Report On Progamming Assignment 5

Implement solutions to Reader-Writer problem using Semaphores

CS20BTECH11018

Design:

Mutual Exclusion Implementing using semaphores:

For implementing semaphores we used semaphore objects which are introduced in c++20 under semaphore header. Here they defined two kinds of semaphores(counting and binary) .But we only need binary semaphores to solve reader -writer problem whether it is fair or normal

RW:

Here we define two binary semaphores rw_mutex,r_mutex which respectively used to control mutual exclusion for reading or writing and updating readcount.

Reader:

Here we use r_mutex semaphore to update readcount(no of readers entering the CS).We will try to acquire rw_mutex before entering CS if the readcount is 1(when the first reader to enter CS).We will release rw_mutex before leaving the CS if the readcount is 0(when the last reader finished the CS).

Writer:

Here we will try to acquire rw_mutex when enter the CS and release it when leaving the CS.And we are not sure when the writer will acquire rw_mutex,so writer may starve.

FRW:

Here we define three binary semaphores serviceQueue,resource,rmutex semaphore respectively used to control servicing(checking before entering CS),previleges to write or read from a file,updating readcount.

Reader:

Here also we use rmutex semaphore to update readcount(no of readers entering the CS).We will try acquire resource before entering CS if the readcount is 1(when the first reader to enter CS).We will release resource before leaving the CS if the readcount is 0(when the last reader finished the CS).The only difference from rw algorithm is we use serviceQueue semaphore such that only one reader can wait and try to acquire the resource semaphore(if there is no other writer is waiting for resource).

Writer:

Here we will try to acquire rw_mutex when enter the CS and release it when leaving the CS. The only difference from rw algorithm is we use serviceQueue semaphore for acquiring resource(So as to stop other readers to be serviced before the writer which enter before them).

Time calculations:

In order to calculate the present time with precision upto milliseconds for logging we used chrono functions system_clock.now() and calculated duration of time since epoch (casted it to microseconds) then counted it.We then converted the no of micro seconds into time for logging using ctime() and some manipulations.As we only need a part of that string (produced by ctime()) so use substr() function to get it and join micro seconds using concatenations and then use it

Exponential Distribution Sleep times:

In order to simulate random sleep times following exponential distribution we used a pseudo random generator in random library called as exponential_distribution in combination with a default random generator. Here output of default random generator is the input of exponential_distribution. So this function tries to convert the pdf of random variable generated.

Threads Implementation:

We created c++ kind of threads using thread library rather than the posix threads(defined in pthread library) may be because the variety of functions it will support to create threads. We can create c++ threads from functions having arguments of different types. This is not possible using pthreads as they require strict function signature void* func(void*).We used bind() function to bind arguments to the function while defining threads after declaration.

We passed one argument “id” to the thread to indicate the id of the thread created so that it will be helpful when logging.

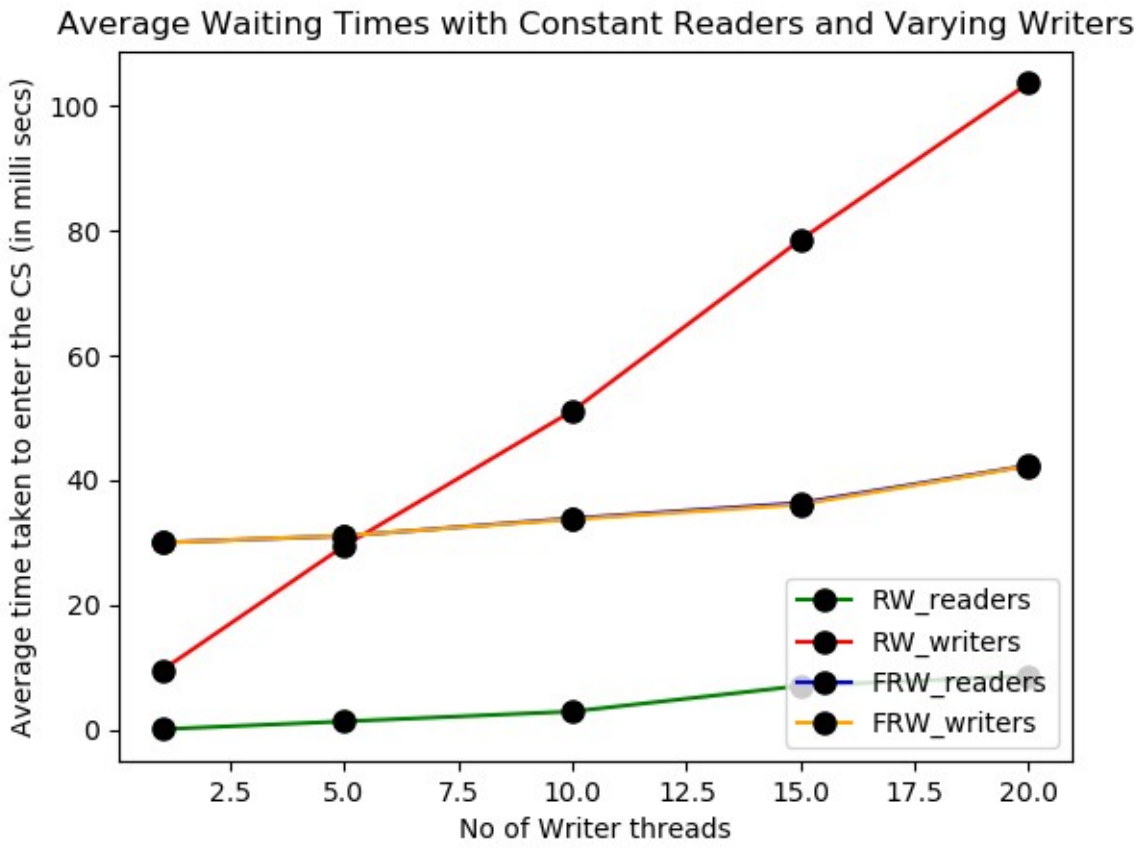
After creating the threads the main thread will wait for different threads to join

