Operating Systems–2: CS3523 2019 Programming Assignment 4: Implement solutions to Readers-Writers and Fair Readers-Writers problems using Semaphores Report

Sai Harsha Kottapalli
CS17BTECH11036

March 18, 2019

1 Aim

We have to solve the Readers-Writers problem and Fair Readers-Writers problem using Semaphores.
We then compare the average and worst-case time taken for each thread to access the critical section (shared resources).

2 Design of the Program :

We use Semaphores to implement the RW and Fair RW problems.

2.1 Critical Section :

This part of the code should be accessed by only one writer or multiple readers respectively at a time. We can avoid race conditions by making sure that no two writer/reader enter their Critical Sections at the same time.

2.2 Entry Section :

This part of the code is accessible for all the writers/readers. The job of this code is to make sure only one writer/reader is given access to the critical section at a particular time.

2.3 Exit Section :

This part of the code is where the writer/reader has successfully executed the critical section of the program, and will reset the lock so that other writer/readers can now enter the critical section(if any) while this writer/reader completes 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 Worst Waiting Time :

This is defined as the highest amount of time taken by a writer/reader respectively to critical section takes after requesting for the same.

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.
  usleep - accepts microseconds as a parameter to sleep, used for simulating the critical/remainder section.
- **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.
- **cmath**
  For using **floor()** to compare worst case waiting time.

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

- **Number of writers** - nw
- **Number of readers** - nr

- **Number of times each writer should write** - kw

- **Number of times each reader should read** - kr

- **lam1 and lam2**
  Represents the mean of value of two exponential distribution curves which is to be constructed.
  Given in milli-seconds.

3.3 writer()

The writer() function simulates a writer trying to write data to an object. Only one writer at a time, can be inside the CS to avoid synchronization issues.

3.4 reader()

The reader() function simulates a reader trying to read data to an object. Only readers at a time, can be inside the CS to avoid synchronization issues.

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 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.

- **thread *th for writers and readers**
  th_w and th_r are the pointers for storing the array of nw and nr threads respectively. This will be useful for later calling **join()** to properly exit the threads spawned previously.

- **Average and worst waiting times for writers and readers** Used to calculate the average time as well as worst waiting time, a writer/reader has to wait for getting access to the CS.

4 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.

5 RW

- read_count is a shared variable which stores number of reader thread currently accessing the CS.
- I have used two semaphores, namely - rw_m and mutex.
- **rw_m**
  This semaphore is initialized to 1.
  It is used to ensure that only one writer can access CS at any time.
- **mutex**
  This semaphore is initialized to 1.
  It is used to keep count of number of reader threads currently in CS and to correspondingly allow a writer thread to access CS if there are no reader threads currently accessing the CS.
- **Algorithm :**

Writer

```
sem_wait(&rw_m);
\\ CS
sem_post(&rw_m);
\\ Rem-S
```

Reader

```
sem_wait(&mutex);
read_count++;
if(read_count == 1) {
        sem_wait(&rw_m);
}
sem_post(&mutex);
\\ CS
sem_wait(&mutex);
read_count–;
if(read_count == 0) {
        sem_post(&rw_m);
}
sem_post(&mutex);
\\ Rem-S
```

6 Fair RW

- I have used three semaphores, namely - in, out, wrt and three variables - ctrin, ctrout, wait.
- **in**
  This semaphore is initialized to 1.
- **out**
  This semaphore is initialized to 1.
- **wrt**
  This semaphore is initialized to 0.
- **ctrin**
  It is an integer initialized to 0.
  Signifies the number of reader threads which got access to the CS.

- **ctrout**
  It is an integer initialized to 0.
  Signifies the number of reader threads which completed reading the CS.

- **wait**
  It is a boolean value initialized to false.
  Used to check if any writer thread is waiting for access to CS incase any readers are currently inside CS.

- **Algorithm :**

  Writer

```
sem_wait(&in);
sem_wait(&out);
if(ctrin == ctrout) {
        sem_post(&out);
} else {
        wait = 1;
        sem_post(&out);
        sem_wait(&wrt);
        wait = 0;
}
\\ CS
sem_post(&in);
\\ Rem-S
```

  Reader

```
sem_wait(&in);
ctrin++;
sem_post(&in);
\\ CS
sem_wait(&out);
ctrout++;
if(wait == 1 && ctrin == ctrout) {
        sem_post(&wrt);
}
sem_post(&out);
\\ Rem-S
```

7 Output

- Log files for RW and Fair RW.
  These store the events which occur throughout the simulation.

