# CS 474 Project #1: Exploring the Implications of the Use of Shared Memory in Cooperating Processes

Liliana Aguirre Esparza / NMSU
Department of Computer Science
Las Cruces, NM
lili91@nmsu.edu

Jose Franco Baquera / NMSU
Department of Computer Science
Las Cruces, NM
jose5913@nmsu.edu

***Abstract* – Shared memory among several processes working concurrently can lead to race conditions and unexpected results if the memory is not protected adequately. A short program was created in order to explore the implications of unprotected shared memory. The program created four separate child processes and made them modify a shared data counter. This paper explores, discusses, and interprets the results produced by this short program.**

***Keywords – Shared memory; race conditions; shared data***

## I. INTRODUCTION

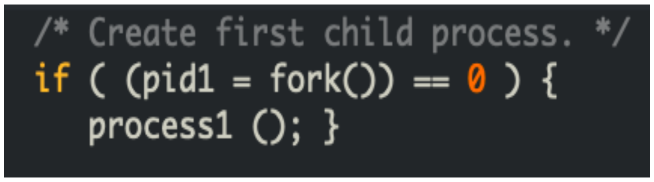
Processes are the "unit[s] of work in a modern time-sharing system" [1]. In most modern-day systems, processes must work concurrently and, in many cases, cooperate with each other to get some job done. Cooperating processes must share data and communicate through shared memory (or message passing), which means that one process can affect the outcome of the other. In a system that allows for process cooperation, it is vital to make sure that shared data is modified correctly by each process and shared memory is accessed properly. When not protected adequately, shared memory can lead to inconsistencies and issues in any computer system. Most noticeably, unprotected shared memory can lead to a race condition, which is a "situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place" [1]. The project aims to allow students to learn more about the concept of shared memory and some of the issues that may result from race conditions.

## II. METHODOLOGY

To gain an understanding of the implications and complications of shared memory, a complete C program was written implementing the creation of four cooperating processes. The program was composed of a header file and a main C program.

The header file consists of several important components. First, all necessary libraries were included. Some of these are standard libraries such as *stdio.h* and *stdlib.h*, while others are system libraries necessary to implement shared memory inter-process communication using the appropriate system calls, such as *sys/shm.h* and *sys/ipc.h*. Once the libraries were included, a shared memory key was defined. In the program this key will be used to identify shared memory segments, and each process created during the execution of the program will use it to access the shared memory segment. After the definition of the shared memory key, the header file defines a structure that represents a shared memory segment and consists of a single integer variable. Finally, a single shared memory pointer was defined; this is the shared memory segment that will be accessible to all processes created when the program is executed, and the integer value in the segment will later be modified by each process.

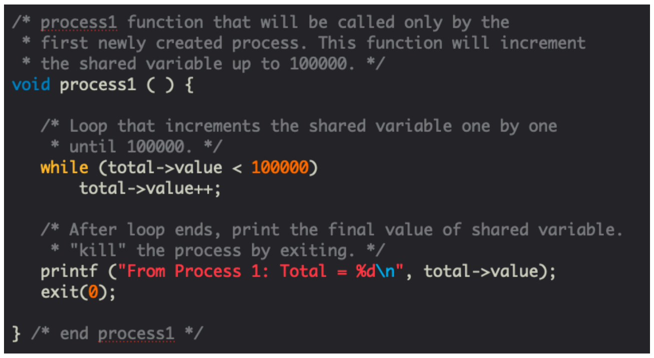
The project also contains a main C program. This particular program first allocates a shared memory segment using the *shmget* system call and the shared memory key declared in the header file. After this memory segment is allocated, the *shared_mem* pointer declared in the header file is attached to the allocated memory address space. The program then initializes the counter stored in shared memory to zero and creates a total of four processes using four separate if statements and *fork()* system calls. Fig. 1 shows how process 1 is created. When the program calls *fork(),* it assigns the return value to a variable. The return value stored in the variable is then compared to zero, and if this condition is true, the function *process1()* will be called. In contrast, the initial parent process will not call the *process1()* function, but instead continue to execute the following statement and create a new child. This is because the *fork()* system call returns zero only to the newly created child process. The remaining three children processes are created in a similar manner as displayed in Fig. 1. However, it is important to note that each child process calls a unique function (i.e. the first child process will only call function *process1()*, the second child process will only call function *process2()*, and so on).



```
/* Create first child process. */
if ( (pid1 = fork()) == 0 ) {
   process1 (); }
```

Figure 1 – Sample code of how process 1 is created.

