REPORT

DESIGN:

Testing and implementation of every algorithm will be the same. After reading the input file, we create the reader and writer threads separately and give them each the functions ‘*reader ()*’ and ‘*writer ()*’ functions to run in each thread with the implementation of algorithms. Each algorithm’s implementation code differs in these functions ‘*writer ()*’ and ‘*reader ()*’.

FOR RW:

We use the semaphores to implement the first writer-readers solution, where it has the drawback of starvations of writers. We used the following semaphores:

> *Semaphore rw_mutex = 1;*
>
> *Semaphore mutex = 1;*
>
> *int read_count = 0;*

The binary semaphores mutex and *rw_mutex* are initialised to 1; *read_count* is a counting semaphore initialised to 0. The semaphore *rw_mutex* is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable *read_count* is updated. The *read_count* variable keeps track of how many methods are reading the object—the semaphore *rw_mutex* functions as a mutual exclusion semaphore for the writers. The first or last reader also uses it to enter or exit the critical section. It is not operated by readers that enter or exit while other readers are in their critical areas.

If a writer is in the critical section and n readers are waiting, one reader is queued on *rw_mutex*, and *n – 1* readers are queued on the mutex. Also, observe that when a writer executes *signal(rw_mutex),* we may resume the execution of either the waiting readers or a single waiting writer. The scheduler makes the selection.

FOR FAIRRW:

we use the semaphores to implement the third writer-readers solution, known for its fair distribution of writer and reader threads. We used the following semaphores:

int read_count = 0;

Semaphore resource = 1;

Semaphore rmutex = 1;

Semaphore serviceQueue = 1;

The binary semaphores resource, *rmutex* and *serviceQueue* are initialised to 1; *read_count* is a counting semaphore initialised to 0. The semaphore resource is common to both reader and writer processes. The *rmutex* semaphore ensures mutual exclusion when the variable *read_count* is updated. The *read_count* variable keeps track of how many processes are currently reading the object—the semaphore resource functions as a mutual exclusion semaphore for the writers. The first or last reader also uses it to enter or exit the critical section. It is not operated by readers that enter or exit while other readers are in their critical sections. *serviceQueue* is used to queue up the readers after a writer has requested, even though the instant reader thread holds the resource semaphore. This keeps the algorithm fair for both readers and writers.

If a writer is in the critical section and n readers are waiting, one reader is queued on resource, and *n – 1* readers are queued on the *rmutex*. Also, observe that, when a writer executes *signal(resource)*, we may resume the execution of the writer if *seriviceQueue* is held by writer and reader if not. The scheduler makes the selection on *serviceQueue*. Once the writer requests, all the readers are queued on the *serviceQueue*.

Finally, we used the '*exponential_distribution*' feature to calculate the waiting time in the critical section and remainder section to create the illusion of necessary heavy work.

We used a different semaphore, "*print*", to print the logs to the output file to avoid clumsiness and overlapping by keeping mutual exclusion in mind, as several readers run simultaneously.

