CS3523: Programming Assignment-4

Kartikeya Mandapati
CS22BTECH11032

March 15, 2024

1 Introduction

To solve the Readers-Writers problem (writer preference) and Fair Readers-Writers problem using Semaphores in C++.

2 Code Overview

This program is written in C++ and follows modular structure. It is divided into two separate files one for writer preference solution and the other, a fair solution.

2.1 Main Program

The main program will read Nw, Nr, Kr, Kw, uCS and uRem from an input file "input.txt". It then creates Nw writer threads and Nr reader threads. The main program will then wait for all the threads to finish. The threads perform read and write on the imaginary shared resource. The code flow is as follows:

1. Read the input values from `input.txt` file.
2. Initialize the lock, thread attributes, and other variables for the exponential distribution.
3. Creates Nw writer threads and Nr reader threads using a for loop, each starts to run from the `writer()` and `reader()` functions.
4. Wait for all the threads to finish.
5. Print the average and worst case times into output file.

The code utilizes the following functions:

1. `writer()` function: This function is called by each writer thread. It simulates the writer's behavior by writing to the shared resource and then sleeping for a random amount of time.
2. `reader()` function: This function is called by each reader thread. It simulates the reader's behavior by reading from the shared resource and then sleeping for a random amount of time.
3. `printAverageTimes()` This function logs the average times into the output file by computing them using the stored times.
4. Individual custom acquire and release functions for the reader-writer solution using semaphores

Other inbuilt functions used are `pthread_create()`, `pthread_join()`, `sem_wait()`, `sem_post()`, `sem_init()`, `sem_destroy()`,`this_thread::sleep_for()` and inbuilt functions provided by the **chrono** library to measure time and **exponential_distribution** to make the thread sleep for random time.

2.2 Writer function:

The `writer` function is responsible for simulating the behavior of writer threads in the writer preference solution to the reader-writer problem. Below is a step-by-step explanation of its functionality:

1. **Argument Extraction:** The function receives a void pointer `arg` as an argument, which is then casted to an integer pointer to extract the thread ID `id`.
2. **Request Time:** The current system time is obtained using `chrono::system_clock::now()` and converted to a `time_t` object `reqTime` using `chrono::system_clock::to_time_t()`. This time represents the point at which the writer thread requests access to the critical section.
3. **Logging Request:** A string stream `output` is created to format the log message, including the thread ID, request time, and other relevant information. The message is then written to the output file `outfile`.
4. **Lock Acquisition:** The writer thread acquires the write lock using the `rw_lock_writer_acquire` function, ensuring exclusive access to the critical section for writers.
5. **Entry Time:** Similar to the request time, the current system time is obtained and converted to `time_t` object `enterTime`. This time represents the entry of the writer thread into the critical section.
6. **Logging Entry and Wait Time:** Log messages for the writer thread's entry into the critical section and the calculated wait time are formatted and written to the output files `outfile` and `avgfile`, respectively.
7. **Sleep:** The writer thread simulates writing in the critical section by sleeping for a random duration `randCSTime`, generated from an exponential distribution.
8. **Exit Time:** After the sleep duration, the current system time is obtained and converted to `time_t` object `exitTime`, representing the exit of the writer thread from the critical section.
9. **Logging Exit:** A log message for the writer thread's exit from the critical section is formatted and written to the output file `outfile`.
10. **Lock Release:** The writer thread releases the write lock using the `rw_lock_writer_release` function, allowing other threads to access the critical section.
11. **Remainder Section:** Finally, the writer thread simulates executing in the remainder section by sleeping for a random duration `randRemTime`, generated from another exponential distribution.

2.3 Reader function:

1. **Function Signature:** The function `reader(void* arg)` is the function that each reader thread will execute. The `void* arg` parameter is a pointer to the thread's ID.
2. **Thread ID:** The thread ID is retrieved from the argument and stored in the `id` variable.
3. **Request Time:** The current system time is obtained using `chrono::system_clock::now()` and converted to a `time_t` object `reqTime` using `chrono::system_clock::to_time_t()`. This time represents the point at which the writer thread requests access to the critical section.
4. **Logging Request:** A string stream `output` is created to format the log message, including the thread ID, request time, and other relevant information. The message is then written to the output file `outfile`.
5. **Acquire Lock:** The reader thread attempts to acquire the reader-writer lock using the `rw_lock_reader_acquire` function.
6. **Enter Critical Section:** Once the lock is acquired, the thread enters the critical section. The time at which this happens is stored in `enterTime`, and a string containing this information is written to the output file.

