Operating Systems-2: Spring 2024
Programming Assignment 4:
Implement solutions to Readers-Writers (writer preference) and Fair Readers-Writers problems using Semaphores

Waghmare Aditya Abhaykumar
CS22BTECH11061

March 16, 2024

I) Coding Approach

This Programs are Implementations of Readers-Writers (writer preference) and Fair Readers-Writers problems using Semaphores in C++. Readers and Writers are threads accessing a shared resource where multiple readers can access the resource at the same time, but only one writer can access the resource at a time. The low level design of program is explained below-

Common Functions/Code in Both Programs -

Main Function

For Both Programs, `Main function` reads the input from the `inp-params.txt` file and stores the value in global variables. It also initializes the global variables `ofstream out_file` (To Access Output file), `chrono::system_clock::time_point startTime` (to store the program start time i.e. time just before threads started executing), `int64_t **reader_time` (to a 2d array to store time taken by each reader) and `int64_t **writer_time` (to a 2d array to store time taken by each writer) with `void prepare_2d_arrays()` function. It also initializes the binary semaphores.

Once done with Initializing, it creates `nw` writer threads with `void writer(int id);` thread routine and `nr` reader threads with `void reader(int id);` thread routine. After Joining the threads, the Average and Worst time taken by Readers and Writers is Calculated and Written to the Output file.

generateDelay function

Here Output is a delay value exponentially distributed with an Average value averageTime milli-seconds. These time delays aim to simulate these threads performing some complicated, time-consuming tasks. It is Used to Generate randCSTime(sleep time in CS in miliseconds) and randRemTime(sleep time in remainder section in miliseconds) which are exponentially distributed with Average value of μcs and μrem milli-seconds.

Listing 1: generateDelay

```
int generateDelay(double averageTime) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::exponential_distribution<> dist(1.0 / averageTime);
    return static_cast<int>(dist(gen));
}
```

write_output function

This function is used to write the output to the output file. We have to use mutual exclusion techniques to write the output as multiple threads may access the output file at once in the given implementation. This is achieved through the use of a binary semaphore `sem_t out_sem`.

Listing 2: write_output

```
//binary semaphore for output file access
//only one thread can write to the file at a time
sem_t out_sem;
void write_output(string output){
    //aquiring the semaphore
    sem_wait(&out_sem);

    out_file << output;

    //releasing the semaphore
    sem_post(&out_sem);
}
```

Common Reader Code

This Code is Common to Both Programs. The Only Difference is the Entry And Exit Section of the Critical Section. The Reader Thread id is passed by the main function, it ranges from 0 to nr-1. Each reader thread enters the CS kr times. The reqTime, enterTime and exitTime are in milliseconds relative to the program start time. The output is written with the `write_output()` function for mutual exclusion. `generateDelay()` function is used to generate randCSTime and randRemTime. The time taken by the readers to access the CS each time is stored in the `int64_t **reader_time` 2D array. The Common Code is as Follows -

Listing 3: Reader common code

```
    void reader(int id) {
        //each reader thread will enter the CS kr times
        for (int i = 0; i < kr; i++) {
            //get current system time
            auto rt = std::chrono::system_clock::now();
            auto reqTime = chrono::duration_cast<std::chrono::microseconds>(rt - startTime).count();
            //Convert to miliseconds Relative to Program Start Time

            string output = to_string(i) + "th CS request by Reader Thread " + to_string(id) + " at " + to_string(reqTime) + "\n";
            write_output(output);

            // ------------------------------------------------------------------------------------------

            /*
             * Write your code for Readers Writers() and Fair Readers Writers() Using Semaphores here.
             */

            //<ENTRY SECTION>

            //Different for each algorithm

```

```
            // ------------------------------------------------------------------------

            //<CRITICAL SECTION>

            //get entry time relative to start time in miliseconds
            auto ent = std::chrono::system_clock::now();
            auto enterTime = chrono::duration_cast<std::chrono::microseconds>(ent-startTime).count();
            output = to_string(i) + "th CS Entry by Reader Thread " + to_string(id) + " at " + to_string(enterTime) + "\n";
            write_output(output);

            // Simulate a thread reading from CS
            int randCSTime = generateDelay(ucs);
            this_thread::sleep_for(chrono::milliseconds(randCSTime));

            // ------------------------------------------------------------------------
            /*
             * Your code for the thread to exit the CS.
             */

            //<EXIT SECTION>

            //Different for each algorithm

            // ------------------------------------------------------------------------

            //get exit time relative to start time in miliseconds
            auto et = std::chrono::system_clock::now();
            auto exitTime = chrono::duration_cast<std::chrono::microseconds>(et-startTime).count();

            output = to_string(i) + "th CS Exit by Reader Thread " + to_string(id) + " at " + to_string(exitTime) + "\n";
            write_output(output);

            //<REMAINDER SECTION>

            // Simulate a thread executing in Remainder Section
            int randRemTime = generateDelay(urem);
            this_thread::sleep_for(chrono::milliseconds(randRemTime));

            //time taken to get entry since requested
            reader_time[id][i] = chrono::duration_cast<chrono::microseconds>(ent - rt).count();
        }
    }
```

