Operating Systems II : CS3523
Spring 2019

Programming Assignment 3 : Solving Producer Consumer Problem using Semaphores and Locks

Instructor : Dr. Sathya Peri
Author : Puneet Mangla (CS17BTECH11029)

Design of Program :

We implemented and comapred **Bounded buffer producer-consumer problem** using semaphores and locks.

Using Mutex Locks

Two mutex lock are used to protect the shared variables like count , BUFFER_SIZE to be accessed by two threads simuntaneously.

Producer :

```
while(true) {
	// produce an item and put in the buffer.
	while (true){
		mtx2.lock();
		if (count < BUFFER_SIZE){
			count++;
			break;
		}
		mtx2.unlock();
	}
	mtx.lock();
	mtx2.unlock();
	// increment  in and add item in buffer
	mtx.unlock();
}
```

Consumer :

```
while(true) {
    while (true){
        mtx2.lock();
        if (count > 0){
            count--;
            break;
        }
        mtx2.unlock();
    }
    mtx.lock();
    mtx2.unlock();
    //  increment out and take tem out from buffer
    mtx.unlock();
    // consume the picked item
}
```

Using Semaphores

Three semaphores are used as follows :

```
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Producer :

```
while (true) {
    /* produce an item in next produced */
    wait(empty);
    wait(mutex);
    /* add next produced to the buffer */
    signal(mutex);
    signal(full);
}
```

Consumer :

