

Programming Assignment 4

Implement solutions to Readers-Writers (writer preference) and Fair Readers-Writers problems using Semaphores

Submission Date: 15th March 2024, 9:00 pm

Goal: To solve the **Readers-Writers problem (writer preference)** and **Fair Readers-Writers problem** using Semaphores as discussed in the class in C++. You must implement these two algorithms and compare each thread's average and worst-case time to access the critical section (shared resources).

Details. As mentioned above, you have to implement the Readers-Writers problem (writer preference) and Fair Readers-Writers problem using Semaphores, as discussed in the class in C++. For the writer preference solution, the constraint *no writer, once added to the queue, shall be kept waiting longer than necessary*. Implement a multithreaded program for the above algorithms. Your program will read the input from the file and write the output to the file as shown in the example below.

To test the performance of the synchronization algorithms, develop an application as shown below. Once, the program starts, it creates n_w writer threads and n_r reader threads, which execute their respective writer and reader functions. Each of these threads will access the shared object(or Critical Section), k_w or k_r times, depending on whether they are a writer or a reader, respectively. The pseudocode of the application is as follows:

Listing 1: main thread

```
1 void main()
2 {
3     ...
4     ...
5     create  $n_w$  writer threads;
6     create  $n_r$  reader threads;
7     ...
8     ...
9 }
```

Listing 2: Writer Thread

```
1 void writer()
2 {
3     int id = thread.getID();
4
5     for(int i=0; i ≤  $k_w$  ;i++)
```

```

6      {
7
8      reqTime = getSysTime();
9      cout << i << "th CS request by Writer Thread " << id <<
10     " at " << reqTime << endl;
11
12     /*
13     Write your code for Readers_Writers() and
14     Fair Readers_Writers() Using Semaphores here.
15     */
16
17     enterTime = getSysTime();
18     cout << i << "th CS Entry by Writer Thread " << id <<
19     " at " << enterTime << endl;
20     sleep(randCSTime); // simulate a thread writing in CS
21
22     /*
23     Your code for the thread to exit the CS.
24     */
25
26     exitTime = getSysTime();
27     cout << i << "th CS Exit by Writer Thread " << id <<
28     " at " << exitTime << endl;
29     sleep(randRemTime); // simulate a thread executing in Remainder Section
30 }
31 }

```

Listing 3: Reader Thread

```

1
2 void reader()
3 {
4     int id = thread.getID();
5
6
7     for(int i=0; i < kr ;i++)
8     {
9         reqTime = getSysTime();
10        cout << i << " th CS request by Reader Thread " << id <<
11        " at " << reqTime << endl;
12
13        /*
14        Write your code for Readers_Writers() and
15        Fair Readers_Writers() using Semaphores here.
16        */
17
18        enterTime = getSysTime();
19        cout << i << " th CS Entry by Reader Thread " << id << " at
20        " << enterTime << endl;
21
22        sleep(randCSTime); // simulate a thread reading from CS
23
24        /*
25        Your code for the thread to exit the CS.
26        */
27

```

```

28         exitTime = getSysTime();
29         cout << i << " th CS Exit by Reader Thread "<< id <<" at
30         "<< exitTime << endl;
31
32         sleep(randRemTime); // simulate a thread executing in Remainder Section
33     }
34 }

```

Here *randCSTime* and *randRemTime* are delay values exponentially distributed with an average of μ_{CS}, μ_{Rem} milli-seconds. These time delays aim to simulate these threads performing some complicated, time-consuming tasks.

A sample output would be as follows:

```

1st CS Request by Writer Thread 1 at 01:00
1st CS Entry by Writer Thread 1 at 01:01
1st CS Request by Reader Thread 2 at 01:02
1st CS Exit by Writer Thread 1 at 01:03
1st CS Entry by Reader Thread 2 at 01:04
1st CS Request by Reader Thread 3 at 01:05
1st CS Entry by Reader Thread 3 at 01:06
.
.
.

```

Input: The input to the program will be a file named *inp-params.txt*, consisting of the parameters discussed above, which are - *nw*: the number of writer threads, *nr*: the number of reader threads, *kw*: the number of times each writer thread tries to enter the CS, *kr*: the number of times each reader thread tries to enter the CS, μ_{CS}, μ_{Rem} as described above.

Output: Your program must generate an output in the format to a file. A sample output would be as follows:

```

1st CS Request by Writer Thread 1 at 01:00
1st CS Entry by Writer Thread 1 at 01:01
1st CS Request by Reader Thread 2 at 01:02
1st CS Exit by Writer Thread 1 at 01:03
1st CS Entry by Reader Thread 2 at 01:04
1st CS Request by Reader Thread 3 at 01:05
1st CS Entry by Reader Thread 3 at 01:06
.
.
.

```

The output should demonstrate that writer and reader threads are accessing the critical section mutually exclusively. But it should allow multiple reader threads at the same time.

Your program should output the following files:

1. You must display the log of all the events as shown for each of the algorithms. So, your program must generate two output files: *RW-log.txt* and *FairRW-log.txt*, consisting of events, as described above.
2. *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.

Report: You must submit a report and readme for this assignment. The readme will explain how to compile and run your program.

The report should explain the design of your program. It should also contain a comparison graph of the performance of the two algorithms. You must run both of these algorithms five times to compare their performances and generate the following results graphs:

1. Average Waiting Times with Constant Writers: In this graph, you 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:
 - (a) Average time the reader threads take to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
 - (b) Average time the writer threads take to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
2. Average Waiting Times with Constant Readers: In this graph, you 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. Specifically, the graph will have four curves:
 - (a) Average time the reader threads take to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
 - (b) Average time the writer threads take to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
3. Worst-case Waiting Times with Constant Writers: This graph will be similar to the graph in Step 1. In this graph, you 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:
 - (a) Worst-case time taken by the reader threads to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
 - (b) Worst-case time taken by the writer threads to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
4. Worst-case Waiting Times with Constant Readers: This graph will be similar to the graph in Step 2. In this graph, you 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 graph will have four curves:

- (a) Worst-case time taken by the reader threads to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().
- (b) Worst-case time taken by the writer threads to enter the CS for each algorithm: Readers_Writers() and Fair Readers_Writers().

Thus, all four graphs will contain four curves. The report should also explain the behavior of the graphs.

Deliverables: You have to submit the following:

- The source file containing the actual program to execute named as rw-<rollno>.cpp and frw-<rollno>.cpp
- A readme.txt that explains how to execute the program
- The report as explained above.

Zip all the three files and name it as Assn4-<rollno>.zip. Then, upload it on the Google Classroom page of this course. Submit it by the above-mentioned date. Please follow this naming convention. Otherwise, your program will not be evaluated by the TAs.

Grading Policy: We have the following grading policy:

1. Design as described in the report and analysis of the results: 50
2. Execution of the programs based on the description in the readme: 40
3. Code documentation and indentation: 10

As mentioned before, all assignments for this course have the late submission policy of a penalty of 10% each day after the deadline. We will consider a late assignment for a maximum of 6 days. Any submission beyond that will not be considered. **Kindly remember that all submissions are subjected to plagiarism checks.**