Common Writer Code

This Code is Common to Both Programs. The Only Difference is the Entry And Exit Section of the Critical Section. The Reader Thread id is passed by the main function, it ranges from 0 to nw-1. Each writer thread enters the CS kw times. The reqTime, enterTime and exitTime are in milliseconds relative to the program start time. The output is written with the `write_output()` function for mutual exclusion. `generateDelay()` function is used to generate randCSTime and randRemTime. The time taken by the writers to access the CS each time is stored in the `int64_t **writer_time` 2D array. The Common Code is as Follows -

Listing 4: write_output

```
void writer(int id) {//id is the writer thread number

//each writer thread will enter the CS kw times
for (int i = 0; i <= kw; i++) {
    //get current system time
    auto rt = std::chrono::system_clock::now();
    //Convert to miliseconds Relative to Program Start Time
    auto reqTime = chrono::duration_cast<std::chrono::microseconds>(rt-startTime).count();//time relative to program start time

    string output = to_string(i) + "th CS request by Writer Thread " + to_string(id) + " at " + to_string(reqTime) + "\n";
    write_output(output);



    //-------------------------------------------------------------------------------------

    /*
     * Write your code for Readers Writers() and Fair Readers Writers() Using Semaphores here.
     */

    //<ENTRY SECTION>

    //Aquire Writer Count Lock
    sem_wait(&wcLock);
    writerCount++;//add writer to the count
    if (writerCount == 1){//if first writer
        //Aquire Reader Entry Lock
        //Dont let any more readers enter now/ block readers
        sem_wait(&readerEntryLock);
    }
    //Release Writer Count Lock
    sem_post(&wcLock);

    //Aquire Resource Lock
    sem_wait(&resourceLock);

    //-------------------------------------------------------------------------------------


    //<CRITICAL SECTION>

    //get entry time relative to start time in miliseconds
    auto ent = std::chrono::system_clock::now();
    auto enterTime = chrono::duration_cast<std::chrono::microseconds>(ent-startTime).count();

```

```
        output = to_string(i) + "th CS Entry by Writer Thread " + to_string(id) + " at " + to_string(enterTime) + "\n";
        write_output(output);

        //sleep for random time(exponential dist with ucs average time) in CS
        int randCSTime = generateDelay(ucs);
        // simulate a thread writing in CS
        this_thread::sleep_for(chrono::milliseconds(randCSTime));


        // ------------------------------------------------------------------------------

        /*
         * Your code for the thread to exit the CS.
         */

        //<EXIT SECTION>

        //Release Resource Lock
        sem_post(&resourceLock);//Let Other writers in

        //Aquire Writer Count Lock
        sem_wait(&wcLock);
        writerCount--;//one less writer waiting
        if (writerCount == 0){//if last writer
            //Release Reader Entry Lock
            sem_post(&readerEntryLock);//Let readers in
        }
        //Release Writer Count Lock
        sem_post(&wcLock);


        // ------------------------------------------------------------------------------

        //get exit time relative to start time in miliseconds
        auto et = std::chrono::system_clock::now();
        auto exitTime = chrono::duration_cast<std::chrono::microseconds>(et-startTime).count();

        output = to_string(i) + "th CS Exit by Writer Thread " + to_string(id) + " at " + to_string(exitTime) + "\n";
        write_output(output);

        //<REMAINDER SECTION>

        // simulate a thread executing in Remainder Section
        int randRemTime = generateDelay(urem);
        this_thread::sleep_for(chrono::milliseconds(randRemTime));

        //save time taken to get entry since requested
        writer_time[id][i] = chrono::duration_cast<chrono::microseconds>(ent - rt).count();
    }
}
```

Algorithms Implemented -

a) **Readers-Writers (writer preference)**

In this Algorithm, Once atleast one writer is waiting, no more readers are allowed to enter their Entry Section. Hence, No more new readers are reading or requesting for shared resource. Readers are only allowed to enter their Entry Section when all writers are done and no new writers are waiting. This is implemented with `sem_t readerEntryLock` binary semaphore. Readers are allowed to enter CS with other readers, but not with writers. writers can only enter CS when no other writer or reader is in CS.