Because each newly created child process must call a separate and unique function, the program includes a definition for four different functions. Fig. 2 demonstrates how function *process1( )* was implemented. In the function definition, a while loop is used to increment the shared variable by one on each iteration until the value 100,000 is reached. Once the while loop ends, the function prints the final value of the shared variable and calls the *exit()* function to terminate the child process. The other three *process* functions are also called by their associated child processes only and are implemented in a similar way. However, each loop in each function will terminate at different values (i.e. *process1* will count to 100,000, *process2* will count to 200,000, *process3* will count to 300,000, and *process4* will count to 500,000).



```
/* process1 function that will be called only by the
 * first newly created process. This function will increment
 * the shared variable up to 100000. */
void process1 ( ) {

    /* Loop that increments the shared variable one by one
     * until 100000. */
    while (total->value < 100000)
        total->value++;

    /* After loop ends, print the final value of shared variable.
     * "kill" the process by exiting. */
    printf ("From Process 1: Total = %d\n", total->value);
    exit(0);

} /* end process1 */
```

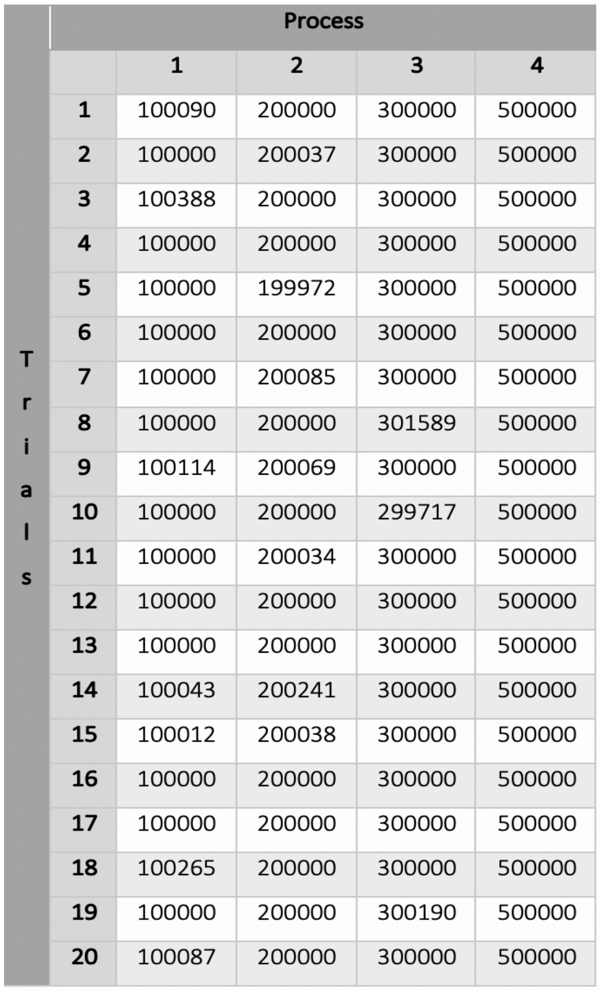
Figure 2 – Sample code of the function called by process one.

After the four child processes are created, the original parent process will wait until all of them have finished executing by using a while loop and the *wait()* function. As each child process finishes execution, the parent process will print the process id of each child. Finally, the parent process will release all shared memory, print a meaningful message, and terminate.

In order to observe the effects of cooperating processes modifying data in a segment of shared memory, the program was executed a total of 20 times. For these 20 trials, the value of the shared variable was recorded after it was modified by each created process. The results for all trials are discussed in the following section.

## III. RESULTS AND DISCUSSION

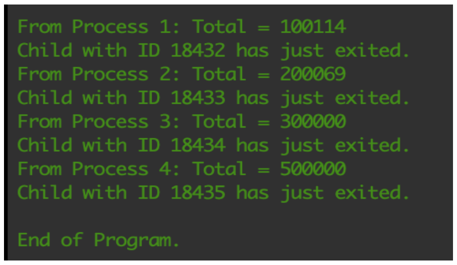
Table 1 lists the results of the 20 trial runs that were conducted on the computer science lab machines. In most trials, the four child processes correctly modified the shared data and terminated after the correct cutoff number (i.e. 100,000, 200,000, 300,000, or 500,000). However, it was observed that several other trial runs contained slight variations in the manipulated data that are relevant to this project's objective. For example, as Fig. 3 illustrates, it was observed that in trial 9, process 1 printed 100,114 for the value in the shared memory segment instead of the correct value of 100,000. These data variations were observed in several other trials for processes 1, 2 and 3. It is important to note that process 4 printed the correct value of 500,000. These discrepancies in the observed data occurred because the memory segment shared among all four processes was not



