Operating Systems - 2
Spring 2019

Instructor - Dr. Sathya Peri
Jatin Chauhan - CS17BTECH11019

Design of Program:

The program has been designed to solve the problem of race condition in producer-consumer situation, with proper synchronization using **semaphores and locks.**
The libraries used in the implementation are -

<pthread.h> - for creating threads and mutex locks
<semaphore.h> - for creating semaphores
<random> - To create random number generating engines.
<unistd.h> - To calculate current system time.
<fstream> - Creating file streams.

The ***producer*** threads produce some item to put up in the ***shared buffer*** which are then consumed by the ***consumer threads.*** Most of the variables are shared thus can create race conditions. The produce should wait whenever the buffer is full, whereas the consumers should wait whenever the buffer is empty. Any item should be placed in the buffer by one producer at a time only, whereas should be consumed by one consumer at a time. Therefore the use of proper synchronization tools in inevitable, lest the data should be misread or may corrupt in case of serious operations.

Implementation of Program:

The implementation of both the programs follow the same structure, except in **lock.cpp,** mutex locks are used, whereas in **sem.cpp** semaphores are used. Threads are created at the start of main function,

along with their attributes. Various producer and consumer threads are created, as the count is taken in form of argument. The *func_producer* function operates over producer threads, while the *func_consumer* operates over consumer threads. The shared variables - **Buffer, in, out , count, capacity** all needs to be saved from race conditions. So these are put in their corresponding sync tools (semaphores or locks here).

Sync_tool is used in rest of the report to signify either ***lock or semaphore.***

Implementation of locks only need one lock to be used by producer or consumer, which is named ***mutex.*** For semaphores, three semaphore objects are used, ***mutex -*** a binary semaphore to be used by producer or consumer, one at a time, ***empty -*** To allow only [SIZE_OF_BUFFER] number of producers to try to acquire the ***mutex, full -*** allowing only [SIZE_OF_BUFFER] consumers to try and acquire ***mutex.***
producer_time, consumer_time - variables to calculate average waiting time for producer and consumer threads respectively.
prod_time_sync_tool, cons_time_sync_tool - used to update the values of ***producer_time and consumer_time*** respectively, since they may also suffer from race conditions.