by invoking join function specific to each thread.

Graphs:

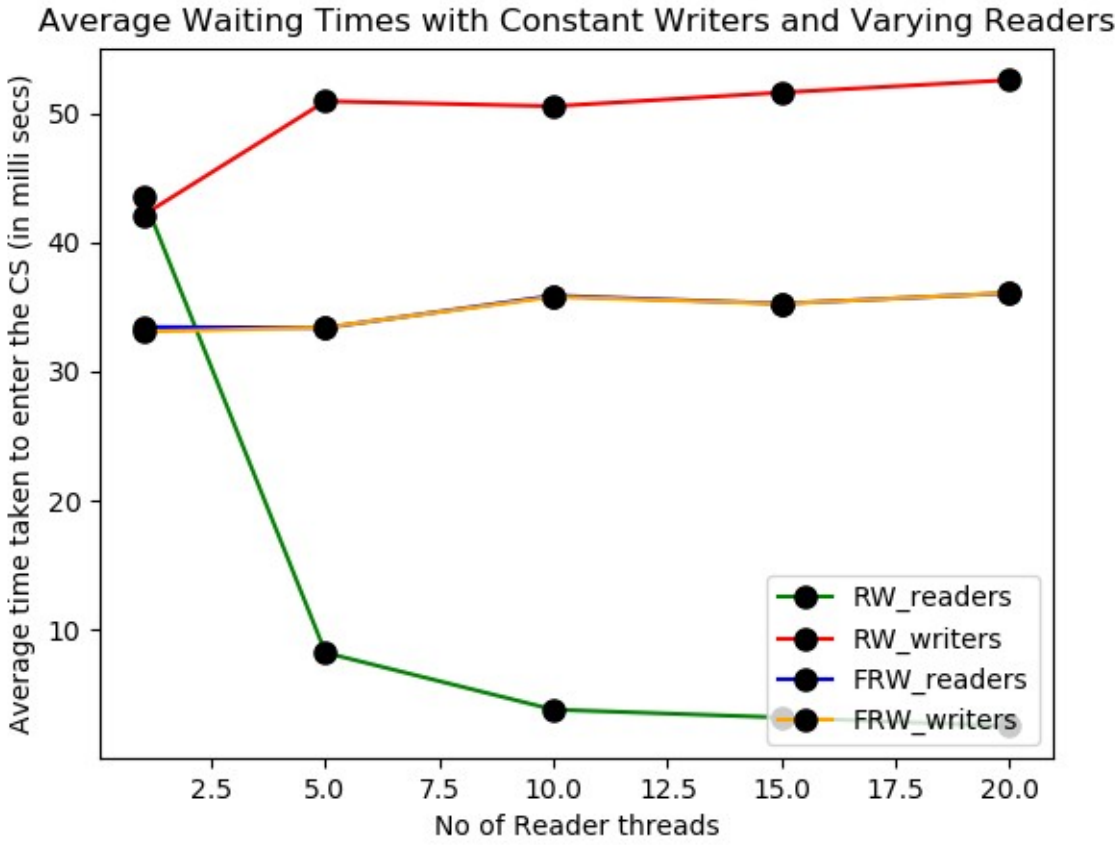
Inorder to construct the graphs below for every situation(const writers/readers & and varying readers/writer threads respectively) we calculated 20 testcases and averaged them and plotted them.Here we took kw=10,kr=10,tCS = 5,tR = 3 for the test cases.