| | Process | | | |
|---|---|---|---|---|
| Trials | 1 | 2 | 3 | 4 |
| 1 | 100090 | 200000 | 300000 | 500000 |
| 2 | 100000 | 200037 | 300000 | 500000 |
| 3 | 100388 | 200000 | 300000 | 500000 |
| 4 | 100000 | 200000 | 300000 | 500000 |
| 5 | 100000 | 199972 | 300000 | 500000 |
| 6 | 100000 | 200000 | 300000 | 500000 |
| 7 | 100000 | 200085 | 300000 | 500000 |
| 8 | 100000 | 200000 | 301589 | 500000 |
| 9 | 100114 | 200069 | 300000 | 500000 |
| 10 | 100000 | 200000 | 299717 | 500000 |
| 11 | 100000 | 200034 | 300000 | 500000 |
| 12 | 100000 | 200000 | 300000 | 500000 |
| 13 | 100000 | 200000 | 300000 | 500000 |
| 14 | 100043 | 200241 | 300000 | 500000 |
| 15 | 100012 | 200038 | 300000 | 500000 |
| 16 | 100000 | 200000 | 300000 | 500000 |
| 17 | 100000 | 200000 | 300000 | 500000 |
| 18 | 100265 | 200000 | 300000 | 500000 |
| 19 | 100000 | 200000 | 300190 | 500000 |
| 20 | 100087 | 200000 | 300000 | 500000 |

Table 1 - Results of the 20 trial runs.

protected, which led to a race condition when the program was executed. A race condition may happen when a process that is executing on the CPU is interrupted and replaced by another process that will also manipulate the same data in the shared memory segment. In the case of this program, each process has a different condition for termination. Therefore, process 1 may increment the data to 100,000, which is the termination condition for that particular process, but may be interrupted before the condition is checked, giving another process the opportunity to increment the shared memory value. Then, the scheduler may allocate the CPU to process 1 again, at which point the termination condition will be met. However, since a different process has accessed and modified the shared data, the value printed by process 1 will not be the correct one.



```
From Process 1: Total = 100114
Child with ID 18432 has just exited.
From Process 2: Total = 200069
Child with ID 18433 has just exited.
From Process 3: Total = 300000
Child with ID 18434 has just exited.
From Process 4: Total = 500000
Child with ID 18435 has just exited.

End of Program.
```

Figure 3 – Screenshot of output for trial 9

The 5-number summary was calculated for each process's final shared value over the 20 trials performed. These results are depicted in Table 2. We note that the median for all four processes is equal to the *correct* value of either 100,000, 200,000, 300,000, or 500,000. This implies that a single execution of the program is more likely to print a correct result than an incorrect one. However, the previously discussed race condition may still ensue depending on how the operating system executes each individual child process.

| | Process 1 | Process 2 | Process 3 | Process 4 |
|---|---|---|---|---|
| **Minimum** | 100000 | 199972 | 299717 | 500000 |
| **Q1** | 100000 | 200000 | 300000 | 500000 |
| **Median** | 100000 | 200000 | 300000 | 500000 |
| **Q3** | 100054 | 200034.8 | 300000 | 500000 |
| **Maximum** | 100388 | 200241 | 301589 | 500000 |

Table 2 - Five number summary for each child process.

Figs. 4 through 7 illustrate the 20 trial runs for each child process and the final value of the shared data once the created process terminates. When comparing all charts, several evident patterns emerge. For instance, child processes 1 and 2 had the wrong shared memory value at termination more often than processes 3 and 4. However, there seemed to be more variation and incorrect final values in process 2 than in process 1. It is worth noting that there was much less variation in the shared data values in process 3 and furthermore, process 4 always ended with the correct shared data value; thus, its associated line graph is a horizontal line centered at 500,000. The reason for this may be that processes 2, 3, and 4 are all executing throughout the whole lifetime of process 1, therefore, the shared memory value is being modified by all four processes. A similar scenario happens through the lifetime of process 2 since the shared memory value is being modified by all 4 processes until process 1 terminates. Following this logic, process 3 experiences less variation because it has a longer lifetime than processes 1 and 2. This means that, at some point, the shared memory value will only be modified by processes 3 and 4. Finally, process 4 will be running on its own for several loop iteration, which is why the shared memory value will always be correct.

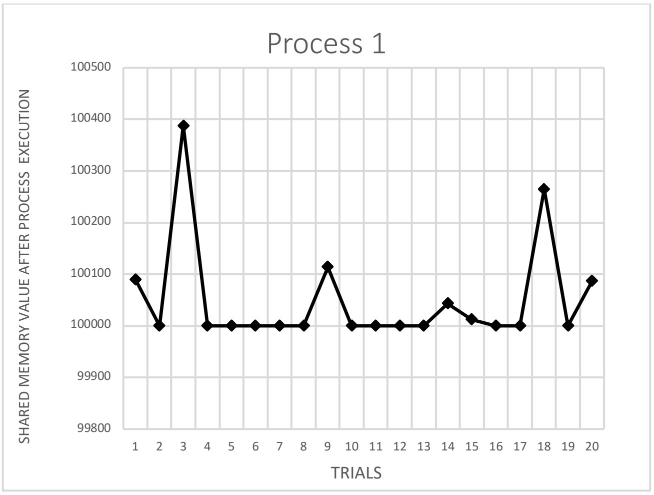


Figure 4 - Graph depicting the variations in the values of shared memory data for process 1.