Listing 5: Initializations

```
//Initialzations for Readers Writers(writer preference) Problem

int readerCount = 0, writerCount = 0;//Count of readers and writers
  waiting
//all are binary semaphores
sem_t rcLock;//reader count variabe lock
sem_t wcLock;//writer count variabe lock
sem_t readerEntryLock;//Reader Entry section lock
sem_t resourceLock;//Lock to the shared resource

chrono::system_clock::time_point startTime;//start time of the program
```

1. Reader

Listing 6: CS Entry Section

```
//<ENTRY SECTION>

//Aquire Reader Entry Lock
sem_wait(&readerEntryLock);

//Aquire Reader Count Lock
sem_wait(&rcLock);
readerCount++;//add reader to the count
if (readerCount == 1){//if first reader
    //Aquire Resource Lock
    sem_wait(&resourceLock);
}
//Release Reader Count Lock
sem_post(&rcLock);

//Release Reader Entry Lock
sem_post(&readerEntryLock);
```

Listing 7: CS Exit Section

```
//<EXIT SECTION>

//Aquire Reader Count Lock
sem_wait(&rcLock);
//One less reader
readerCount--;
if (readerCount == 0){//if last reader
    //Release Resource Lock
    sem_post(&resourceLock);
}
//Release Reader Count Lock
sem_post(&rcLock);
```

2. Writer

Listing 8: CS Entry Section

```
    //<ENTRY SECTION>

    //Aquire Writer Count Lock
    sem_wait(&wcLock);
    writerCount++;//add writer to the count
    if (writerCount == 1){//if first writer
        //Aquire Reader Entry Lock
        //Dont let any more readers enter now/ block readers
        sem_wait(&readerEntryLock);
    }
    //Release Writer Count Lock
    sem_post(&wcLock);

    //Aquire Resource Lock
    sem_wait(&resourceLock);
```

Listing 9: CS Exit Section

```
    //<EXIT SECTION>

    //Release Resource Lock
    sem_post(&resourceLock);//Let Other writers in

    //Aquire Writer Count Lock
    sem_wait(&wcLock);
    writerCount--;//one less writer waiting
    if (writerCount == 0){//if last writer
        //Release Reader Entry Lock
        sem_post(&readerEntryLock);//Let readers in
    }
    //Release Writer Count Lock
    sem_post(&wcLock);
```

b) Fair Readers-Writers

In this Algorithm, We Use a Queue to Give FIFO Access to the Entry Section. With this no two threads are in the entry section at the same time. This Results Into a Fair Access to the Shared Resource.

If Currently a Writer is in the CS, the first in line thread will wait for the writer to exit the CS in its Entry Section. Till the first in line thread does not exit its entry section, no other thread enters the Entry Section. While a Writer is in the CS no Other Thread Enters the CS.

If Currently a Reader is in the CS and the first in line thread is a reader then it will enter the CS with previous reader. After this Reader Thread Enters CS, next in line thread enters the Entry Section. While a Reader is in the CS, other Readers can enter the CS with the previous reader as long as they are contiguously waiting in the queue.

If Currently Readers are in the CS and next in line thread is a writer then it will wait for the readers to exit the CS in writer threads Entry Section. Till the writer thread does not exit its entry section no other threads enter their Entry Section. While Readers are in the CS, no Writer Enters the CS. Even if there are Readers waiting in the queue just after the first in line writer, they don't enter the CS till the first in line writer enters the CS and is done writing.

Hence, this Algorithm is Fair to Both Readers and Writers.