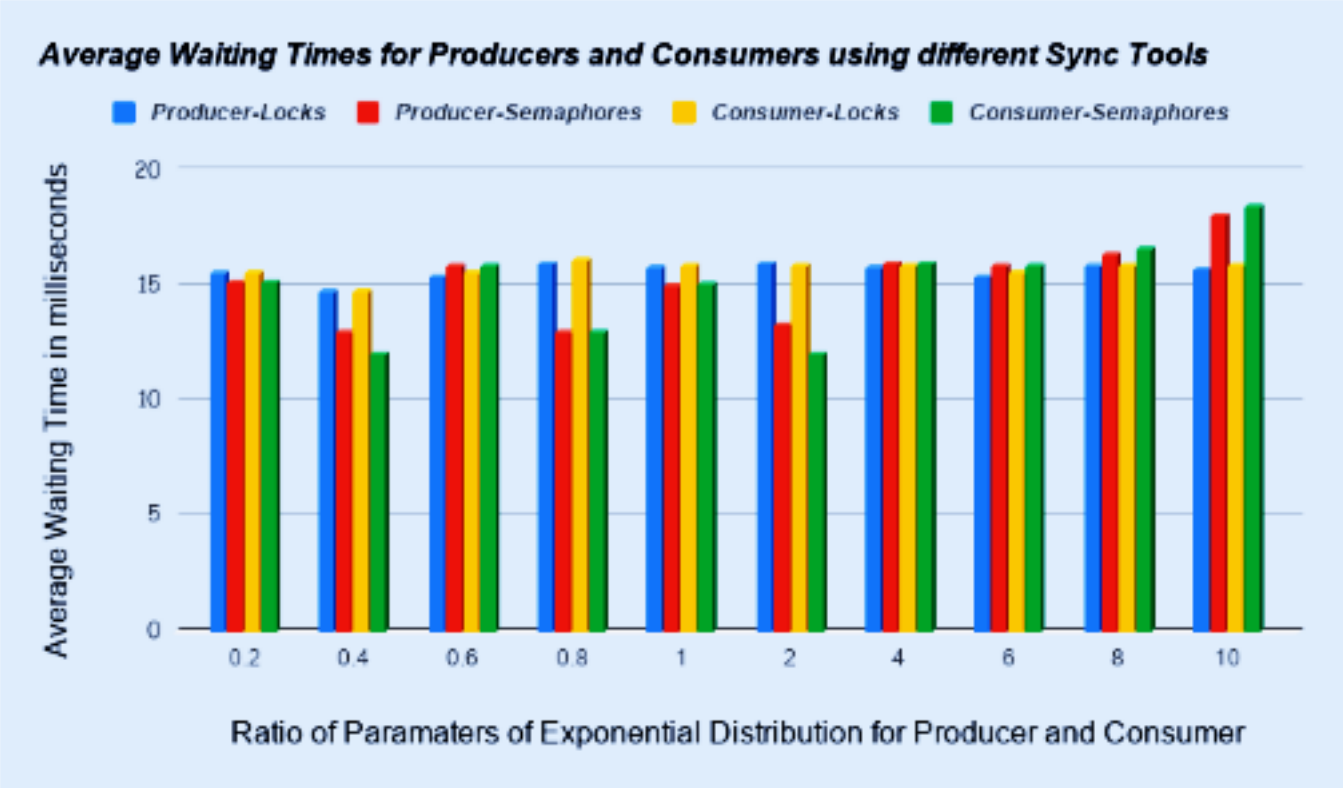
At the start of thread (be it consumer or producer), start time is saved in start_time variable. After finishing current iteration, end time is noted and the difference is added to corresponding time variable, which is used to calculate average waiting over the iterations, over all producers.
During the iteration the producer thread ***busy waits*** if the buffer is already full, in case of ***locks***. In case of semaphores, it busy waits by invoking **wait** on **empty** semaphore, and then **wait** on **mutex.** It then tries to acquire the corresponding mutex to fill the buffer in appropriate position ***in,*** with the produced item (just an integer in this case). The time of production is calculated, then converted to hh:mm:ss format and written to the log

file.The value of ***in*** is then increased along with the value of count, both by one. The sync_tool is then released.
Thereafter the sleep function is called over to signify that the thread is performing some complex operation at that time. The producer thread tries to acquire ***prod_time_sync_tool*** to update the value of ***producer_time.***

Similarly, the consumer thread ***busy waits*** if the buffer is already empty, in case of ***locks***. In case of semaphores, it busy waits by invoking **wait** on ***full*** semaphore, and then **wait** on **mutex.**. Afterwards, it has been filled even by one entry, some consumer thread tries to acquire the corresponding ***mutex***. It reads out from the location in the buffer it needs to. The time of production is calculated, then converted to hh:mm:ss format and written to the log file. Then it updates the value of ***out*** to signify where should it start reading from, the next time. The value of count is decreased. The sync_tool is then released. Thereafter the sleep function is called over to signify that the thread is performing some complex operation at that time. The consumer thread tries to acquire ***cons_time_sync_tool*** to update the value of ***consumer_time.***
Finally, the threads are joined.

Graph:

The graph is drawn for average_waiting_time for producers and consumers (separately) vs the ratio of exponential average factors for producers and consumers. The buffer capacity is fixed to 20, number of producer and consumer threads both to 1000, iterations to average over as 250 for both, and the ratio is then varied.
The y-axis shows average waiting in milliseconds, whereas on x-axis ratio of values of the parameters of exponential distribution are shown.



From the graphs it is evident that the average values of waiting time is less when we use ***locks*** as compared to using ***semaphores.*** One plausible reason may be that semaphore does more complex operations for maintaining atomicity (atomic operations have to be done while - changing the values of semaphore along with busy wait), whereas, for **locks,** simple atomic instructions, like **compare_and_swap** does the work and change the values at hardware level in registers.

Also, in semaphores the value of **waiting_time** seems to **increase** as the ratio **increases,** whereas for locks it seems to **converge** to some finite value.