7. **Wait Time:** The wait time for the thread to enter the critical section is calculated and written to the average file.

8. **Simulate Work:** The thread then simulates doing some work in the critical section by sleeping for a random amount of time.

9. **Exit Critical Section:** After the work is done, the thread exits the critical section. The time at which this happens is stored in `exitTime`, and a string containing this information is written to the output file.

10. **Release Lock:** The reader thread releases the reader-writer lock using the `rw_lock_reader_release` function.

11. **Simulate Remainder Section:** The thread then simulates executing in the remainder section by sleeping for a random amount of time.

12. **Thread Exit:** Finally, the thread exits by calling `pthread_exit(0)`.

2.4 Writer Preference Solution

In this solution, preference is given to the writers. This is implemented using four semaphores `writers_lock`, `readers_lock`, `resource` and `writers_to_readers`.

- We make sure that readers don't get to access the resource while there are writers, using the semaphore `writers_to_readers`. Every reader must wait for this semaphore to be released by the last writer. Once the reader gets this sempahore, it immediately realeases it, so that the next reader can get it.

- The first writer will lock the `writers_to_readers` semaphore and the remaining writers don't care about it. Only the last writer to finish releases the semaphore, so that the readers can access the resource.

- The number of writers and readers are store in the `writers_count` and `readers_count` variables. The `writers_lock` and `readers_lock` semaphores are used to to avoid race conditions on the readers and writers while they are in their entry or exit sections.

- The writer after attaining the `writers_lock` and `writers_to_readers` semaphores, it then waits for the `resource_lock` semaphore to access the shared resource. This lock ensures that no two writers can access the shared resource at the same time even tough they have attained the other locks.

- Once the writer finishes its execution, it updates the `writers_count` and releases the `writers_lock` and `resource_lock` semaphores allowing other writers to access it.

- The reader after attaining the `readers_lock` semaphore, it then waits for the `writers_to_readers` semaphore to access the shared resource and in case it is the first resource it salso waits for the `resource_lock` semaphore to gain access to the resource. If a writer come in between , it immediately locks the `writers_to_readers` semaphore and ensuring that no other reader thread that comes after it can gain access to the shared resource.

- Once the reader finishes its execution, it updates the `readers_count` and in case it is the last reader current having control of the resource it releases the `resource_lock`, ensuring that no writer gains control while some other reader is running currently.

Listing 1: Writer Preference Solution

```
// Function to acquire write lock
void rw_lock_writer_acquire(rwlock *rw){
    sem_wait(&rw->writers_lock);
    rw->writers_count++;
    if(rw->writers_count==1){
        sem_wait(&rw->writers_to_readers);
    }
```

```
    sem_post(&rw->writers_lock);
    sem_wait(&rw->resource_lock);
}
// Function to release write lock
void rw_lock_writer_release(rwlock *rw){
    sem_post(&rw->resource_lock);
    sem_wait(&rw->writers_lock);
    rw->writers_count--;
    if(rw->writers_count==0){
        sem_post(&rw->writers_to_readers);
    }
    sem_post(&rw->writers_lock);
}

// Function to acquire read lock
void rw_lock_reader_acquire(rwlock *rw){
    sem_wait(&rw->writers_to_readers);
    sem_wait(&rw->readers_lock);
    rw->readers_count++;
    if(rw->readers_count==1){
        sem_wait(&rw->resource_lock);
    }
    sem_post(&rw->readers_lock);
    sem_post(&rw->writers_to_readers);
}

// Function to release read lock
void rw_lock_reader_release(rwlock *rw){
    sem_wait(&rw->readers_lock);
    rw->readers_count--;
    if(rw->readers_count==0){
        sem_post(&rw->resource_lock);
    }
    sem_post(&rw->readers_lock);
}
```

2.5 Fair Solution

This solution implements a fair solution such that no thread starves. This is implemeted using three semaphores `resource`, `readers_lock` and `squeue` which maintains the FIFO policy and along with `read_count` which keeps track of current running readers.