Average Waiting Times with Constant Readers and Varying Writers:



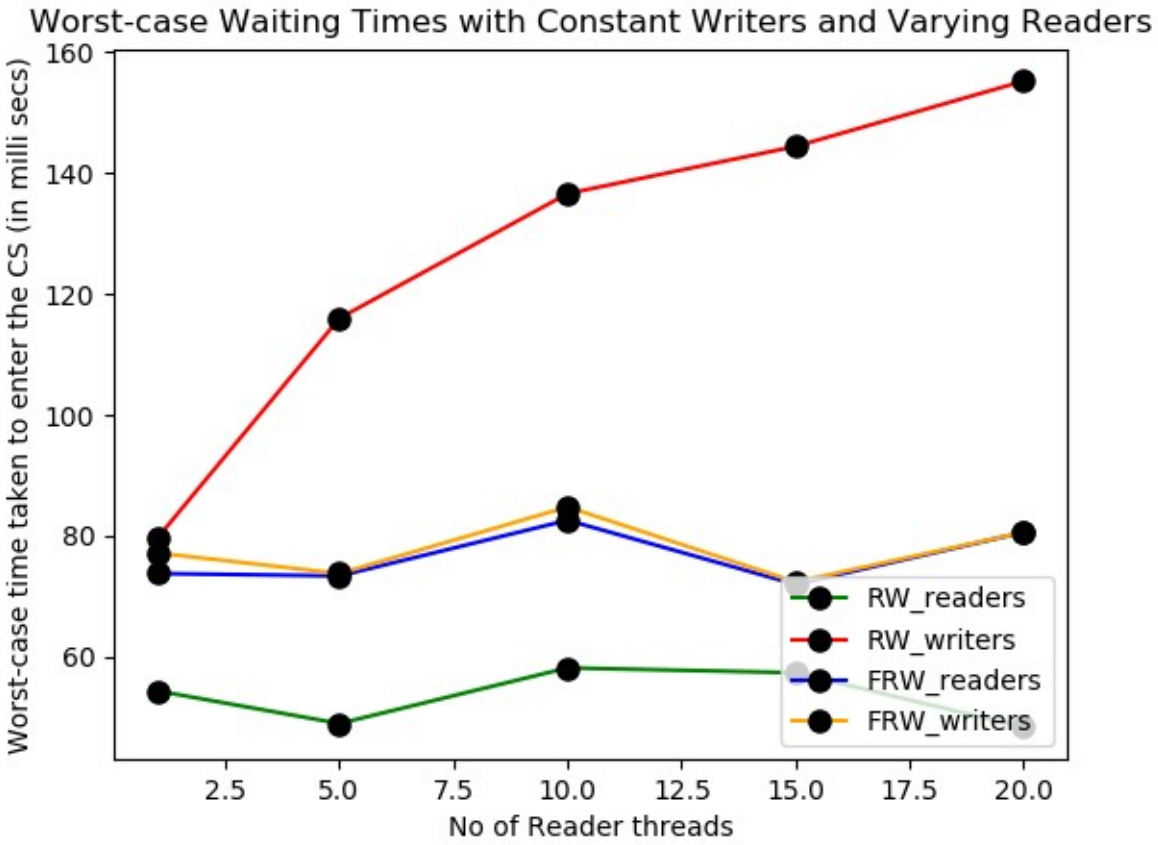
- Average waiting times for RW_writers is significantly higher compared to FRW readers and writers which are also significantly higher than RW_readers.
- Rate at which average waiting times increase with no of writer threads is high for RW_writers. But it seems to be less for FRW_readers,writers and RW_readers.

Average Waiting Times with Constant Writers and Varying Readers:



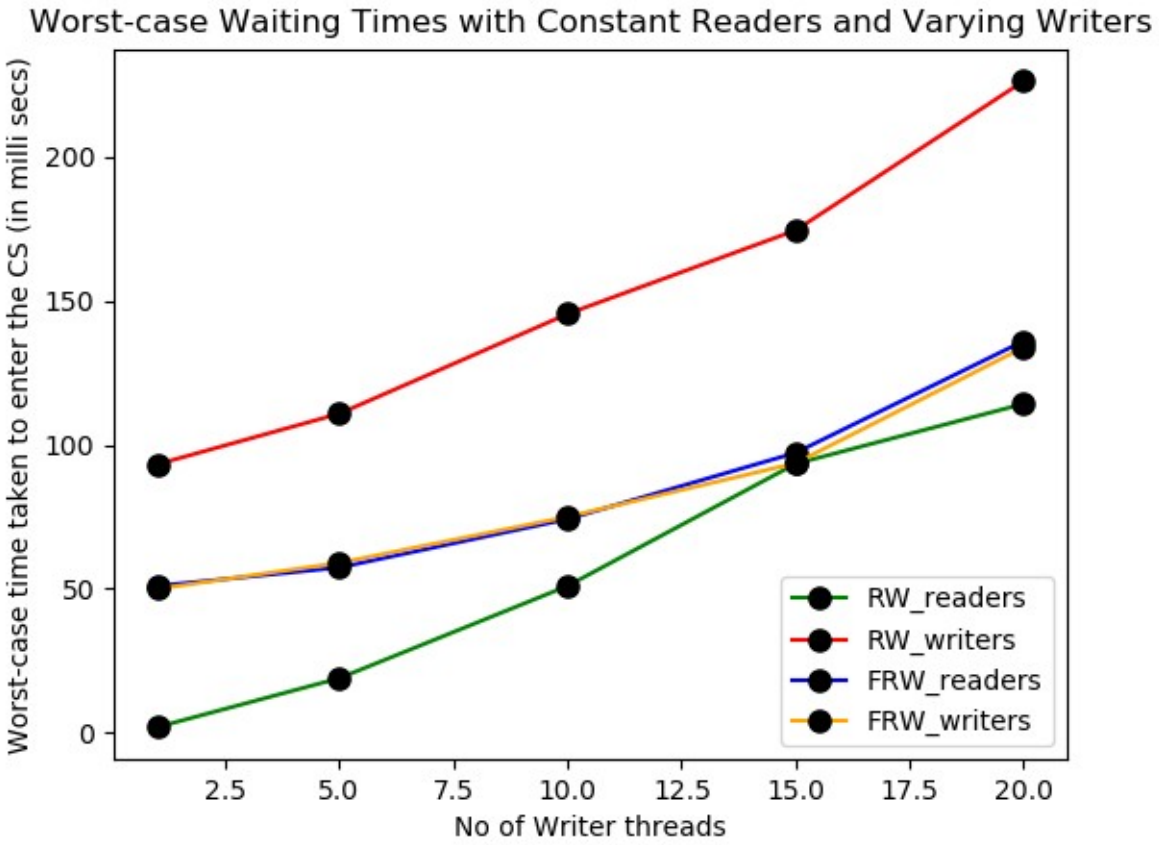
- Here also average waiting times of RW writers are higher than FRW readers,writers and they are higher than RW_readers.
- Here average waiting times of RW_writers,FRW readers,writers seems to be steady with increase in no of readers of course with some local fluctuations whereas average waiting times of RW_readers decreases sharply at start and then decreases steadily with respective to increase in no of reader threads.

Worst-case Waiting Times with Constant Writers and Varying Readers:



- Here,Worst-case waiting times of RW writers is much higher than FRW readers,writers and they are also significantly higher than RW readers.
- Here,worst-case waiting times of RW writers seems to increase sharply when going from (1 to 5 reader threads) but became steady while going from (5 to 10 to 15 to 20 reader threads)
- worst-case waiting times of remaining RW readers,FRW readers and writers seems to be steady with some fluctuations.

Worst-case Waiting Times with Constant Readers and Varying Writers:



- Here,Worst-case waiting times of RW writers is much higher than FRW readers,writers and they are slighly higher than RW readers.
- Worst-case waiting times of RW_writers,FRW readers and writers seems to increase steadily with increase in no of threads whereas RW_readers increased with some local fluctuations.

Analysis:

- FRW writers and readers average waiting times and worst waiting times seem to be very close.It may be because of fairness imposed by the FRW algorithm. (There closeness is hard to believe considering variability of sleep times,it may be because we average over a large no of testcases)
- General trend of worst-case waiting times and average waiting times over FRW,RW readers and writers is

RW_writer > FRW_writer ~ FRW_reader > RW_reader.

It may be because RW is unfair to writer and bias towards readers whereas FRW is considered to be fair.

Conclusion:

FRW writers seems to be fair to writers than compared to RW. They also guarantees that writers will not starve which RW will not guarantee. If we consider consider the good condition waiting times(those near to most favourable condition, when all readers access the CS at the same time and writers cyclically access the CS along with the reader group) RW seems to work better as writers in it will not stop readers to enter the CS when a reader is in CS there by encouraging the grouping of readers. Also, RW algorithms is easier to work with compared to FRW. But this advantages of RW will not be helpful in reality.So FRW solution seems to be more promising than RW solution.