Programming Assignment 1 – Shared Memory

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

I. Introduction/Project Overview

This project's primary objective was to introduce the use of shared memory, allowing students to visualize how processes interact with one another. Students were asked to create a program that creates and connects to a shared memory segment, creates four child processes, outputs their PIDs and at what moment they finish running, then disconnects and releases the shared memory. By doing this, the students were introduced to hands-on creation of shared memory and the importance of it being adequately protected.

II. Finding a Solution to the Problem

The main problem that arose in the programming project was figuring out the most efficient way to create four child processes and have them all access the same shared memory segment at the same time. Each process needed to increment the shared value, "total", at the same time while they were running within the parent process. When approaching this issue, I first created a program that seemed to work but only incremented shared memory one process at a time. This was an issue, as the memory was not being accessed by all processes simultaneously. Instead of the four if-statements I originally had, I switched to a for-loop and checked if each "fork()" was returning 0. If so, I called a function and began incrementing the shared memory until the process was finished.

```
//Create process for each pid and increment total as needed (with function)
for (int i = 0; i < 4; i++) {
    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }
    if (pid == 0) {
        processInc(pidLimits[i], i+1);
    }
}
```

Figure 1

```
//Function to increment "total" based on individual process limits
void processInc(int numToIncrement, int num) {
    for (int i = 1; i <= numToIncrement; i++) {
        total->value++;
    }
    printf("From Process %d: counter = %d.\n", num, total->value);
    exit(0);
}
```

Figure 2

Creating a for-loop to create four processes, then calling a function to increment "total" based on incremental values given by the assignment (i.e., 100,000, 200,000, and so on) allowed the processes to run efficiently while all accessing the shared memory at the same time. We can see in the figures (3.1-3.3) below that after the program begins, the total is incremented, and each process executes and terminates before the end of the parent process.

```
[patt0233@gpel8:~$ ./OPsP1.exe
Successfully created shared memory.
Successfully connected to shared memory.
From Process 1: counter = 105601.
Child with ID: 16611 has just exited.
From Process 2: counter = 222160.
Child with ID: 16612 has just exited.
From Process 3: counter = 310795.
Child with ID: 16613 has just exited.
From Process 4: counter = 477286.
Child with ID: 16614 has just exited.
Successfully detatched from shared memory.
Successfully deleted shared memory.
End of simulation.
```

Figure 3.1

```
patt0233@gpel8:~$ ./OPsP1.exe
Successfully created shared memory.
Successfully connected to shared memory.
From Process 1: counter = 86600.
Child with ID: 16624 has just exited.
From Process 2: counter = 221009.
Child with ID: 16625 has just exited.
From Process 3: counter = 315570.
Child with ID: 16626 has just exited.
From Process 4: counter = 520437.
Child with ID: 16627 has just exited.
Successfully detatched from shared memory.
Successfully deleted shared memory.
End of simulation.
```

Figure 3.2

```
patt0233@gpel8:~$ ./OPsP1.exe
Successfully created shared memory.
Successfully connected to shared memory.
From Process 1: counter = 100000.
Child with ID: 16594 has just exited.
From Process 2: counter = 244516.
Child with ID: 16595 has just exited.
From Process 3: counter = 403606.
Child with ID: 16596 has just exited.
From Process 4: counter = 540596.
Child with ID: 16597 has just exited.
Successfully detatched from shared memory.
Successfully deleted shared memory.
End of simulation.
```

Figure 3.3

Another issue I faced was figuring out how to make sure the parent would wait for each process to execute before terminating. After a few short test runs and reading the assignment closely, I decided on creating a for-loop with four increments for each child process. Within the for-loop, I waited for each process to finish, then output their completion.

```c
//Wait for children to finish running
for (int i = 0; i < 4; i++) {
    pid_t temp_pid = wait(NULL);
    printf("Child with ID: %d has just exited.\n", temp_pid);
}
```

Figure 3.4

III. Post-Execution Discussion

In the above outputs we can recognize that the counter (which is the shared memory) varies in value from execution to execution. Despite the code remaining the same and the parent process still creating four child processes, both the PIDs of the children and the speed at which they increment “total” changes. Since the PIDs are increasing incrementally, we can conclude their value is based on the previous PID- they increment with each new process creation. How the shared memory is affected is much different. In figure 3.1, the shared memory is first printed as 105,601 and printed last at 477,268. However, in figure 3.2, the counter is first printed at 86,600 and printed last at 520,473. These numbers begin and end with completely different values.

After examination, we can observe that varying “total” amounts is due to a couple of reasons. Firstly, forking to create child processes works differently from system to system. I experienced this while getting different results on my own machine versus on the

Gpel machine. Secondly, we know that multiple children are attempting to access and modify one segment of shared memory at the same time. This is called “race condition”, which causes a variance in the value of “total” when output each time. Whichever process accesses the memory first, increments it first. The process that first accesses the memory is not guaranteed to be the first-created child each time. The order is unpredictable and may vary.

IV. Conclusion

Overall, the project helped me learn about shared memory, how it works, and its unpredictability when faced with race conditions. I will move forward in programming with knowledge of child processes, child-child process relations, child-parent relations, and shared memory access.