- To acquire the lock, the reader threads waits for the `squeue` semaphore, which maintains the FIFO policy ensuring that no thread starves, no which the thread is if it had requested earlier, it gains the access. It then updates the `read_count` for which it uses `readers_lock` semaphore to avoid race conditions among readers.
- After acquiring the lock on `squeue` it waits for the `resource` semaphore to access the shared resource. Only the first reader will wait for the `resource` semaphore, the remaining readers don't care about it. Only the last reader to finish releases the semaphore, so that the writers can access the resource. After acquiring the `squeue` semaphore, they release it so that other reader can access but do not release `resource` so that writer doesn't get access which a reader is already running.
- `readers_count` is 0 indicates that it is the last raeder currntly running and the resource lock is released.
- The writer waits only for two semaphores `resource` and `squeue`. The `squeue` semaphore is used to maintain the FIFO policy and the `resource` semaphore is used to gain access to the shared resource. The writer after acquiring the `squeue` semaphore, it waits for the `resource` semaphore to access the shared resource. Once it finishes its execution, it releases the `resource` semaphore.

Listing 2: Writer Preference Solution

```
// Function to acquire write lock
void rw_lock_writer_acquire(rwlock *rw){
    sem_wait(&rw->squeue);
    sem_wait(&rw->resource_lock);
    sem_post(&rw->squeue);
}
// Function to release write lock
void rw_lock_writer_release(rwlock *rw){
    sem_post(&rw->resource_lock);
}

// Function to acquire read lock
void rw_lock_reader_acquire(rwlock *rw){
    sem_wait(&rw->squeue);
    sem_wait(&rw->readers_lock);
    rw->readers_count++;
    if(rw->readers_count==1){
        sem_wait(&rw->resource_lock);
    }
    sem_post(&rw->squeue);
    sem_post(&rw->readers_lock);
}

// Function to release read lock
void rw_lock_reader_release(rwlock *rw){
    sem_wait(&rw->readers_lock);
    rw->readers_count--;
    if(rw->readers_count==0){
        sem_post(&rw->resource_lock);
    }
    sem_post(&rw->readers_lock);
}
```

2.6 Output Files

display the log of all the events as shown for each of the algorithms is displayed. Two output files are generated **RW-log.txt** and **FRW-log.txt** for the reader-writer and fair reader-writer solutions, respectively. The log files contain the following information for each thread:

- Request time
- Entry time
- Exit time

Average_time.txt, consisting of the average time a thread takes to gain entry to the Critical Section for each algorithm: RW and Fair-RW.

2.7 Complications

- The main challenge was to implement the reader-writer solution with writer preference and fair reader-writer solution using semaphores. Even small mistake in ordering or releasing the locks could lead to deadlocks and improper solution.
- Another challenge was to simulate the behavior of the threads accurately, including the random sleep times and the logging of events. This required the use of the `chrono` and `random` libraries to generate random sleep times and measure time intervals.
- Logging the output into a single file was also a issue as it was causing synchronization issues and had to use another semaphore for it.

nr	Fair RW (Readers)	Writers Pref (Readers)	Fair RW (Writers)	Writers Pref (Writers)
1	98	96	95	73
5	114	105	107	77
10	144	110	136	82
15	154	111	146	79
20	173	121	161	90

Table 1: Average Waiting Time Values (ms)

