Project 1 Report

Daniel Yowell
113558774
CS3113
Dr. Song Fang
2/23/2023

***Abstract*—This project demonstrated the basics of forking processes and creating shared memory in C, particularly the importance of protecting shared memory to prevent future issues.**

Keywords—C, process, shared memory, fork, wait

I. Overview

Our goal for this project was to create an integer variable "total" which could be shared between four different child processes, each of which would call a function to increment the variable by a certain amount. Each time a child process was created, the parent process would wait until its child finished executing before continuing its own code.

II. Concepts

A. Process IDs, fork(), and wait()

Each process is associated with an ID stored as an integer value. When fork() is called, the process is split into two processes, a parent and a child, which both execute the remaining code simultaneously. However, by calling wait() in the parent process, we can pause the parent's execution until it receives a signal that a child process has finished execution. The wait() function also returns the process ID of the terminating child process, which the parent can store and print to output (such as in this project).

By default, all data is duplicated when a process is forked. This means changing a variable in a parent process will not change the variable in its child process, and vice versa. However, by allocating a region of "shared memory," each process can modify the same variables in real time.

B. Shared Memory

In researching shared memory online, I discovered several methods of sharing memory between processes in C, such as the mmap() function and the POSIX API. I even successfully implemented some of these workflows with the help of online guides. However, I decided to base my final program off of the class lecture slides, which utilize the functions shmat(), shmdt(), and shmctl(), among others.

It is important to detach and delete shared memory after each program execution, otherwise the data will remain stored in the computer system indefinitely unless it is deleted from the command line.

III. Analysis

I was not able to exactly replicate the sample output specified in the project instructions. However, I still managed to implement every required feature, and the final build of my program successfully compiled and ran when tested on once of OU's Linux servers.

Below is an example output:

```
From Process 1: counter = 100000.
Child with ID: 12898 has just exited.
From Process 2: counter = 200000.
Child with ID: 12899 has just exited.
From Process 3: counter = 300000.
Child with ID: 12900 has just exited.
From Process 4: counter = 500000.
Child with ID: 12901 has just exited.

End of simulation.
```

The first thing I noticed during testing was that the process IDs of each child seem to increment after each fork. I deduced that when creating a new process, the child process receives the next available ID in increasing order. By extension, I speculated that the process ID of the earliest parent would be 1 less than its first child (such as 12897 in the run shown above). I later confirmed this theory by calling getpid() in the parent process during a test run.

I also noticed the process IDs continually growing larger after each execution, with the starting ID of a new run being a few dozen integers larger than the last child of the previous run. I concluded that this process of assigning IDs must be universal for every process in the computer system.

IV. Conclusion

My program successfully accomplished each task specified in the project instructions, albeit not exactly matching the sample output. The process of writing this program was a valuable crash course in allocating shared memory in C, while also demonstrating the importance of protecting and deleting this data properly.