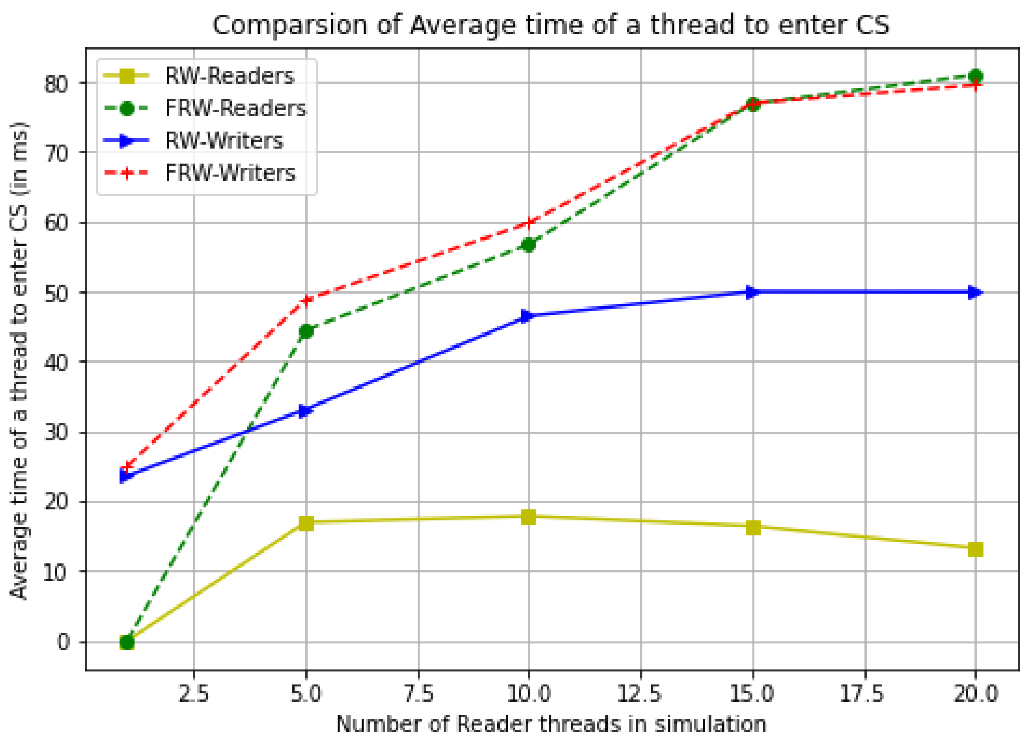
PERFORMANCE COMPARISON:

We used *mucs* value as 5, and *murem* as 20

a. The comparison of average waiting time, where the number of reader threads is varied keeping remaining all parameters same for all iterations which are 10, the graphs contain fours curves for RW-writer, FRW-writer, RW-reader, and FRW-reader threads:

   X-axis: The number of reader threads varying from 1 to 20(1, 5, 10, 15, 20)

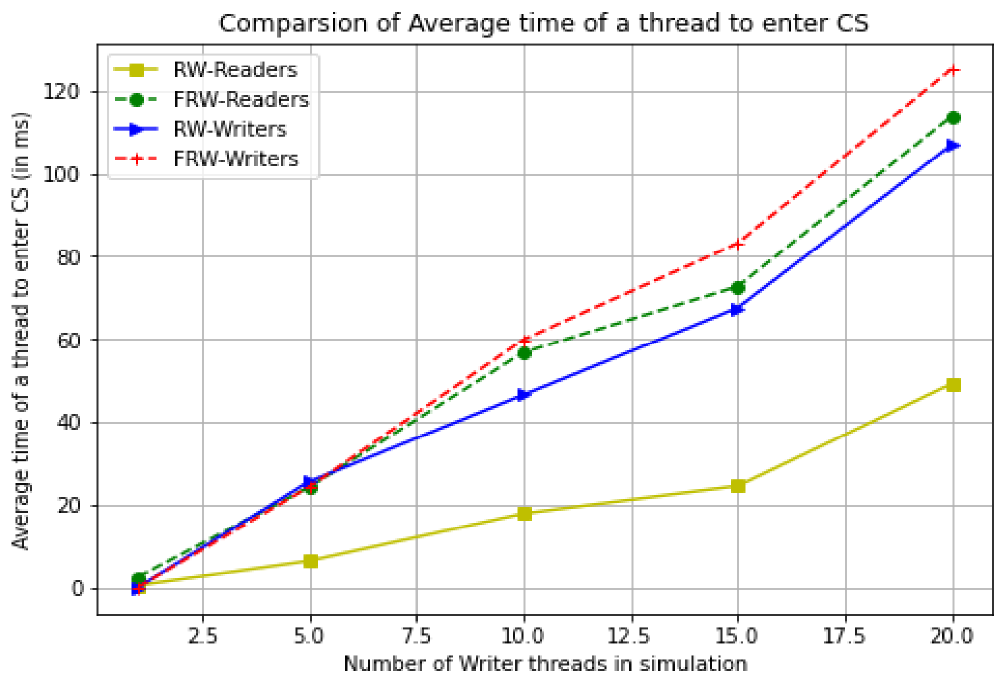
   Y-axis: The average time to reach critical section in milliseconds.



