Operating Systems–2: CS3523 2019 Programming Assignment 3: Solving Producer Consumer Problem using Semaphores and Locks Report

Sai Harsha Kottapalli
CS17BTECH11036

March 8, 2019

1 Aim

The goal of this assignment is to solve the bounded buffer producer-consumer problem using Semaphores and Locks in C++.

2 Design of the Program :

We use Locks and Semaphores to implement the bounded buffer producer-consumer problem.

2.1 Critical Section :

This part of the code should be accessed by only one producer/consumer respectively at a time or as an atomic equivalent. We can avoid race conditions by making sure that no two producers/consumers enter their Critical Sections at the same time.

2.2 Entry Section :

This part of the code is accessible for all the producers/consumers. The job of this code is to make sure only one producer/consumer is given access to the critical section at a particular time.

2.3 Exit Section :

This part of the code is where the producer/consumer has successfully executed the critical section of the program, and will reset the lock so that other producers/consumers can now enter the critical section(if any) while this producer/consumer completed the remainder section parallelly.

2.4 Average Waiting Time :

This is defined as the amount of time each access to critical section takes after requesting for the same.

2.5 Buffer section :

The producer and consumer which are currently in their respective critical section access buffer section such that only one of them is inside this section and in case the (producer tries to produce)/(consumer tries to consume) from buffer when it is full/empty, it gives the access of buffer to the other.

3 Explanation of program - Common Part

3.1 Header files used

- **iostream**
  Header that defines the standard input/output stream objects

- **thread**
  Used for implementing threads.
  It defines the class to represent individual threads of execution.
  A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

- **fstream**
  Input/output stream class to operate on files.
- **random**
  This header introduces random number generation facilities.

- **cstdio**
  Used for **sprintf**, that is, for the format in which the time has to be printed to the log file.
- **unistd.h**
  Used for miscellaneous symbolic constants and types, and declares miscellaneous functions -
  localtime - which is used to get a structure of time, with hours, minutes and seconds.
  time - For capturing current time which is processed by above command.
- **string**
  For processing strings.
- **sys/time.h**
  For using **gettimeofday()** to obtain the time a producer/consumer has to wait to get access through the lock or semaphore.

3.2 getInput()

This is a helper function which helps in keeping the code modular.
It reads the value of the following from the input file, specified in ReadMe.txt

- Capacity of the buffer
- Number of Producers - n_p

- Number of Consumers - _c
- Number of times each producer should produce - rep_p

- Number of times each consumer should consume - rep_c

- lam1 and lam2
  Represents the mean of value of two exponential distribution curves which is to be constructed.

3.3 producer()

The producer function is responsible to produce data for the consumption of consumer.
The producer won't try to add data into the buffer if it's full.

3.4 consumer()

The consumer function is responsible to consume data produced by producer.
The consumer won't try to remove data from the buffer if it's empty.

3.5 currTime()

Takes input time_t structure to generate the time in HH:MM:SS syntax and return this as a string.

3.6 Buffer

We use circular queue to implement the buffer for the producer/consumer problem.

3.7 Other important variables used

- **exponential_distribution**
  Produces random non-negative floating-point values x, distributed according to probability density function defined for exponential distribution about the constant rate given as a parameter.

- **default_random_engine**
  This is a random number engine class that generates pseudo-random numbers.

- **buffer**
  Dynamic array of bool type, based on capacity given in input file.

- **thread *th for producers and consumers**
  th_p and th_c are the pointers for storing the array of n_p and n_c threads respectively. This will be useful for later calling join to properly exit the threads spawned previously.

- **waiting time for producers and consumers** Used to calculate the average time a producer/consumer has to wait to get access through locks or semaphores.

4 Locks

A lock or mutex (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy.
Header file used - **mutex**
The following functions of semaphore in c is utilized to solve the current problem -

- **mutex::lock** If the mutex isn't currently locked by any thread, the calling thread locks it (from this point, and until its member unlock is called, the thread owns the mutex).
  If the mutex is currently locked by another thread, execution of the calling thread is blocked until unlocked by the other thread (other non-locked threads continue their execution).
  If the mutex is currently locked by the same thread calling this function, it produces a deadlock (with undefined behavior).

- **mutex::unlock** Unlocks the mutex, releasing ownership over it.
  If the mutex is not currently locked by the calling thread, it causes undefined behavior.

5 Semaphores

A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment.
Header file used - **semaphore.h**
The following functions of semaphore in c is utilized to solve the current problem -

- **sem_init(sem_t *sem, int pshared, unsigned int value);**
  Initializes the unnamed semaphore at the address pointed to by sem.
  The value argument specifies the initial value for the semaphore.
  The pshared argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

- **sem_wait(sem_ t *sem);**
  decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

- **sem_post(sem_t *sem);**
  increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore.

- **sem_destroy(sem_t *sem);**
  destroys the unnamed semaphore at the address pointed to by sem.

- We use "full" and "empty" semaphores, initialized to 0 and buffer_size respectively for producers and consumers while a binary semaphore for the buffer.

