Assignment 4

Kernel Synchronization Primitives

Structs used, Logic & Working

To Implement this Producer Consumer Problem, I have utilized the queue data structure, and made a queue of fixed size of length 5 using the kmalloc() function. The queue stores the 8-byte blocks in the form of unsigned long integers (since unsigned long is also 8 bytes). To implement the queue inside the kernel, I have used the array implementation of the queue for enqueuing and dequeuing.

All of the following codes and changes take place in the kernel/sys.c file:

To create the semaphores, I used the struct semaphore that is already defined in the kernel.
3 kernel semaphores have been used for this implementation. These are ‘empty’, ‘full’ and ‘mutex’. These 3 counting semaphores would be initialized with counts of 5 (size of queue buffer), 0 and 1 respectively. The functions from the Kernel Semaphore APIs have been utilized such as sema_init(), up(), down(), etc.

To add a syscall for ‘writer’, I used the SYSCALL_DEFINE1() macro, which takes the 8 byte data that is to be enqueued as its parameter. In case the kernel queue has not been initialized yet (i.e is null), then it also initializes the queue with kmalloc(), as well as the required semaphores along with it. Similarly, I used the SYSCALL_DEFINE0() macro for the ‘reader’ syscall, which also initializes the queue and the semaphores, if not already initialized.

The ‘writer’ syscall calls the enqueue() function and passes the 8 byte data to be enqueued as a parameter to the function call. The enqueue() function is responsible to enqueue the data using proper kernel synchronization primitives to ensure no racing conditions, etc. Hence, the enqueue function first performs down(&empty) and down(&mutex), to make sure that the queue isn’t full and there is no mutex lock present, so that it can proceed to safely put the data in the queue. After successfully enqueuing, up(&mutex) and up(&full) is performed so that it releases the locks or increases the counts of the semaphores that it had acquired.

Similarly, the ‘reader’ syscall calls the dequeue() function. The dequeue() function is responsible to dequeue the data using proper kernel synchronization primitives to ensure no racing conditions, etc and return the dequeued data back. Hence, the dequeue function first performs down(&full) and down(&mutex), to make sure that the queue isn’t empty and there is no mutex lock present, so that it can proceed to safely remove the data in the queue. After successfully dequeuing, up(&mutex) and up(&empty) is performed so that it releases the locks or increases the counts of the semaphores that it had acquired. The dequeued value is then returned back to the ‘reader’ syscall, which further returns it back to the user.

In the producer’s test program, the random 8 bytes are read from dev/urandom, and I have converted them into an unsigned long representation (since each 8 byte data block can also be represented in a 8 byte unsigned long integer).

In both the test programs, each producer and consumer program would try to read and write by calling the respective syscalls. Each program would produce/consume 500 times. Both programs run simultaneously, and proper output is shown, implying no data loss or corruption, and hence, the proper use of the synchronization primitives.