  Example :
  1st CS Request by Writer Thread 20 at 02:25:57
  1st CS Exit by Writer Thread 1 at 02:25:57
  1st CS Entry by Writer Thread 2 at 02:25:57

- Average_time.txt
  This file stores the average times as well as worst waiting time for writers and readers.
  The values are in milli-seconds.

  Example:
  Average waiting time for writers(ms): 11989.221430
  Average waiting time for readers(ms): 12301.840290
  Worst time for writers to enter CS(ms): 18841.611000
  Worst time for readers to enter CS(ms): 18298.801000

8 Comparision

8.1 Graphs

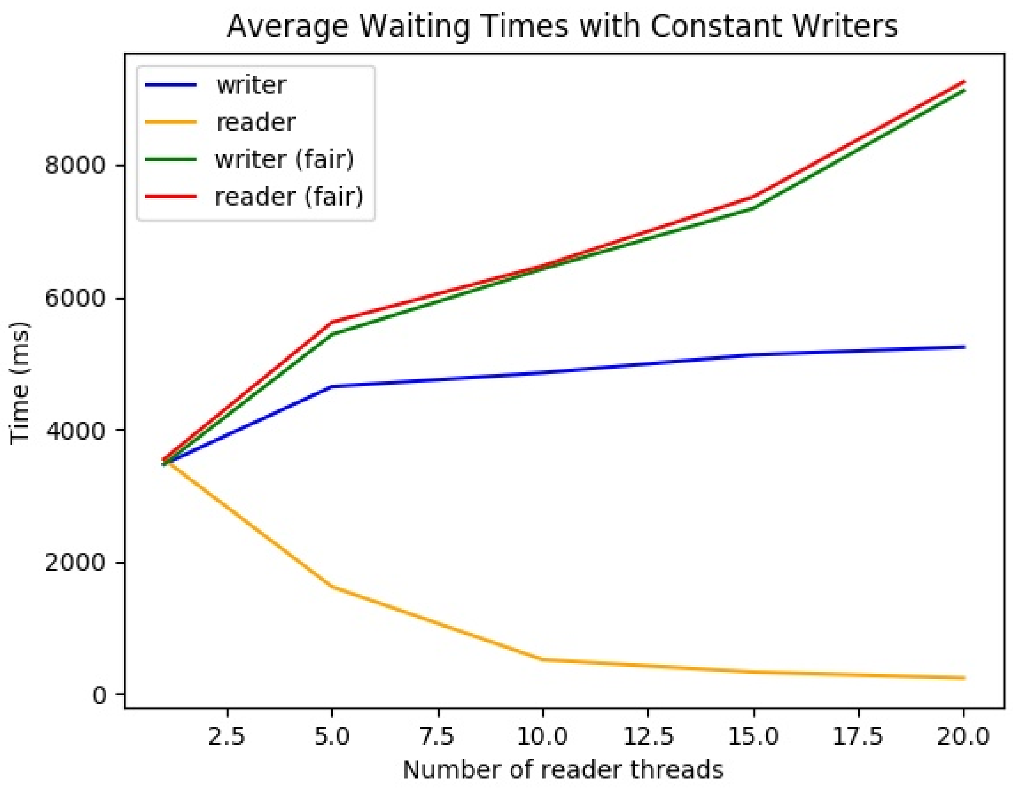

Figure 1: Comparision between Average time taken to enter the CS by reader and writer threads with a constant number of writers for RW and Fair RW