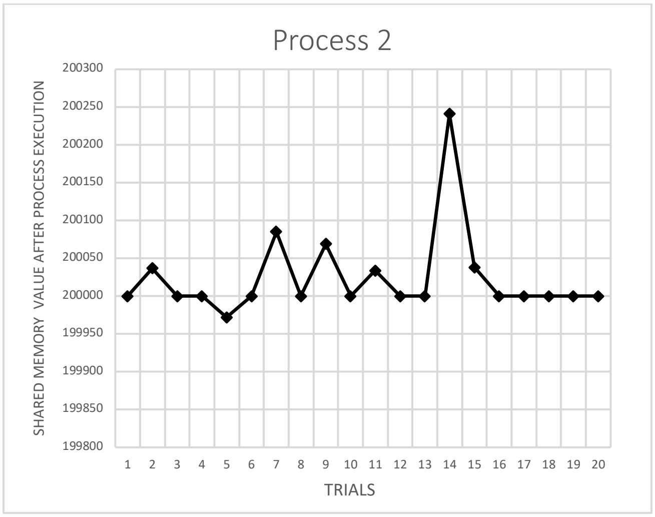


Figure 5 - Graph depicting the variations in the values of shared memory data for process 2.

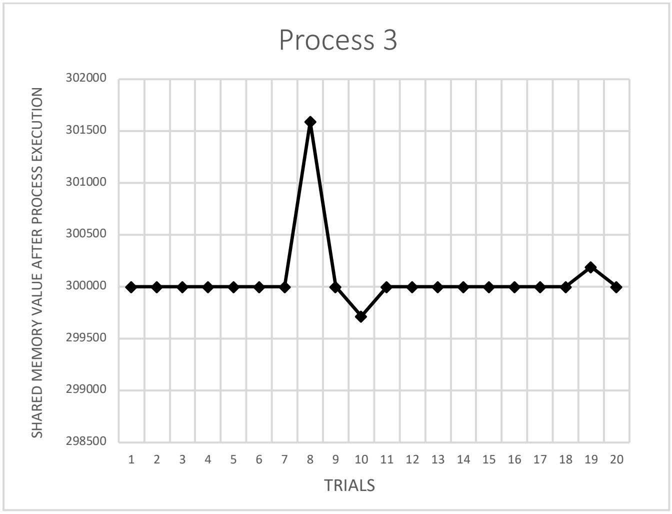


Figure 6 - Graph depicting the variations in the values of shared memory data for process 3.

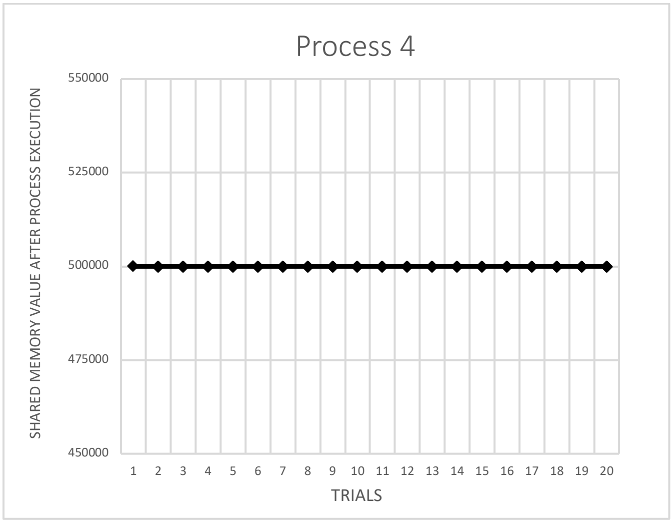


Figure 7 - Graph depicting the variations in the values of shared memory data for process 4.

## IV. CONCLUSION

In conclusion, based on the observations and results gathered, it can be determined that using shared memory as a means of interprocess communication between cooperating processes can lead to data inconsistency and erroneous results. Given that most present-day systems allow for process cooperation, even small inconsistencies in shared memory data may bring a system to its knees and create errors that are difficult to trace.

## V. REFERENCES

[1] S. Abraham, G. Peter, and G. Greg, *Operating System Concepts.* Hoboken, NJ: John Wiley and Sons.