b. The comparison of average waiting time, where the number of writer threads is varied keeping remaining all parameters same for all iterations which are 10, the graphs contain fours curves for RW-writer, FRW-writer, RW-reader, and FRW-reader threads:

X-axis: The number of writer threads varying from 1 to 20(1, 5, 10, 15, 20)

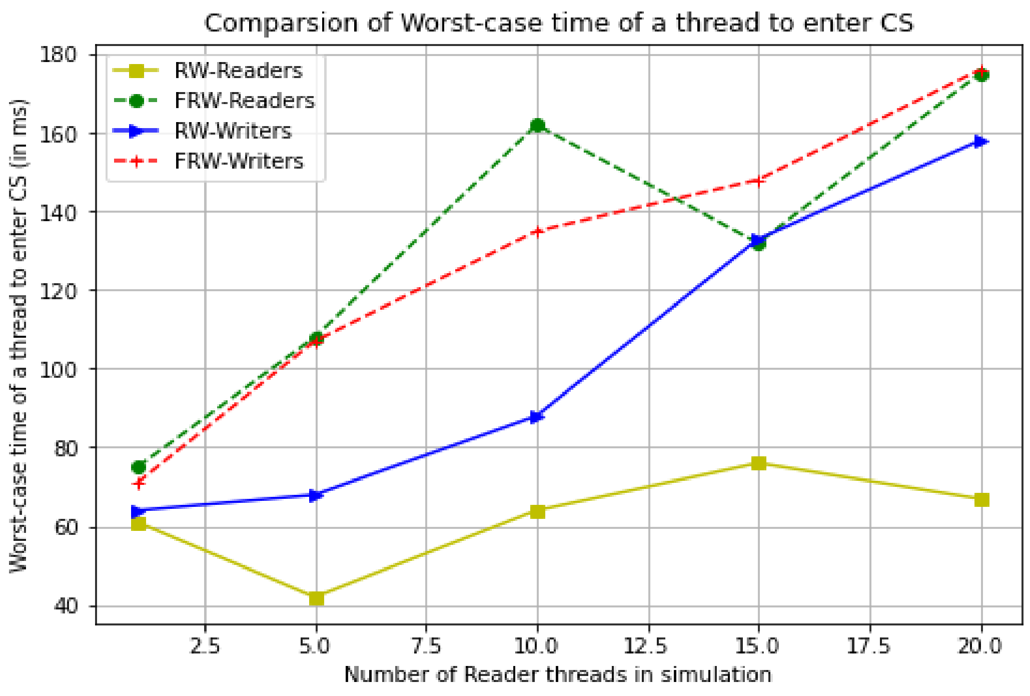
Y-axis: The average time to reach critical section in milliseconds.



c. The comparison of worst-case waiting time, where the number of reader threads is varied keeping remaining all parameters same for all iterations which are 10, the graphs contain fours curves for RW-writer, FRW-writer, RW-reader, and FRW-reader threads:

X-axis: The number of reader threads varying from 1 to 20(1, 5, 10, 15, 20)