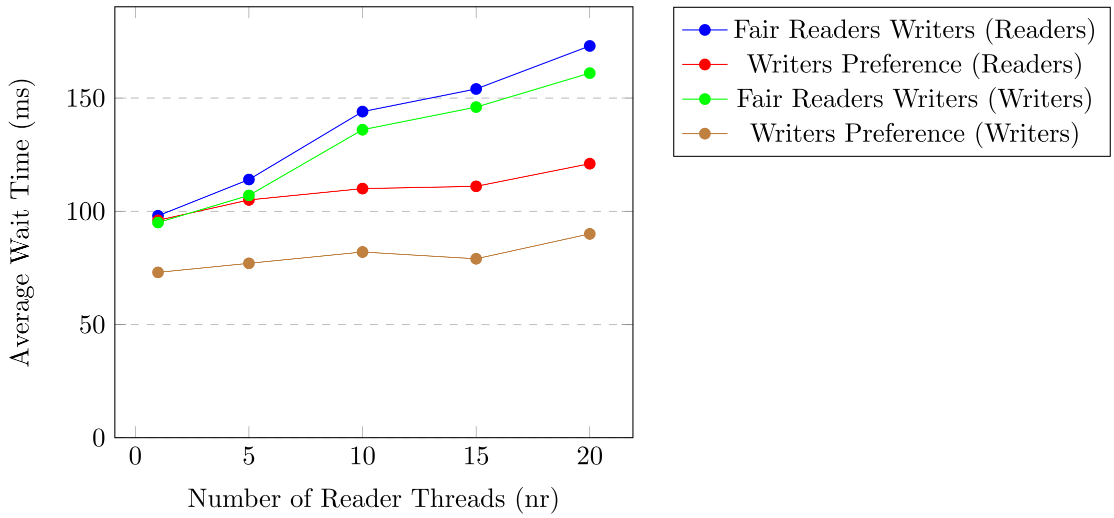

Figure 1: Average Waiting Time for Reader and Writer Threads

3 Experiments:

3.1 Average Waiting Times with Constant Writers:

In this plot, we measure the average time to enter the CS by reader and writer threads with a constant number of writers. Here, we vary the number of reader threads nr from 1 to 20 in increments of 5 on the X-axis. All the other parameters are fixed: Number of writer threads, nw = 10, kr = kw = 10. The Y-axis will have time in milliseconds and will measure the average time taken to enter CS for each reader and writer thread. Specifically, the graph will have four curves:

1. Average time the reader threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()
2. Average time the writer threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()

3.2 Analysis:

- We observe that in the case of Fair reader writer solution the average waiting time is nearly same for both readers and writers with slighly higher value for readers.
- For both solutions the average waiting time **increases** for both writer and readers as the number of increases, the writers have to wait for more readers and same for readers when some writer has already requested before them.
- In the case of writer preference solution the average waiting time is higher for readers as compared writers as expected as more preference is given to writers.

nw	Fair RW (Readers)	Writers Pref (Readers)	Fair RW (Writers)	Writers Pref (Writers)
1	19	23	18	22
5	71	43	73	25
10	119	108	120	72
15	197	153	182	120
20	220	196	213	162

Table 2: Average Waiting Time Values (ms)

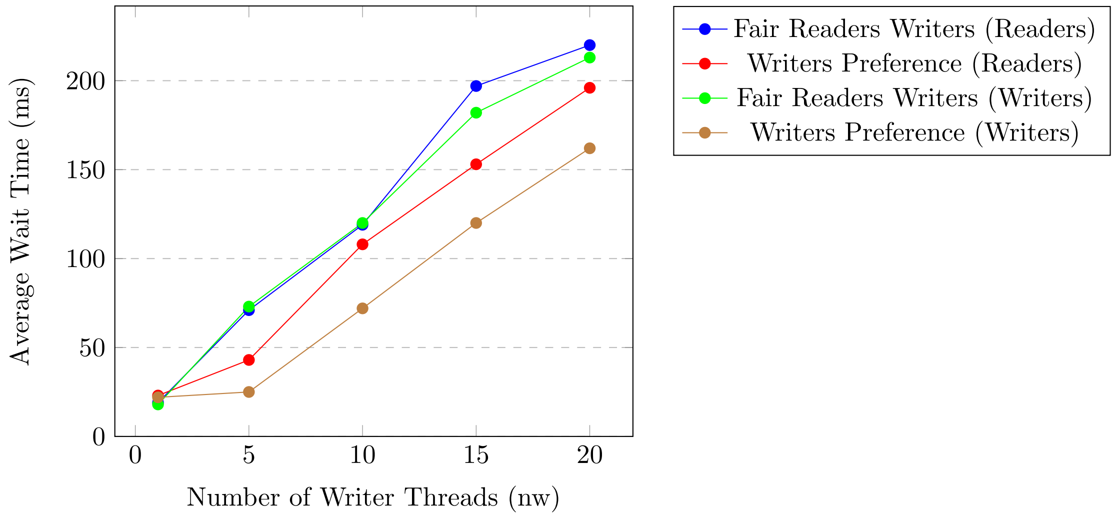

Figure 2: Average Waiting Time for Reader and Writer Threads

- Comparing between fair solution and writer preference one, the increment is higher in fair solution as in the process of increasing fairness, we are making threads wait for longer time.

