OS Programming Assignment 4

Implementing Solution to Readers-Writers (Writer's preference) and the Fair Readers Writers problem

I have designed a separate program for each of the two problems.

First, in both the programs, I have taken input from the inp.txt file, and initialzed the required variables. I have used the structure

threadInfo, with instances of the structure – readersInfo and writersInfo – which store the threadId, as well as the kw(for writersInfo) and kr(for readersInfo).

Following this, I have created nr readers and nw writers threads, passed on the threadInfo structure, and now they run the writers and readers function.

From now onwards, the two programs start to differ.

Readers-Writers with Writer's Preference

Here we have a total of 4 semaphores, and a file mutex semaphore used. rmutex, wmutex, readTry, and writelock

Acquiring the read lock –

```
// Acquiring the Lock
sem_wait(&rw->readTry);
sem_wait(&rw->rmutex);
rw->readers++;
if (rw->readers == 1)
    sem_wait(&rw->writelock);
sem_post(&rw->rmutex);
sem_post(&rw->readTry);
```

Here we first wait on the readTry semaphore, to request/try to aquire the lock. If in case a writer is already inside, the readTry will be set to 0 by the writer – since this is **writer's preference**, and the reader will have to wait until the writers are done with writing. The rmutex ensures that the readers requests do not cause race condition Once this is done, We check if this is the first reader. If that is indeed the case, we need to wait on writelock – until all the writers have vacated the CS. If it is not the first reader thread – it means some reader thread is already executing in the CS, and this thread can just go in.
After this, we post on the rmutex and readTry, to allow other reader threads who might be trying to get the CS.

Releasing the read lock -

```
// Releasing the Lock, as the CS execution is over
sem_wait(&rw->rmutex);
rw->readers--;
if (rw->readers == 0)
    sem_post(&rw->writelock);
sem_post(&rw->rmutex);
```

To release the readlock, we need to first acquire rmutex. Once we are in, we first decrement the readers by 1, and in case we have exited the last reader thread as well, we post the writelock – to signal that the readers are done, and writers can enter. We release the rmutex at the end

Acquiring the write lock -

```
// Acquiring the lock
sem_wait(&rw->wmutex);
rw->writers++;
if (rw->writers == 1)
{
    sem_wait(&rw->readTry);
}
sem_post(&rw->wmutex);
sem_wait(&rw->writelock);
```

We use the wmutex to prevent multiple writer threads from trying to access the lock at once. If it is the first writer trying to enter, we prevent the readers from even trying to enter the CS – since this is **writers preference.**
At the end, we also acquire the writelock – which blocks the resource just for writers.

Releasing the write lock -

```
// Releasing the Lock, CS execution is over
sem_post(&rw->writelock);
sem_wait(&rw->wmutex);
rw->writers--;
if (rw->writers == 0)
{
    sem_post(&rw->readTry);
}
sem_post(&rw->wmutex);
```

We first release the resource – by doing sem_post on writelock – allowing other writers to enter the CS, and if all the writers are done – readers to enter. Wmutex is just to prevent race conditions while releasing the lock. We check if this was the last writer to exit – and if that is really the case, we release readTry – allowing the readers to now try to enter the CS(This is because of the writer's preference, that the readers can only try after all the writes are done).

Fair Readers-Writers Solution

Here we have a total of 3 semaphores, rmutex, writelock, and serviceQueue

Aquiring the read lock -

```
    //Acquiring the lock
    sem_wait(&rw->serviceQueue);
    sem_wait(&rw->rmutex);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock);
    sem_post(&rw->serviceQueue);
    sem_post(&rw->rmutex);
```

Here, we first wait in the service queue for our request to be serviced. Rmutex is used to prevent potential race condition amongst the threads requesting the readLock. If it is the first reader thread, we aquire the writelock as well,

as we do not want any writer to write while the readers are reading. Now, We release the service queue to allow other threads' request to be serviced.

Releasing the read lock –

```
//CS exection is over, we are exiting it now
sem_wait(&rw->rmutex);
rw->readers--;
if (rw->readers == 0)
    sem_post(&rw->writelock);
sem_post(&rw->rmutex);
```

The function of rmutex is same as before. To release the readers lock, we check if the current reader is the last reader in the critical section. If that is indeed the case, we release the writelock as well – to allow the writers to get into the critical section.

Acquiring the write lock –

```
//Code to acquire the Lock
sem_wait(&rw->serviceQueue);
sem_wait(&rw->writelock);

sem_post(&rw->serviceQueue);


//We are now into the CS
```

Here, we first wait in the service queue for our request to be serviced. Once our turn has come, we wait on the writelock – which we get only when there are no readers, and no other writer in the Critical section. After this – we release the service queue to allow other threads to be serviced.