6 Comparision

6.1 Graphs

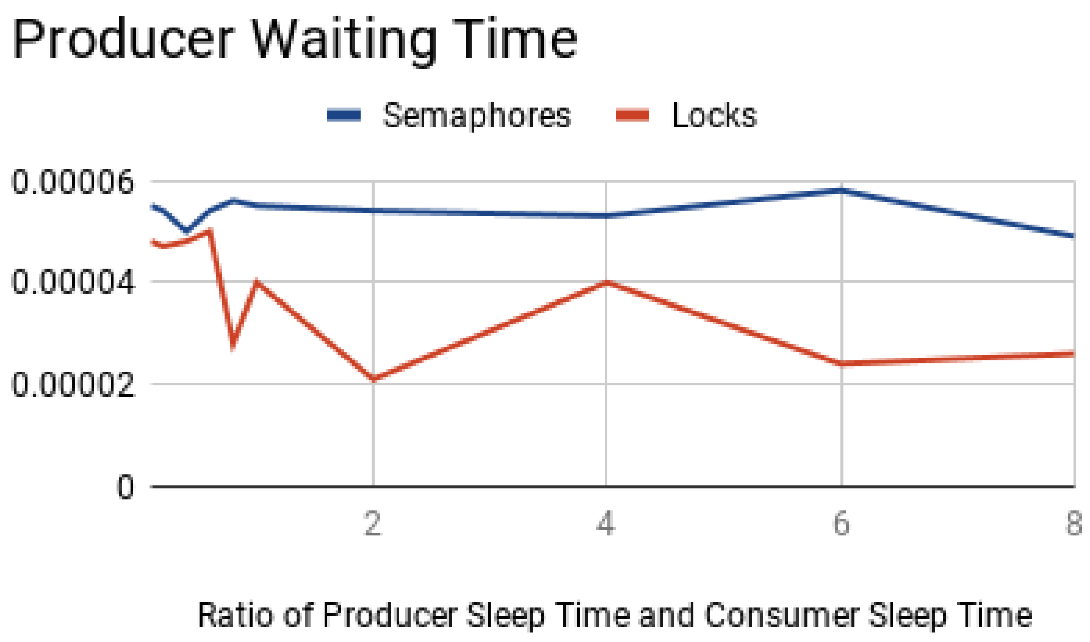

Figure 1: Comparision between Average time taken for accessing the critical section by producers to produce data using locks and semaphores

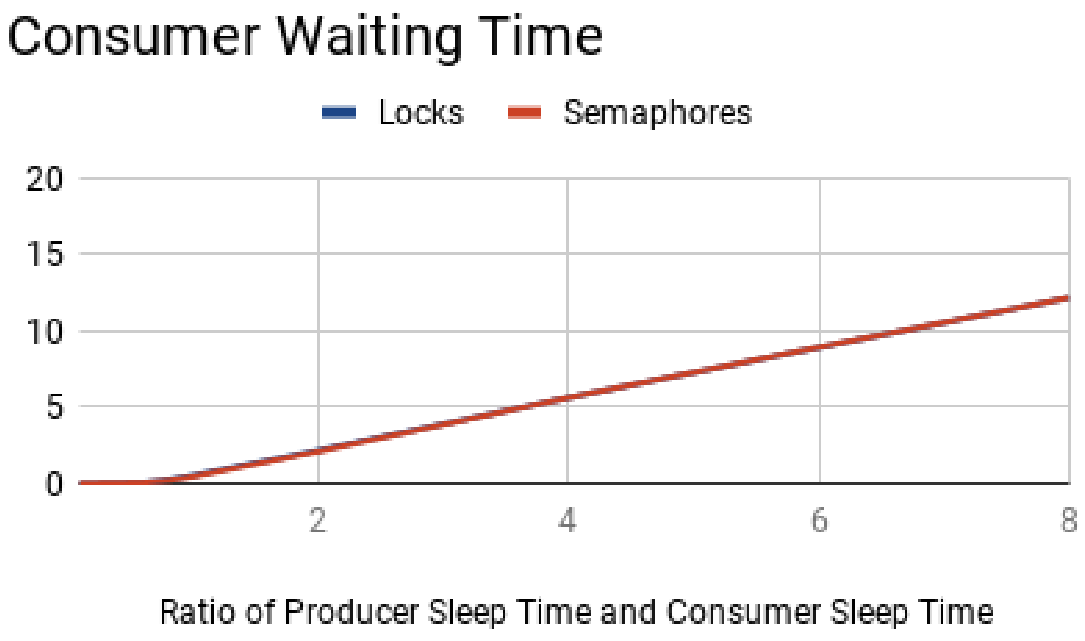

Figure 2: Comparision between Average time taken for accessing the critical section by consumer to consume data using locks and semaphores

6.2 Notes

- Parameters used for test cases:
  capacity = 100
  n_p = 10
  n_c = 15
  rep_p = 15
  rep_c = 10
  Λ_1 = a seconds and Λ_2 = b seconds, such that,
  a + b = 20 and a/b = 10, 8, 6, 4, 2, 1, 0.8, 0.6, 0.4, 0.2, 0.1.
- Each of the plotted points is the average of 5 testcases.
- Locks perform better than semaphores.
- Sleep and wakeup calls are genarally more expensive.

- When buffer is small, usage of locks is suggested because busy waiting seems better than sleep and wakeup.
- CPU usage is very high when locks are used when compared to that of semaphores.
  This is because of busy-waiting in case of locks as opposed to sleep and wakeup in semaphores.
-