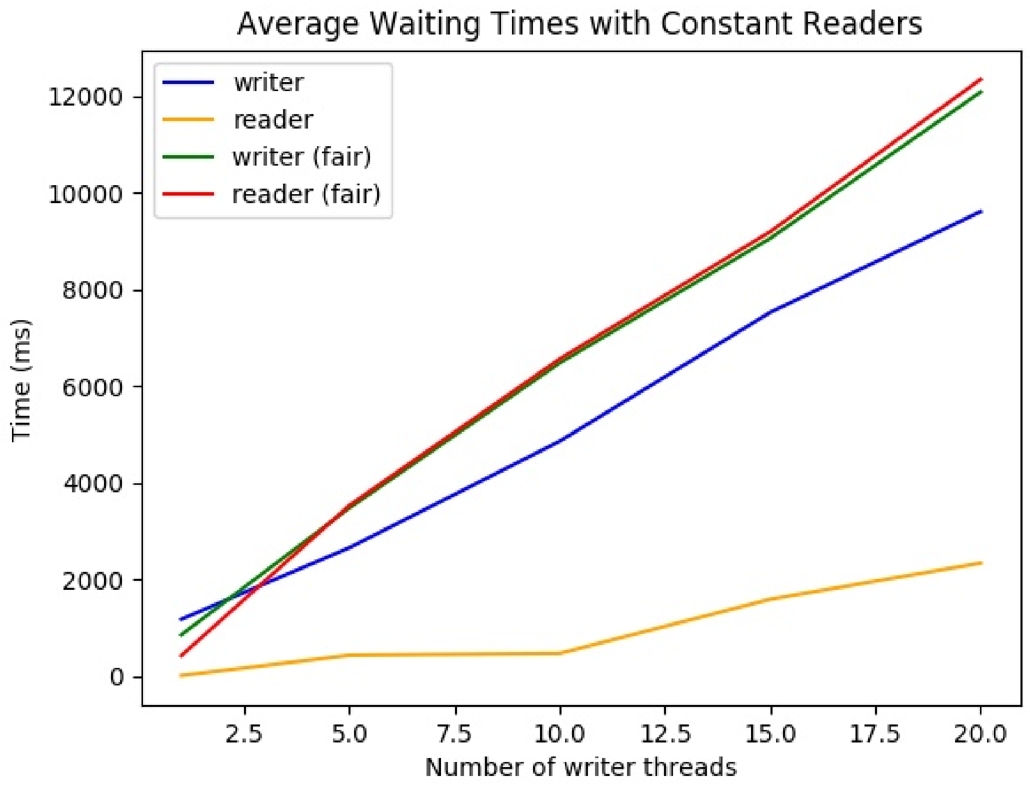

Figure 2: Comparision between Average time taken to enter the CS by reader and writer threads with a constant number of readers for RW and Fair RW

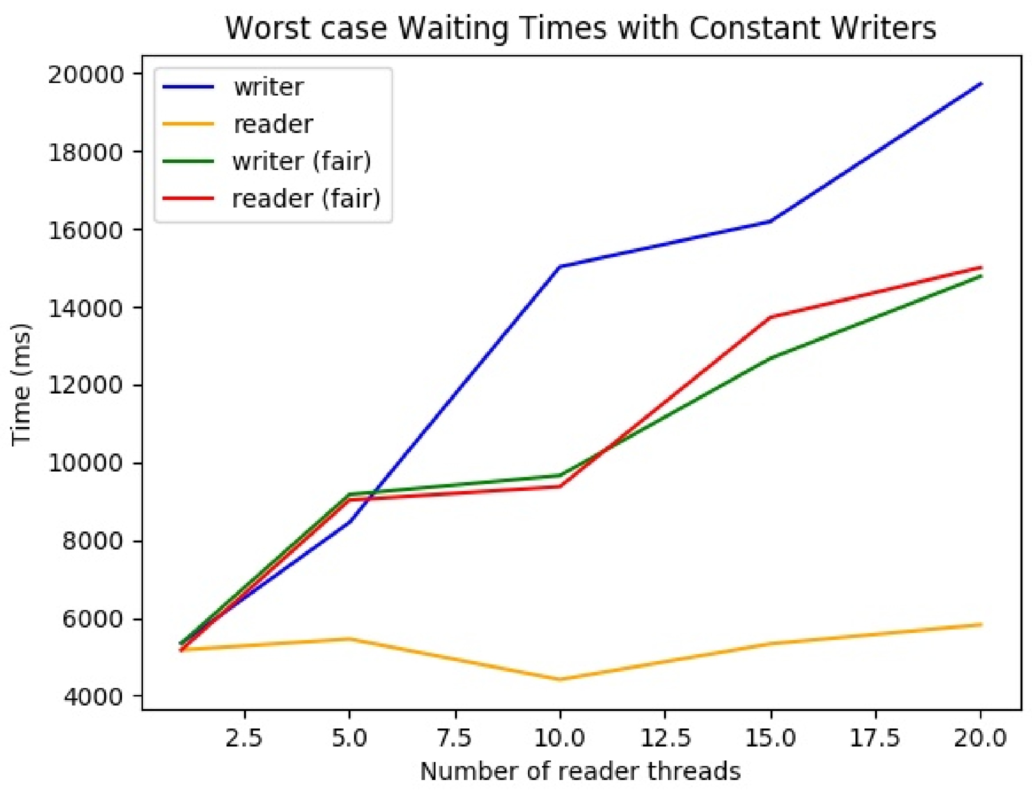

Figure 3: Comparision between worst-case time taken to enter the CS by reader and writer threads with a constant number of writers for RW and Fair RW