```
while (true) {
    wait(full);
    wait(mutex);
    /* remove an item from buffer to next consumed */
    signal(mutex);
    signal(empty);
    /* consume the item in next consumed */
}
```

Comparison metrics :

1. **Average waiting time :** the average time taken by a thread to enter the CS.

Algorithm :

1. The main thread reads the "inp-params.txt" and stores all the parameters as global for all theads to be accessible.
2. n_p producer threads and n_c consumer threads are created and their id are stored in array pthread_t producer_tid[n_p] and consumer_tid[n_c] respectively.
3. default_random_engine is used to generate random numbers from exponential_distribution to stimulate that these threads are performing some complicated time consuming tasks. Parameters to the distribution were were $1/\mu_p$ and $1/\mu_c$ respectively and seed was thread id.
4. void* producer/consumer functions stimulate producer and consumer . Their implemntation depends on whether we are using locks or semaphores.
5. For printing logs fprintf/printf are used instead of cout as cout streams causing mixing of logs.

Graphs :

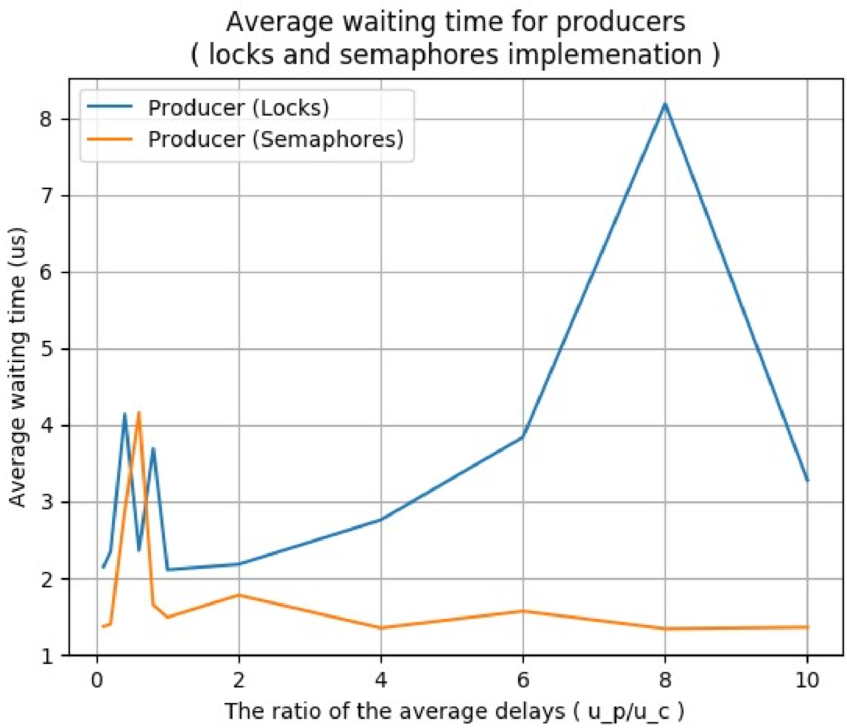
For comparison of spinlock and semaphores in terms of average waiting time for producers and consumers . The x-axis of the graph will consist of ratio of μ_p/μ_c, i.e. the ratio of the average delays of the producer thread to the consumer thread. It will specifically consist of the following 11 values:

10, 8, ... 2, 1, 0.8, 0.6, ... 0.2, 0.1

The y axis consists of average waiting time taken by producers and consumers in both spinlock and semaphore implementation.
For a fair comaprison initial parameters are kept constant to :

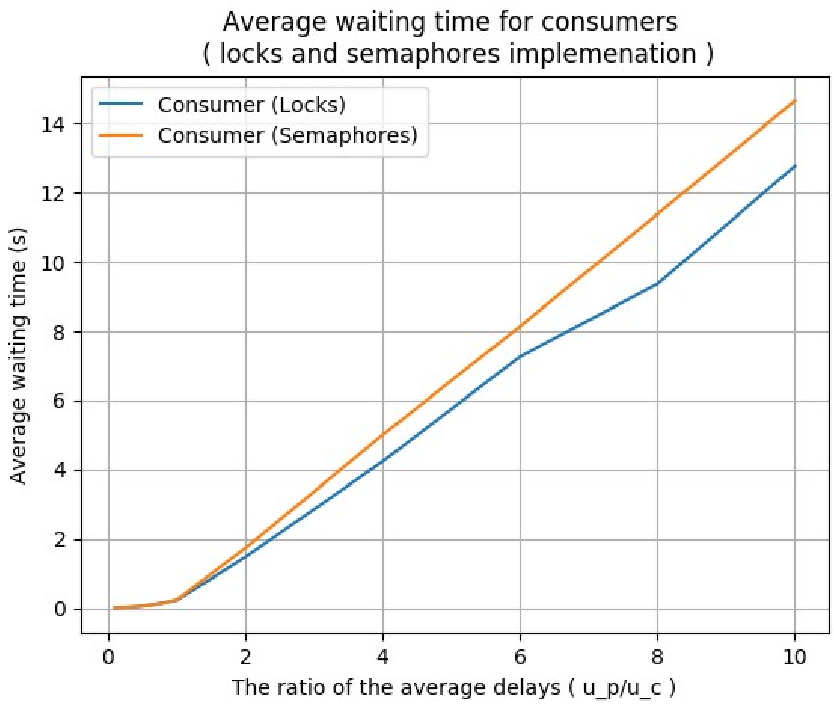
100 10 10 10 10 μ_p μ_c

The ratio μ_p/μ_c is varied and results are noted.



For the above graph (producers) semaphores perform better than locks. The reason is number of producers is same as buffer size , so no producer has to wait . It can always get out of the locks/semaphore wait without waiting/sleeping .

In the locks implementation while one producer is checking the wait condition , other producer will wait (due to `mtx2.lock()`)till first comes out of it adding extra overhead whereas in semaphores first producer can call wait and check the waiting condition without blocking another producer .



In the graph above (consumers) locks perform better than semaphores . The reason is the average delay ratio is increasing causing producers to take more time generating item as compared to time taken by consumers to consume them . This implies conumers will starve as the ratio increases.

In semaphore implementation when consumer waits it goes to sleep causing overhead due to context switch whereas in locks implementaton they will be busy waiting which is less expensive.

Conclusion :

If two processes will not be blocked after checking the wait condition then semaphore `wait` is more better than `mtx.lock()` .

If two processes will be blocked after checking the wait condition then `mtx.lock()` is more better than semaphore `wait` .