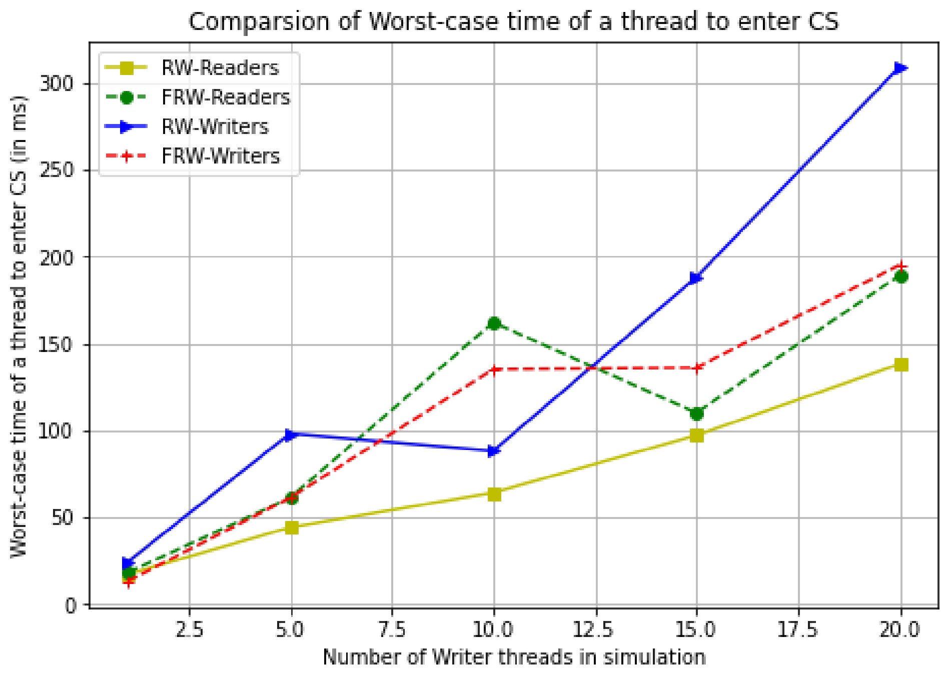
Y-axis: The worst-case time to reach critical section in milliseconds.



d. The comparison of worst-case waiting time, where the number of writer threads is varied keeping remaining all parameters same for all iterations which are 10, the graphs contain fours curves for RW-writer, FRW-writer, RW-reader, and FRW-reader threads:

   X-axis: The number of writer threads varying from 1 to 20(1, 5, 10, 15, 20)

   Y-axis: The worst-case time to reach critical section in milliseconds.



ANALYSIS OF GRAPHS:

1ST GRAPH:

1. Here, we vary the number of reader threads keeping all other parameters the same.
2. We can observe that the writer threads in RW have a higher average waiting time with a large gap from the reader thread of RW.
3. The reader and writer threads in FRW follow the same pattern, and the gap between the FRW reader and writer curves is less.

4. The average waiting time has increased in the case of FRW, but the reader and writer threads have the same share of the waiting time in FRW. This looks fair without any partiality between reader and writer thread. But we can see this in the case of RW.

2ND GRAPH:

1. Here, we vary the number of writer threads keeping all other parameters the same.
2. We can observe that the writer threads in RW have a higher average waiting time with a large gap from the reader thread of RW.
3. The reader and writer threads in FRW follow the same pattern, and the gap between the FRW reader and writer curves is less.
4. The average waiting time has increased in the case of FRW, but the reader and writer threads have the same share of the waiting time in FRW. This looks fair without any partiality between reader and writer thread. But we can see this in the case of RW.
5. This looks like the 1st Graph, But the average waiting time looks more than the 1st Graph. The reader threads can run simultaneously, but the writer threads need to wait until the previous writer thread completes and here, we vary the number of writer threads. This is the reason for the increase in the average waiting time.

3RD GRAPH:

1. Here, we vary the number of reader threads keeping all other parameters the same.
2. We can observe that the writer threads in RW have a higher worst-case waiting time with a large gap from the reader thread of RW.
3. The reader and writer threads in FRW follow the same pattern, and the gap between the FRW reader and writer curves is less.

4. The worst-case waiting time has increased in the case of FRW, but the reader and writer threads have the same share of the total time in FRW. This looks fair without any partiality between reader and writer thread. But we can see this in the case of RW.

4TH GRAPH:

1. Here, we vary the number of writer threads keeping all other parameters the same.

2. We can observe that the writer threads in RW have a higher worst-case waiting time with a large gap from the reader thread of RW.

3. The reader and writer threads in FRW follow the same pattern, and the gap between the FRW reader and writer curves is less.

4. The same pattern followed by the FRW curves show that the time is divided equally. This looks fair without any partiality between reader and writer thread. But we can see this in the case of RW.

5. This looks like the 3rd Graph, But the worst-case waiting time looks more than the 3rd Graph. The reader threads can run simultaneously, but the writer threads need to wait until the previous writer thread completes and here, we are varying number of writer threads. This is the reason for the increase in the worst-case waiting time.