Listing 10: Initializations

```
    //Initialzations for Fair Readers Writers Problem
```

```
    
    int readerCount = 0;//Number of readers in the CS

    // all semaphores are binary semaphores
    sem_t resourceLock;//Lock to the shared resource
    sem_t rcLock;//reader count variabe lock
    sem_t queueLock;//lock for the queue

    //Waitlist Queue for threads
    queue<int> waitlist;

    //Atomically adds elements to the end of the queue
    void enqueue(int id){
        //Aquire Queue Lock
        sem_wait(&queueLock);

        waitlist.push(id);

        //Release Queue Lock
        sem_post(&queueLock);
    }

    //Atomically removes the first element from the queue
    void dequeue(){
        //Aquire Queue Lock
        sem_wait(&queueLock);

        waitlist.pop();

        //Release Queue Lock
        sem_post(&queueLock);
    }

    chrono::system_clock::time_point startTime;//Stores the start time of
       the program
```

1. Reader

Listing 11: CS Entry Section

```
    //<ENTRY SECTION>

    //add itself to the waitlist queue
    enqueue(id+nw);//to differentiate from writer threads
    //wait till its turn
    while(waitlist.front() != id+nw){//if not first in the queue
        //wait for turn
    }
    //This threads turn

    //Aquire Reader Count Lock
    sem_wait(&rcLock);
    //One more reader
    readerCount++;
    if (readerCount == 1){//if first reader
        //Aquire Resource Lock
        sem_wait(&resourceLock);
    }
```

```
    //Release Reader Count Lock
    sem_post(&rcLock);

    //Let next thread in the queue pass
    dequeue();
```

Listing 12: CS Exit Section

```
    //<EXIT SECTION>

    //Aquire Reader Count Lock
    sem_wait(&rcLock);
    //One less reader
    readerCount--;
    if (readerCount == 0){//if last reader
        //Release Resource Lock
        sem_post(&resourceLock);
    }
    //Release Reader Count Lock
    sem_post(&rcLock);
```

2. Writer

Listing 13: CS Entry Section

```
    //<ENTRY SECTION>

    //Add itself to the waitlist queue
    enqueue(id);
    //wait till its turn
    while(waitlist.front() != id){//if not first in the queue
        //wait for turn
    }
    //This threads turn

    //Aquire resource lock
    sem_wait(&resourceLock);

    //Let next thread in the queue pass
    dequeue();
```

Listing 14: CS Exit Section

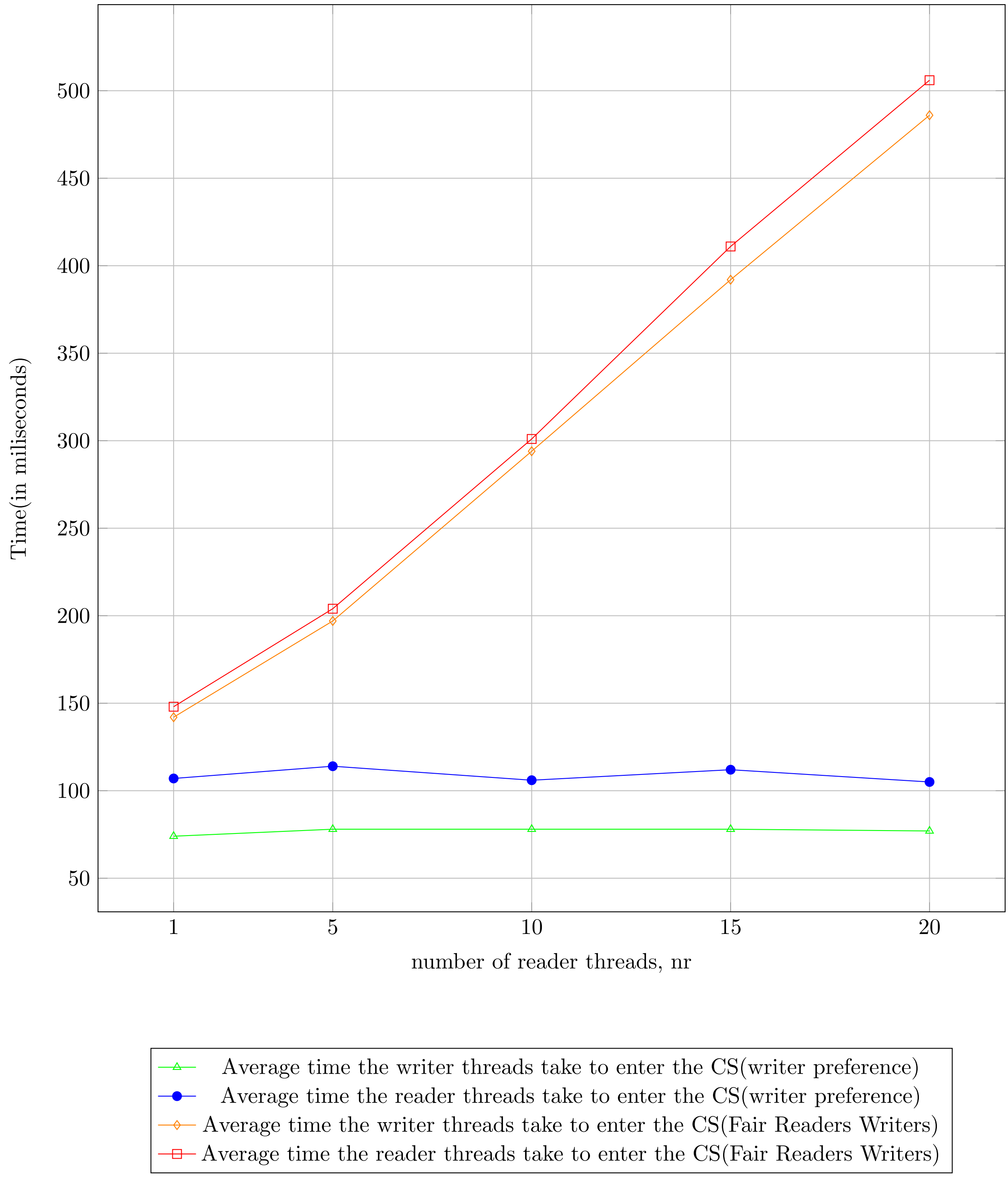
```
    //<EXIT SECTION>

    //Release Resource Lock
    sem_post(&resourceLock);//Can be Aquired by next in line thread
```