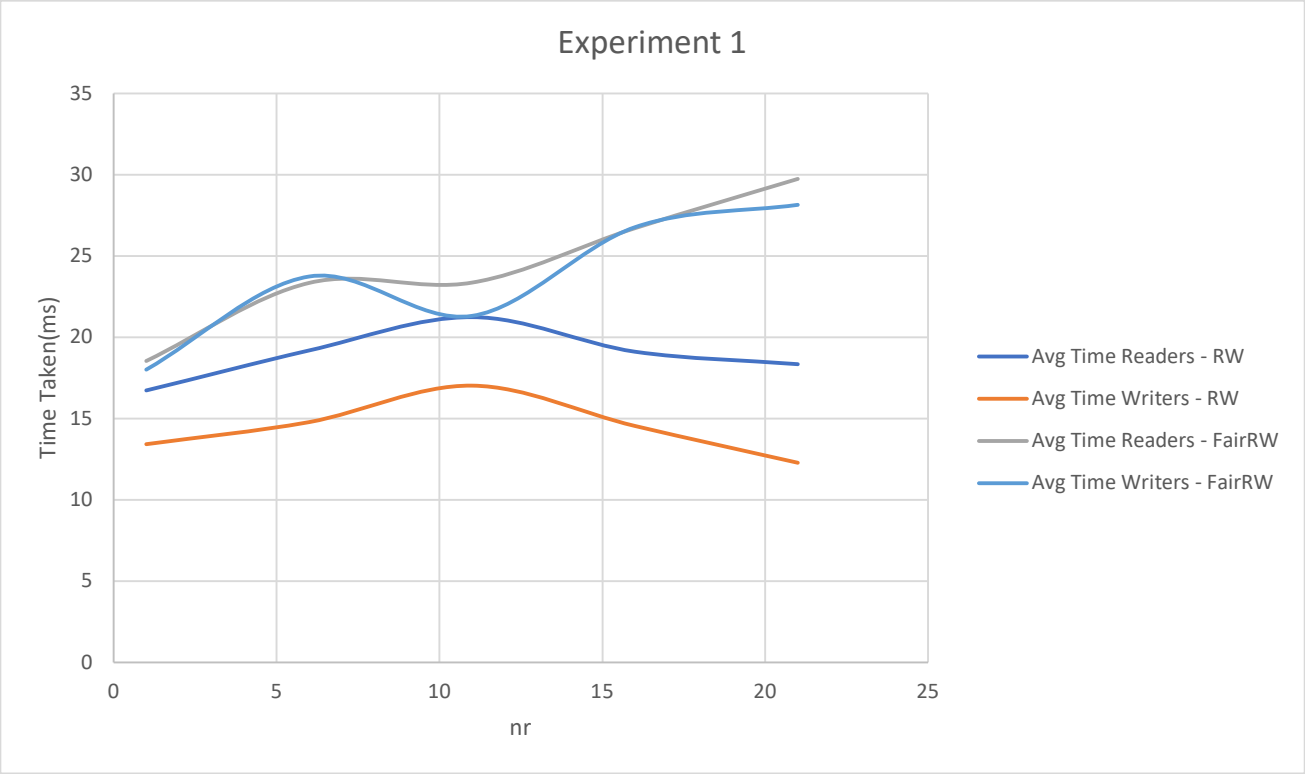
Releasing the write lock – We simply do sem_post on the writelock semaphore to allow the next thread in service queue to be serviced.

```
//Releasing the Lock
sem_post(&rw->writelock);
```

After this, we simply calculate the average and worst case times for both readers and writers, and print them into the outfile. This is same for both the programs.

Graphs for Experiments

Experiment 1 – Average waiting times with constant writers(readers increasing from 1 to 21)



Observations: Here we can observe that for the Writer's preference solution – we don't see much of an increase/decrease on increasing the readers threads, while in the fair solution we see a clear increase in the average times of both the readers and the writers.