3.3 Average Waiting Times with Constant Readers:

In this plot, we measure the average time taken to enter the CS by reader and writer threads with a constant number of readers. Here, we vary the number of writer threads nw from 1 to 20 in increments of 5 on the X-axis. All the other parameters are fixed: Number of reader threads, nr = 10, kr = kw = 10. The Y-axis will have time in milliseconds, and the average taken to enter CS will be measured for each reader and writer thread. The plot will have four curves:

1. Average time the reader threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()
2. Average time the writer threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()

3.4 Analysis:

- In this case the result is as expected. The average waoting time for both readers and writers increases for both the solutions as the number of writers increase, as the readers have to wait for more writers and the writers have to wait for more writers too.
- Average waiting time if higher for readers and is close for both solutions as they need to wait for writer anyways.
- For writers, the average waiting time is higher in case of fair solution as they need to wait for readers too.

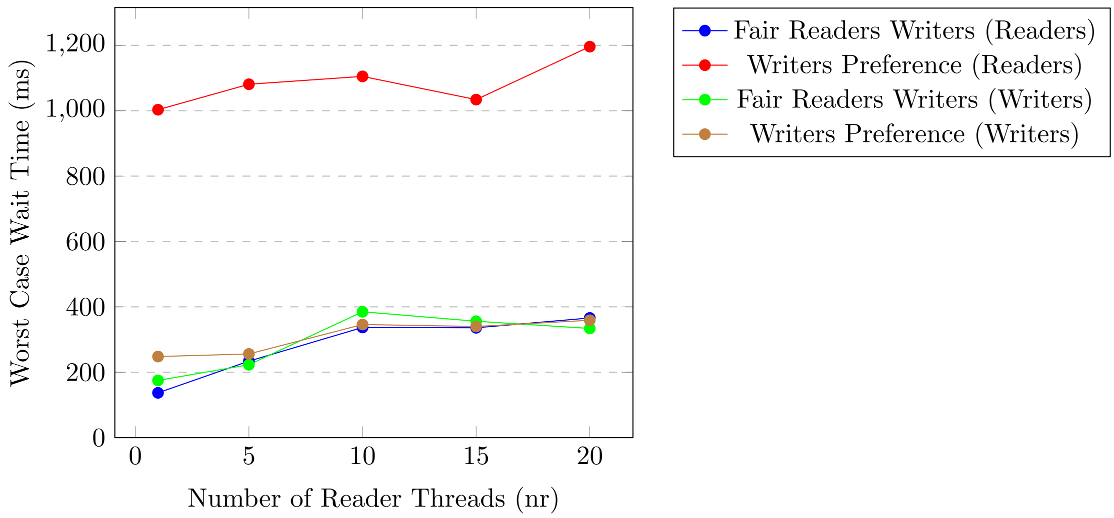

Figure 3: Worst Case Waiting Time for Reader and Writer Threads

3.5 Worst-case Waiting Times with Constant Writers:

This plot will be similar to the graph in Step 1. In this graph, we measure the worst-case (instead of average) time taken to enter the CS by reader and writer threads with a constant number of writers. Here we vary the number of reader threads nr from 1 to 20 in the increments 5 on the X-axis. All the other parameters are fixed: Number of writer threads, nw = 10, kr = kw = 10. The Y-axis will have time in milli-seconds and measure the worst-case time taken to enter CS by the reader and writer threads. The graph will have four curves:

1. Worst case time taken by the reader threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()
2. Worst case time taken by the writer threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()

nr	Fair RW (Readers)	Fair RW (Writers)	Writers Pref (Readers)	Writers Pref (Writers)
1	137	175	1003	248
5	234	223	1081	256
10	337	385	1105	346
15	336	356	1034	340
20	366	334	1196	359

Table 3: Worst case Waiting Time Values (ms)

3.6 Analysis:

- In the case of readers for writers preference solution the average waiting time does not increase significantly as the number of reader threads increases. This is because the writer preference solution allows the writer to access the resource as soon as it is free, and the reader threads have to wait for all the writer threads to finish before they can access the resource. So eventhough the number of reader threads increases, they still have to wait only for writer threads which would be same and they don't have to wait for other readers. So the average waiting time lies in the same range with slight increment.