II) Time Analysis

1. Average Waiting Times with Constant Writers:

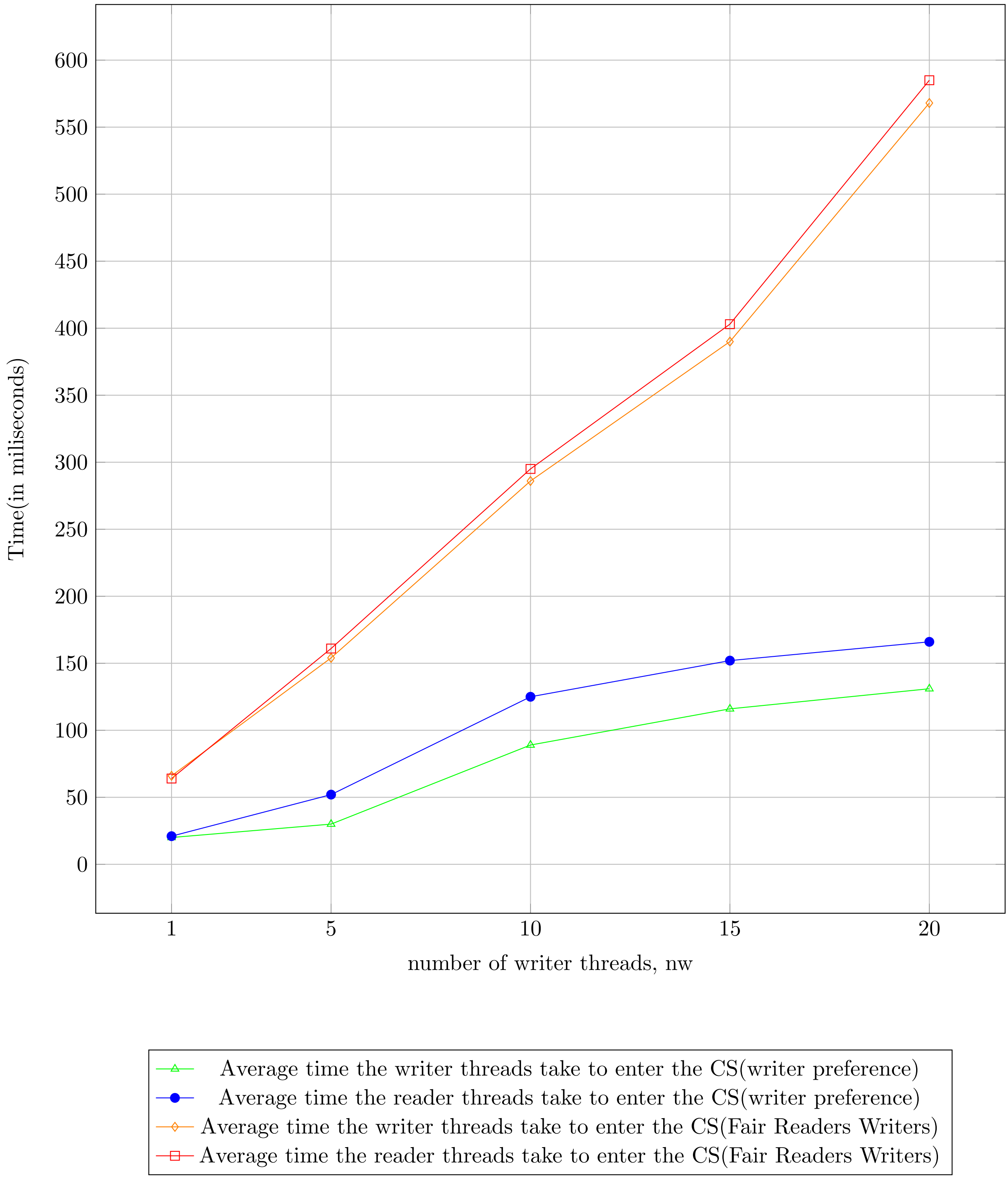
```
nw = 10
kr = 10
kw = 10
μCS = 10
μRem = 5
```