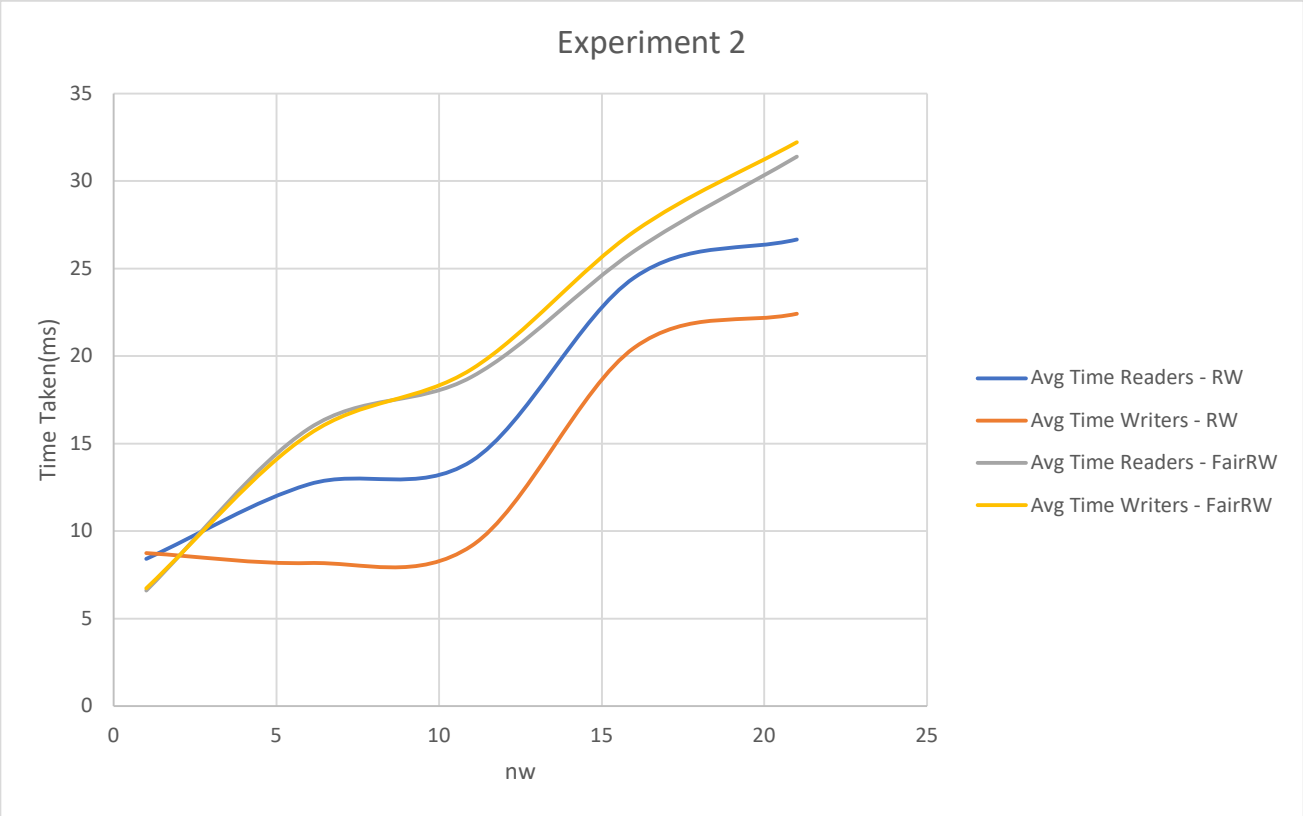
Also, in the writer's preference solution – the average time taken by the writer's threads is lesser than the readers, while in fair solution the average times are more or less the same.

Reason: In the **Writer's preference** solution, readers are only allowed to enter when there are no writers waiting -i.e. the writers are not made to unnecessarily in the queue. As soon as a writer comes into the queue, no more readers are allowed to enter. Thus – the average time taken totally depends on the number of writers – because they are only dictating the time given to the threads. The readers are simply executing in parallel when **there is no writer waiting**. Hence – increasing/decreasing the number of readers will not have any significant impact on the average times. In the fair solution – we see an increase in the

average times with an increase in the reader threads because here both the readers and the writers dictate the average time(both are placed into the service queue – and serviced) – and hence both impact it.

Regarding the second observation, in the writer's preference solution the average time taken by writers is lesser because we are always giving a priority to the writer threads over reader thread, and they are not made to unnecessarily wait in the queue. Whereas, in the fair solution – we are equally prioritizing both of them – hence we do not observe any such priority.