- In the case of readers with fair solution there is increment in average waiting time as number of readers increases they are also allowed to run in between and the thread requesting later has to wait for both writers and writer that have come before. So the average waiting time increases as the number of reader threads increases.
- For writers in case of writer preference solution increases just slightly as the number of reader threads increases. This is because majority of waiting time for writers still depends mostly on the number of writers.
- In case of fair solution also there is increment as the number of reader threads increases the writer threads need to wait for the reader threads that have requested before it. The higher the number of reader therads, the more would've requested befoer every writer thread.

3.7 Worst-case Waiting Times with Constant Readers:

This plot will be similar to the graph in Step 2. In this graph, we measure the worst-case time taken to enter the CS by reader and writer threads with a constant number of readers. Here, we vary the number of writer threads nw from 1 to 20 in increments of 5 on the X-axis. All the other parameters are fixed: Number of reader threads, nr = 10, kr = kw = 10. The Y-axis will have time in milliseconds and will measure the worst-case time taken to enter CS for each reader and writer thread. The plot will have four curves:

1. Worst case time taken by the reader threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()
2. Worst case time taken by the writer threads.
   (a) Readers Writers()
   (b) Fair Readers Writers()

nw	Fair RW (Readers)	Fair RW (Writers)	Writers Pref (Readers)	Writers Pref (Writers)
1	64	43	74	42
5	147	135	437	121
10	250	298	1076	290
15	609	613	1530	458
20	754	640	1962	629

Table 4: Worst Case Waiting Time Values (ms)

3.8 Analysis:

- In the plot we can see than the worst case wait time increments drastically for reader threads in the writer preference solution as the number of writer threads increases. This is because the writer preference solution allows the writer to access the resource as soon as it is free, and the reader threads have to wait for all the writer threads to finish before they can access the resource.The higher the number of writer threads, for more time the resource is taken over by writers and readers are kept waiting for longer time This leads to a significant increase in the worst-case wait time for reader threads as the number of writer threads increases.
- In case of fair writer reader solution the increment is not as drastic as the writer preference solution. This is because the fair solution allows the reader threads to access the resource as soon as it is free, and the writer threads have to wait for all the reader threads to finish before they can access the resource.
- In the case of writers the difference not as significant as readers as in both writers preference and fair solution the writers have to wait for other writer threads , which is same in case of both solutions and the time spent waiting for reader threads is not very significant as they execute concurrently. So the worst case time increases but not drastically.

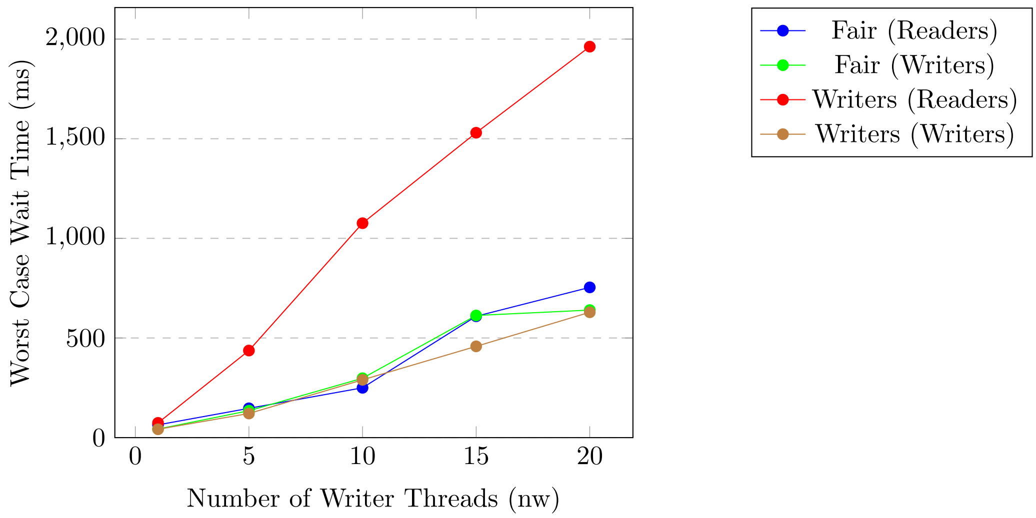

Figure 4: Worst Case Waiting Time for Writer and Reader Threads

3.9 Conclusion

The solutions to reader writer problem were implemented successfully and we were able to implement mutual exclusion along with resource utilization. The trends obtained by varying the number of threads were also nearly as expected.
The instructions to compile and run are mentioned in the readme file.
References used: https://en.wikipedia.org/wiki/Readers