Sr	Observation
1	Writers take least average access time in writer preference algorithm over all nr.
2	Average time the writer threads take to enter the CS(writer preference) is nearly the same for all values of nr. This is because the algorithm is writer preference, so readers do not affect the average time for writers as much. Most Readers enter CS only after all writers are done executing.
3	Average time the reader threads take to enter the CS(writer preference) is nearly the same for all values of nr. This is because in this particular implementation all writer and reader threads are invoked at once, so all/most readers enter CS only after all writer threads are done executing. Hence increasing reader threads does not increase average time as all readers can enter CS at Once.
4	Readers and Writers take more time in `Fair Readers and Writers` compared to `Writer preference Algorithm` due to the spin wait and queue present in it.
5	Average Access Time for Reader is nearly the same as the Writer in `Fair Readers and Writers` over all values of nr. This is Because this Algorithm is fair.
6	As nr increases, Average access time for readers and writers increases in `Fair Readers and Writers`. This is beacuse writer threads are increasing and this Algoritm gives fair access to each thread using a queue.

2. Average Waiting Times with Constant Readers:

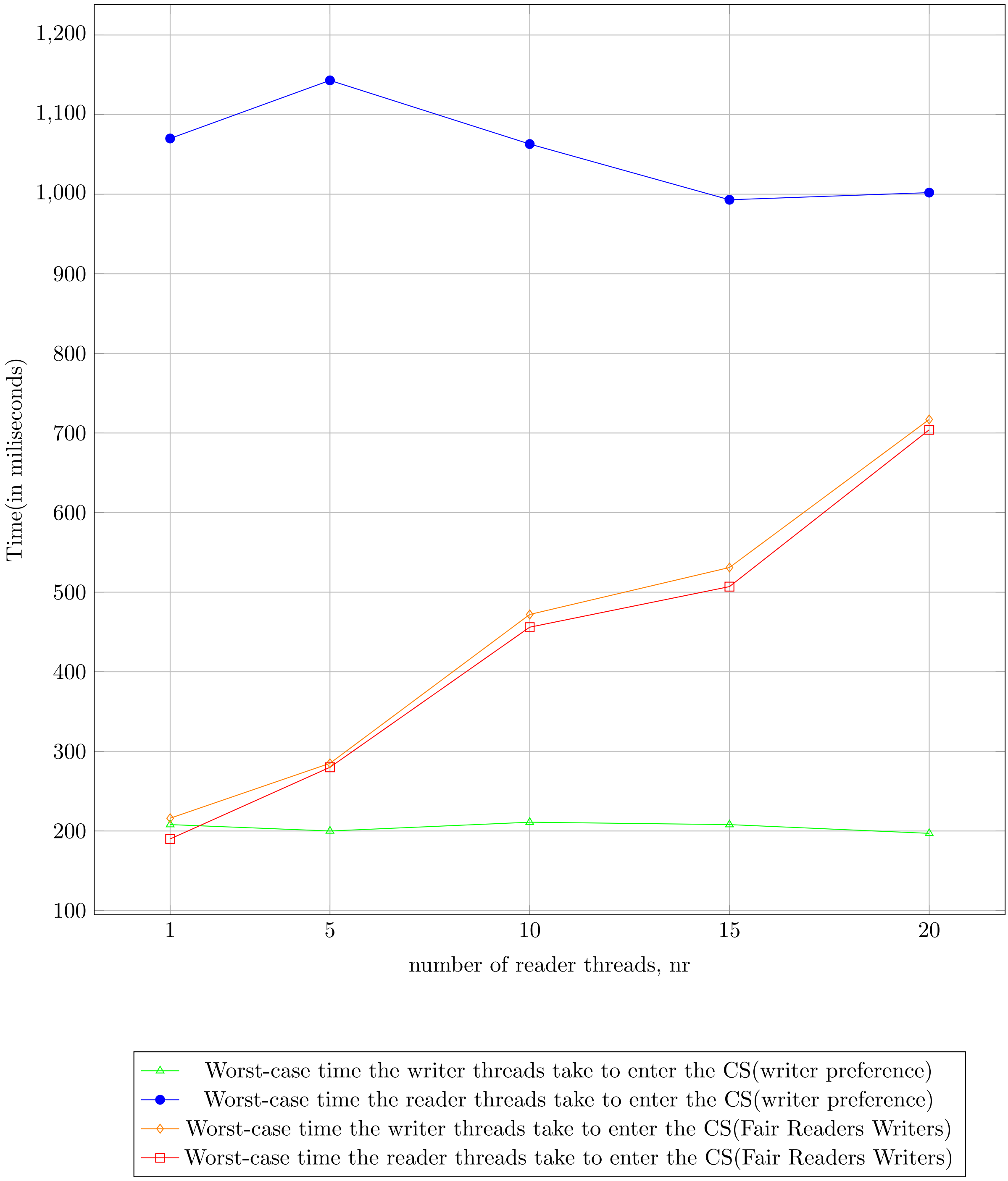
```
nr = 10
kr = 10
kw = 10
μCS = 10
μRem = 5
```