Experiment 2 – Average witing time with constant readers(Writers increasing from 1-21)



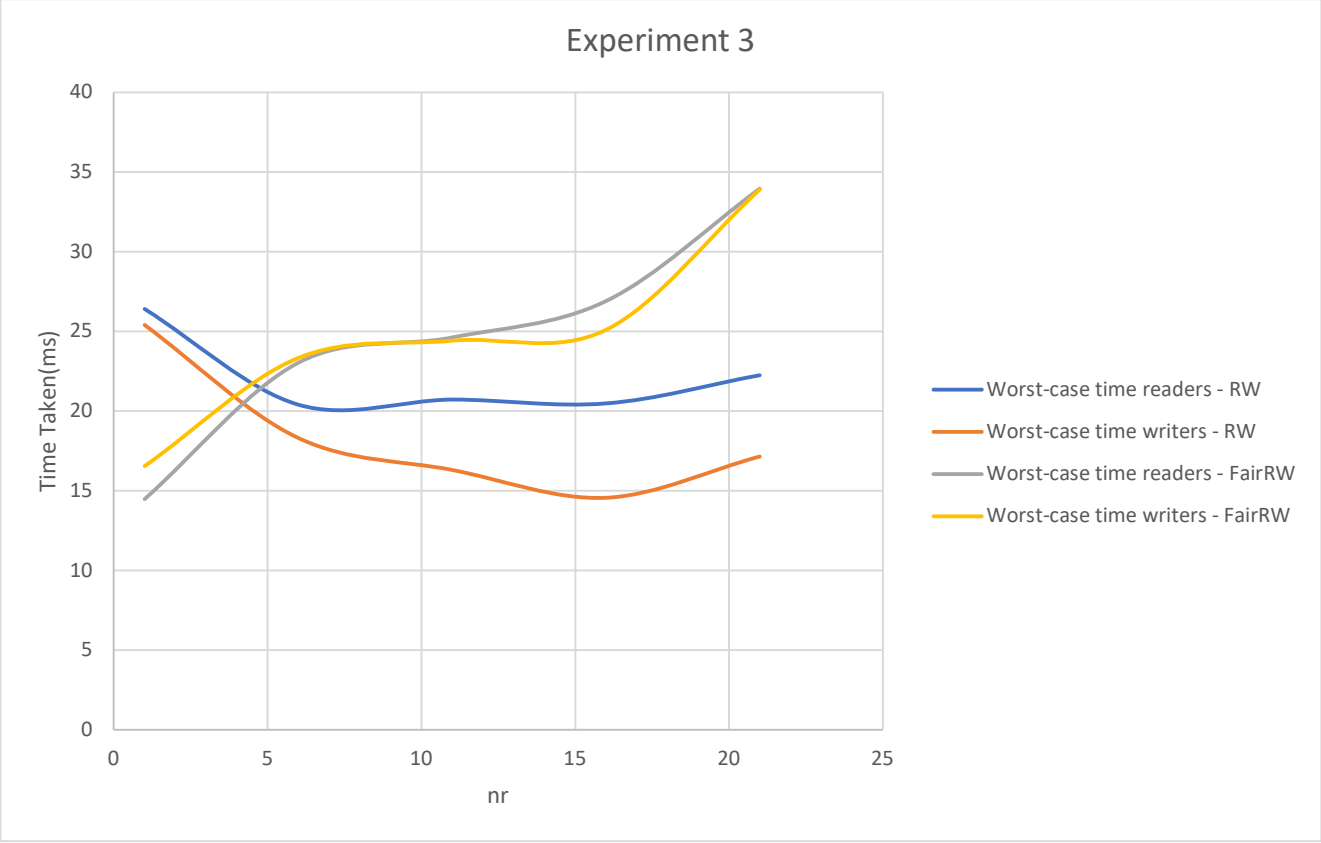
Obeservations: Here, as we can clearly observe – with an increase in the number of writer threads – average time is increasing for all the 4 graphs.

In the writer's preference solution – we observe that the average times taken by the writer threads is lesser than that of reader threads.
In the fair solution – they are almost the same, but the writers worst time is slightly more.

Reason : Here – we are varying the number of writer threads. The number of writer threads impact both the writer's preference solution as well as the fair solution – as explained above. Hence – both of them see an increase in the average time.

In the writer's preference – average time taken by writers is lesser as they are prioritized over readers.
In he fair solution – the average times are more or less the same – but we can sometimes see the readers taking less time – this is because even though the solution is fair – multiple readers can read together, while multiple writers can not. This gives the readers and edge over the writers.