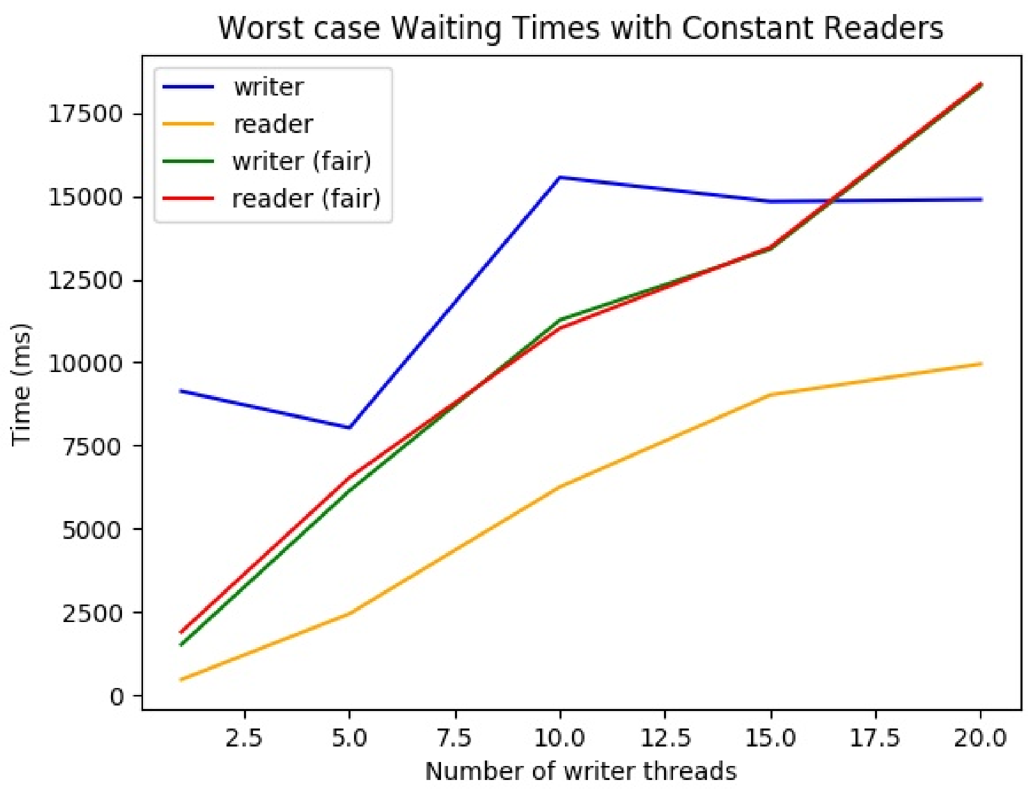

Figure 4: Comparision between worst-case time taken to enter the CS by reader and writer threads with a constant number of readers for RW and Fair RW

8.2 Notes

1. Parameters used for test cases:
   nw = 1, 5, 10, 15, 20
   nr = 1, 5, 10, 15, 20
   kw = 10
   kr = 10
   Λ_1 = 500 milli-seconds and Λ_2 = 800 milli-seconds,

2. Each of the plotted points is the average of 5 testcases.

3. Average waiting times with constant writers

   - As we increase the number of reader threads, we notice that all the threads except reader thread show an increase in waiting times.
   - As for reader, the waiting time decreased with increase of reader threads.
   - There is almost no difference between the waiting times of the reader and writer threads for Fair Readers-Writers when compared to Normal Readers-Writers which shows the effectiveness of the Fair RW algorithm.
   - The starvation of writers in RW algorithm can also be noticed here.
   - Overhead in Fair RW algo, also accounts for increased waiting times, in their case.

4. Average waiting times with constant readers

   - This also follows similar trend as previous graph except even readers waiting time increases with increase in writer threads.
   - The common starting point is lower than previous graph, this is because the number of writer threads are lower than before.
     The writer threads are responsible for other waiting times as well, because this follows a convoy effect.
   - The starvation of writers in RW algorithm can also be noticed here.

5. Worst case waiting times with constant writers/readers.

    - With increase in number of threads, the worst case waiting times also increases.
    - The worst case waiting time for Fair RW is very close for writers and readers, which shows that starvation is taken care of.
      As for normal RW, the writer are starving so they have much higher waiting time when compared to readers.
    - The difference in starting points in the two graphs is due to the same reason as stated earlier - because of writer threads and their convoy effect.