Sr	Observation
1	Writers take least average access time in writer preference algorithm over all nw.
2	As nw increases Average access time in `Writer preference` Algorithm increases. This is beacause number of writer threads increases and algorithm is writer prefernce.
3	Readers and Writers take more time in `Fair Readers and Writers` compared to `Writer preference Algorithm` due to the spin wait and queue present in it.
4	Average Access Time for Reader is nearly the same as the Writer in `Fair Readers and Writers` over all values of nw. This is Because this Algorithm is fair.
5	As nw increases, Average access time for readers and writers increases in `Fair Readers and Writers`. This is beacuse reader threads are increasing and this Algoritm gives fair access to each thread using a queue.

3. Worst-case Waiting Times with Constant Writers:

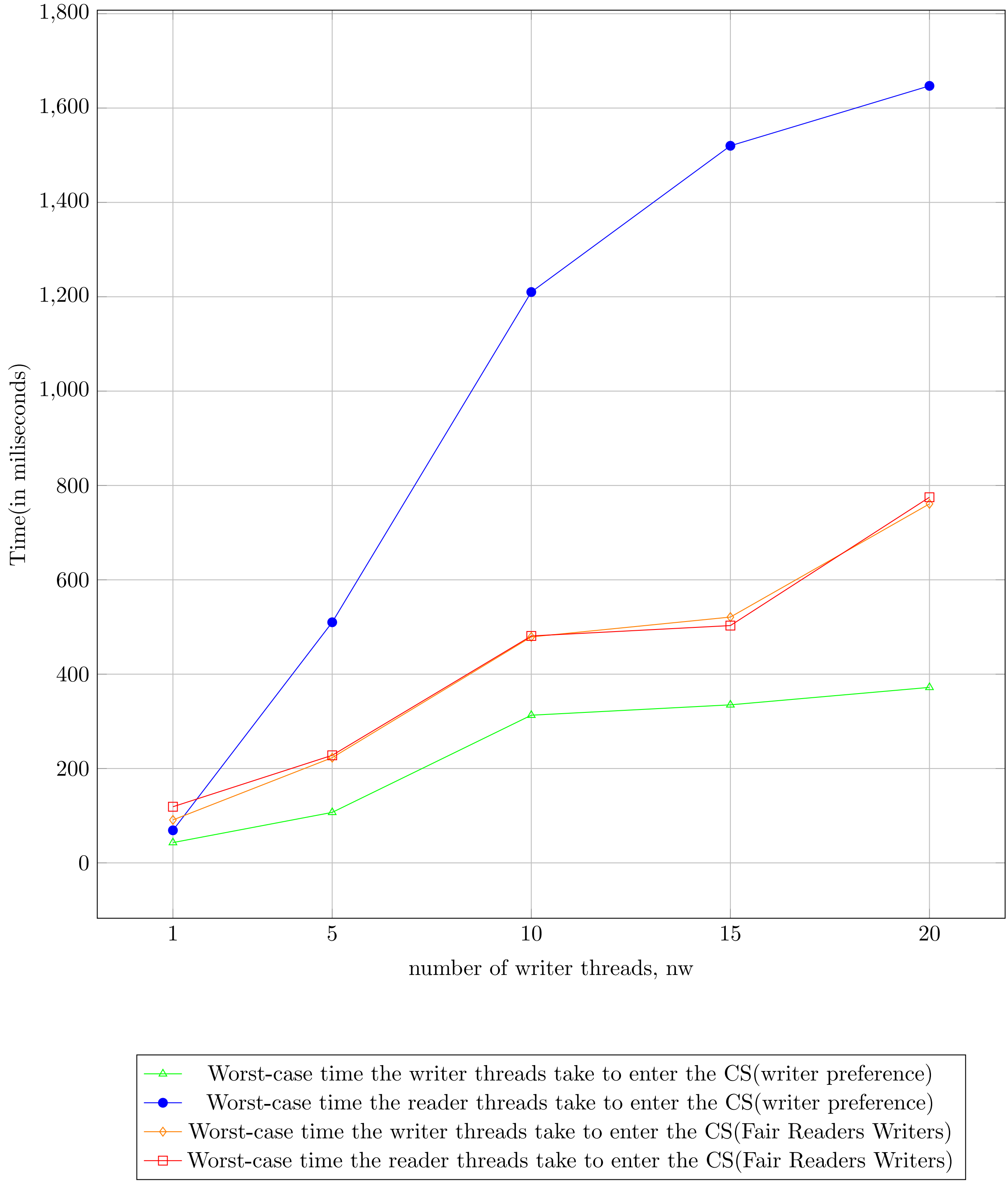
```
nw = 10
kr = 10
kw = 10
```
μCS = 10
μRem = 5



Sr	Observation
1	Writers take least Worst-case access time in writer preference algorithm over all nr.
2	Worst-case time the writer threads take to enter the CS(writer preference) is nearly the same for all values of nr. This is because the algorithm is writer preference, so readers do not affect the average time for writers as much. Most Readers enter CS only after all writers are done executing.
3	Worst-case time the reader threads take to enter the CS(writer preference) is nearly the same for all values of nr. This is because in this particular implementation all writer and reader threads are invoked at once, so all/most readers enter CS only after all writer threads are done executing. Hence increasing reader threads does not increase average time as all readers can enter CS at Once.
4	Writers take more time in `Fair Readers and Writers` compared to writers in `Writer preference Algorithm` because in second algorithm writers are preferred.