Experiment 3 – Worst case waiting times with constant writers((readers increasing from 1 to 21)



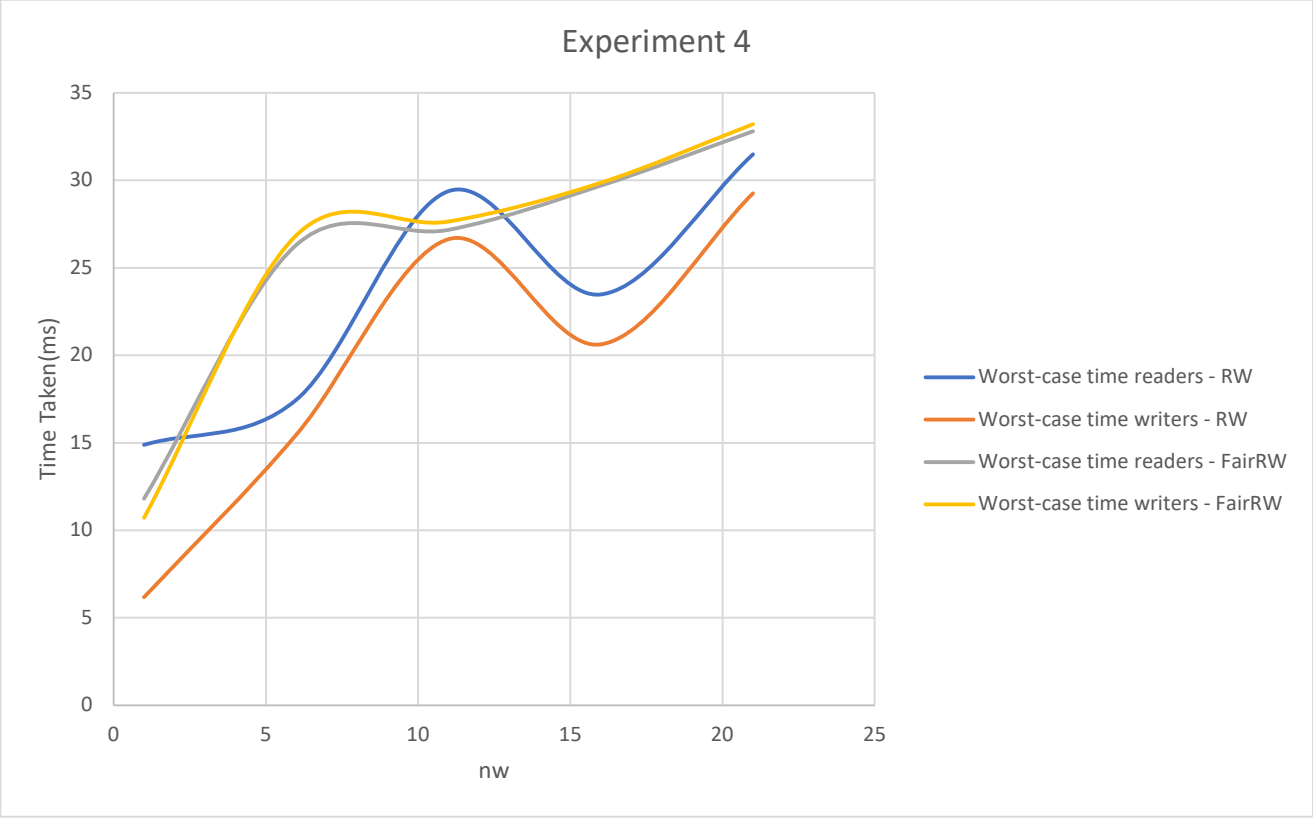
Observations: On increasing the number of reader threads – we can observe that the worst case times for the threads in writer's preference don't see much of an increase/decrease. While in the fair solution – we see an increase in the times.

Also, in the writer's preference, the worst case times of the reader threads is more than the writer threads, whereas in the fair solution – they are almost the same.

Reason: As discussed earlier, the number of readers do not play an important role in deciding the times for the writer's preference solution, and hence the worst case times are not much impacted by it. However – the fair solution gives equal priority to both the readers and writers, and hence on increasing the number of reader threads, we see a clear increase in the worst case times of both the reader and writer threads.

In the writer's preference, as the name suggests – writers are given a priority, and hence the worst case times are larger for the readers. Whereas – in the fair solution, the worst case times of both the readers and writers are almost the same.

Experiment 4: Worst case waiting times with constant readers(Writers increasing from 1-21)



Obsevations: We can observe an increase in the worst case times of all 4 graphs.

In the writer's preference solution – the worst case time for the readers is more than the writers. In the fair solution – they are almost the same, but the writers worst time is slightly more.

Also, I can observe a peak in the writer's preference solution – when nw = 11 and nr = 10. There is no particular reason for observing this sudden increase in the worst case time, it is probably just a special case edge case.

Reason: As discussed above, the number of writer threads play a major role in deciding the times in both the writer's preference as well as the fair solution. Hence – we see an increase in the worst case time of both.

In the writer's preference solution – since the writers are given a priority, we can see that they have less worst case time.
In the fair solution – although both the readers and writers are given equal priority, there can be multiple readers who can read the file, whereas only one writer can write into the file. Hence – the readers are slightly better off here.