OS Project 2: Shared Buffer and Semaphores

Marco Salazar
Department Of Computer Science
New Mexico State University
Las Cruces, NM, USA
marcoams@nmsu.edu

ABSTRACT

In this project the purpose was to learn how to use semaphores to protect a limited size resource. It built upon the previous project that dealt with the protection of shared memory. As such the classic consumer and producer circular buffer problem was implemented. In this case the two threads shared the circular buffer and took turns pushing and popping values from the array, filling it with data from a file "mytest.dat"

From this I found that adequate semaphore protection can be achieved quite simply for critical sections. Through the use of three semaphores I was able add and read from the buffer without corrupting any data and preserving the invariants of the problem. As long as the processes are implemented correctly so as to avoid deadlock, there will not be a problem.

KEYWORDS

Processes, Protection, Shared Memory, Semaphores, Threads.

1 Process

The shared memory was created through a simple procedure. A struct was created and defined that stored a 15-character circular buffer, and a global variable was made of that type. It was initialized in the main function with shmeat, and the result of shmeat. Once this shared memory was made, the semaphores were initiated such that the first one indicated 15 spots available, and the other listed non as ready to read, while the last one served as the mutex of the critical section. The two threads were then created in the parent process and joined. Once the threads finished the shared memory was released so that there would not be and lingering memory to slow down the computer, and the semaphores and threads were destroyed.

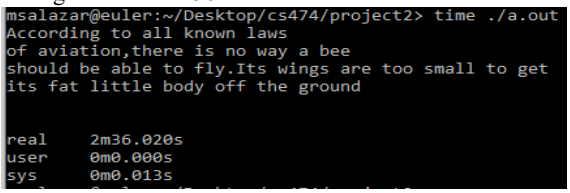
The producer thread simply opened up the file "mytest.dat" and scanned in every character in a while loop. Inside the while loop the available semaphore was called with wait and the critical section was locked, then the buffer was filled with the next character. Afterwards the pointer to the next position is incremented with modular arithmetic, and the critical section is released, and the full semaphore was signaled. Once the entire file was read, the producer put one last "*" character in the buffer to signal the end of the transmission.

The consumer thread contained an infinite while loop. Inside it waited for the readable semaphore to get a value so that it could enter into the critical section. When inside the critical section it copied the value and then performed a check. If the character was a "*" then it broke out of the loop. Otherwise, it printed out the character and incremented the pointer with modular arithmetic. It then signaled the critical section and that more space was available. At the end of the loop it would sleep for 1 second.

2 Results

The following results were found when executed on the NMSU CS department lab computers that run at 1920 MHz on average based on the lscpu command. Through a trial of ten runs of a file with 11 characters ("11 seconds ") it took an average total of 12 seconds to finish computing.

However, when I increased the character count to 150, it took an average of around 156 seconds:



3 Analysis

From what I could see, it seemed that the one trade off for keeping data from being corrupted when shared, was a time tradeoff. For the first test I expected it to only take 11 seconds, but it took 12. While this is still better from the 22 seconds I would have expected if both threads had a second sleep in it and were running not in parallel, it was still interesting to observe the slowdown

Altogether, this shows that it is necessary to lose some time for concurrency, but this is a needed sacrifice. It is much better to wait and guarantee that the data is safe, then to go fast and corrupt the data along the way.