Oregon Institute of Technology
Software Engineering Technology, Wilsonville Oregon

CST 352 - Operating Systems
Lab 2 – Producer-Consumer Problem

Design and implement three solutions of the Producer-Consumer problem – the first version using spinlocks (no mutual exclusion), another one using semaphores for synchronization of critical sections between the producer and consumer, and the third version that includes a mutex for mutual exclusion of shared data values among threads.

The final program will create 4 producer and 3 consumer threads. The job of the producer will be to generate a random number and place it in a bound-buffer. The role of the consumer will be to remove items from the bound-buffer and print them to the screen. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. Sounds simple right? Well, yes and no. We must address 2 primary issues here: one is concurrency issues between the producer and the consumer, and the other is synchronization among the threads when accessing shared data, so we'll address each of these issues independently by writing our code incrementally.

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution 1

Our first solution will be simple and use a spin lock, which is just continuously checking for some condition to be satisfied. The Producer thread should generate a random number. It should check to see if there is room in the buffer. If so, it should store it in the buffer and display it on standard output, then increment the buffer count. The Consumer should check if the buffer is empty. If not, then remove the next item from the buffer and print the value on standard output, then decrement the counter.

If the buffer is full, the producer will "spin" waiting for the consumer to remove an item. In the same way, the consumer will spin if it finds the buffer to be empty.

What are the issues with this solution? Write them up as part of Solution 2.

Solution 2

We need a more efficient way of synchronizing the Producer and Consumer processes. We will introduce a semaphore . The semaphore provides an automatic way to communicate between the proceses so they can simply wait or sleep until some condition is met.

We will use two semaphores, fillCount and emptyCount, to solve the problem. fillCount is incremented and emptyCount decremented when a new item has been put into the buffer. If the producer tries to decrement emptyCount while its value is zero, the producer is put to sleep. The next time an item is consumed, emptyCount is incremented and the producer wakes up. The consumer works analogously.

Solution 3

What if we have multiple producers and consumers? We need a way to make sure that only one producer is executing putItemIntoBuffer() at a time. In other words we need a way to execute a critical section with mutual exclusion. To accomplish this we use a binary semaphore called mutex. Since the value of a binary semaphore can be only either one or zero, only one process can be executing between pthread_mutex_lock and pthread_mutex_unlock.

Testing:
You can start with one producer thread and one consumer thread for testing, and gradually use more producer and consumer threads. For each test case, you need to make sure that the random numbers generated by producer threads should exactly match the random numbers consumed by consumer threads (both their orders and their values).

Getting started:
You can use the provided sample lab Lab2-unsynchronized.c as a starting point for your solutions, or you can implement your own from scratch.

- **NB 1**: The provided code will require modification to complete without error in the final version.
- **NB 2**: You need to link two special libraries to provide multithreaded and semaphore support: the pthread library for any program that uses pthreads and the runtime library rt for any program that uses semaphores