, if two processes are trying to increment this value by one at the same time, both processes read in the memory value of zero, increment it by one, and put it back into memory. So instead of the value being two, it is only one. While the shared memory integer value should be 1.1 million by the end, it usually is no where near that value. These problems can be solved through atomic operations which are hardware level instructions. Here are some examples of the output.

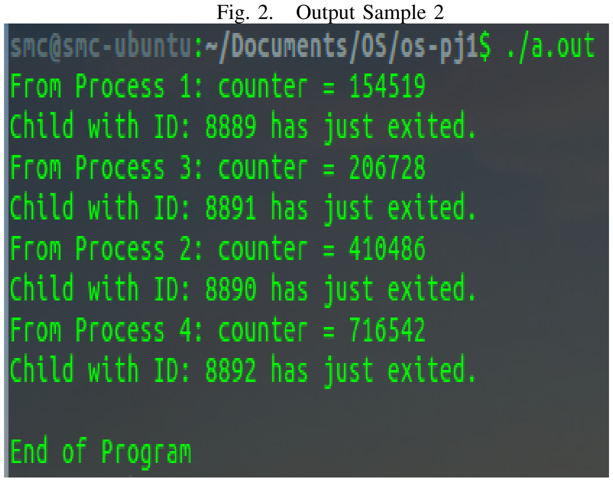
Fig. 1. Output Sample 1



```
smc@smc-ubuntu:~/Documents/OS/os-pj1$ ./a.out
From Process 1: counter = 76807
Child with ID: 8884 has just exited.
From Process 2: counter = 163569
Child with ID: 8885 has just exited.
From Process 3: counter = 409561
Child with ID: 8886 has just exited.
From Process 4: counter = 605318
Child with ID: 8887 has just exited.

End of Program
```

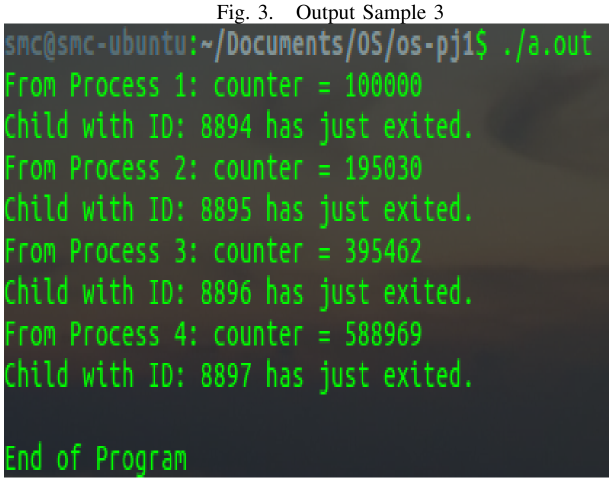
Fig. 2. Output Sample 2



```
smc@smc-ubuntu:~/Documents/OS/os-pj1$ ./a.out
From Process 1: counter = 154519
Child with ID: 8889 has just exited.
From Process 3: counter = 206728
Child with ID: 8891 has just exited.
From Process 2: counter = 410486
Child with ID: 8890 has just exited.
From Process 4: counter = 716542
Child with ID: 8892 has just exited.

End of Program
```

Fig. 3. Output Sample 3



```
smc@smc-ubuntu:~/Documents/OS/os-pj1$ ./a.out
From Process 1: counter = 100000
Child with ID: 8894 has just exited.
From Process 2: counter = 195030
Child with ID: 8895 has just exited.
From Process 3: counter = 395462
Child with ID: 8896 has just exited.
From Process 4: counter = 588969
Child with ID: 8897 has just exited.

End of Program
```

III. Conclusions

As you can see from the figures above, the count value is way off. This is due to no protection of the shared memory along with memory access collisions. I believe these can be fixed using atomic operations.