5	Readers take more time in `Writer preference Algorithm` compared to readers in `Fair Readers and Writers` because in first algorithm writers are preferred over readers. Worst-case Access time for readers in `Writer preference Algorithm` is significantly greater than all others.
6	Worst-case Access Time for Reader is nearly same as the Writer in `Fair Readers and Writers` over all values of nr. This is Because this Algorithm is fair.
7	As nr increases, Worst-case access time for readers and writers increases in `Fair Readers and Writers`. This is beacuse reader threads are increasing and this Algoritm gives fair access to each thread using a queue.

4. Worst-case Waiting Times with Constant Readers:

```
nr = 10
kr = 10
kw = 10
μCS = 10
μRem = 5
```



Sr	Observation
1	Writers take least Worst-case access time in writer preference algorithm over all nw.
2	As nw increases Worst-case access time in `Writer preference` Algorithm increases. This is beacause number of writer threads increases and algorithm is writer prefernce.
3	Writers take more time in `Fair Readers and Writers` compared to writers in `Writer preference Algorithm` because in second algorithm writers are preferred.
4	Readers take more time in `Writer preference Algorithm` compared to readers in `Fair Readers and Writers` because in first algorithm writers are preferred over readers. Worst-case Access time for readers in `Writer preference Algorithm` is significantly greater than all others.
5	Worst-case Access Time for Reader is nearly same as the Writer in `Fair Readers and Writers` over all values of nw. This is Because this Algorithm is fair.
6	As nw increases, Worst-case access time for readers and writers increases in `Fair Readers and Writers`. This is beacuse writer threads are increasing and this Algoritm gives fair access to each thread using a queue.