Operating Systems - II: CS3523

Programming Assignment - III

Implement solutions to Readers-Writers & Fair Readers-Writers problems using Semaphores

Assignment Report

Sagar Jain - CS17BTECH11034

March 31, 2019

Contents

Program Design

Both the algorithms i.e. **Readers-Writers** and **Fair Readers-Writers** have been implemented using Semaphores from the header **semaphore.h**. Implementation details for both the solutions is given below along with code-snippets.

Reader-Writers

Semaphores have been implemented using the header ***semaphore.h***

1. Readers-Writers is solved using the two semaphores ***rw_mutex***, ***mutex*** and an integer read_count.

2. The semaphore ***mutex*** is used to avoid race condition when accessing the variable ***read_count***.

3. The variable ***read_count*** is used to keep a count on the number of readers, when the number of readers is ≥ 1, then the semaphore ***rw_mutex*** makes sure that no writer can access its critical section.

4. Once the number of readers is 0, the semaphore ***rw_mutex*** is signalled which gives a chance to the writers to enter their critical section, all this is accomplished by putting conditions on the variable ***read_count*** everytime a new reader attempts to enter its critical section.

5. **Algorithm**

   This is how the above can be implemented in C++.

   For Reader:

```
sem_wait(&mutex);
read_count++;
if (read_count == 1)
        sem_wait(&rw_mutex);
sem_post(&mutex);
// Critical Section
sem_wait(&mutex);
read_count--;
if(read_count == 0)
        sem_post(&rw_mutex);
sem_post(&mutex);
```

For Writer:

```
sem_wait(&rw_mutex);
// Critical Section
sem_post(&rw_mutex);
```

Fair Readers-Writers

1. Fair Readers-Writers is implemented using the semaphores ***in***, ***out***, ***wrt*** and the integers ***ctrin***, ***ctrout***.

2. In the previous solution there exists the possibility of writer threads starving when new reader threads keep requesting to enter the critical section.

3. This problem is solved in this approach by using seperate semaphores for entry and exit and the boolean ***wait***. Once a writer thread requests for entry into the critical section, no new reader thread can enter the critical section before the writer thread finishes since the writer thread blocks the reader threads using the semaphore ***in***.

4. The reader threads are allowed to exit once the writer thread requests entry into the critical section, then through the semaphore ***wrt***, the writer thread is notified that it is safe to enter the critical section.

5. ***out*** is used to ensure no race condition in the access to ctrin, ctrout and wait.

6. **Algorithm**

   For Reader:

   ```
   sem_wait(&in);
   ctrin++;
   sem_post(&in);
   // Critical Section
   sem_wait(&out);
   ctrout++;
   if(wait == 1 && ctrin == ctrout)
           sem_post(&wrt);
   sem_post(&out);
   ```

   Similarly, for **Writer**.

```
sem_wait(&in);
sem_wait(&out);
if(ctrin == ctrout)
        sem_post(&out);
else {
        wait = 1;
        sem_post(&out);
        sem_wait(&wrt);
        wait = 0;
}
// Critical Section
sem_post(&in);
```

Mutually Common Elements

The following code elements are used in both the algorithms:

1. ***get_formatted_time***, this function is used to get the current time in the format hh::mm::ss in the form of a string, it is used to output the logs in the output files.
2. To generate random numbers, I have used ***default_random_engine*** and ***exponential_distribution*** from the header **random**. To seed the random generators, I have used ***random_device()***.
3. To output to the log file, I have used fprintf since it does not have any concurrency related issues.
4. To create threads, I have used ***pthreads*** and have passed a pointer to integers from an integer arrays to give IDs to the threads.
5. To measure time in high resolution, I have used ***chrono***.

Program Output

Each of the programs output two files. One file is the log file containing discrete events of the threads entering and exiting the critical section and another file is the file which contains the statistics related to the waiting times of the threads on the semaphores like average waiting times of the reader and writer threads.

Example Output:

Log File:

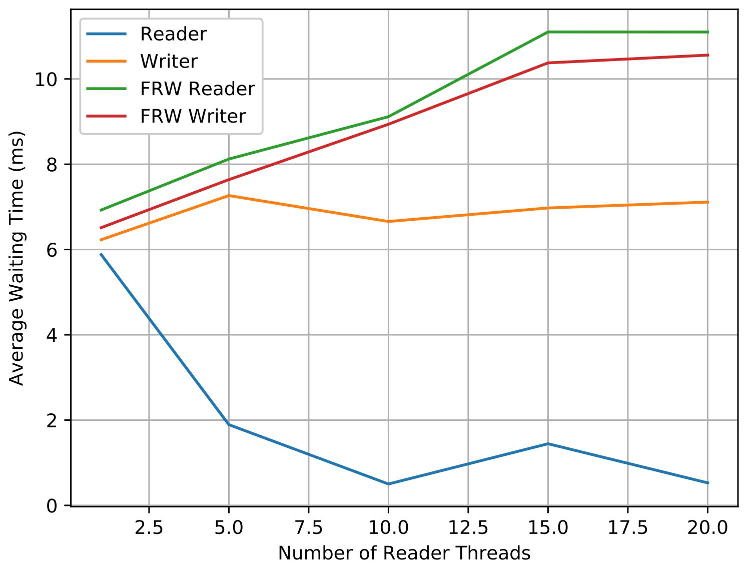
```
4 th CS entry by Reader thread 3 at 01:46:42.
5 th CS entry by Reader thread 5 at 01:46:42.
5 th CS entry by Reader thread 1 at 01:46:42.
5 th CS entry by Reader thread 6 at 01:46:42.
4 th CS entry by Reader thread 0 at 01:46:42.
5 th CS entry by Reader thread 2 at 01:46:42.
5 th CS exit by Reader thread 6 at 01:46:42.
```

Average File:

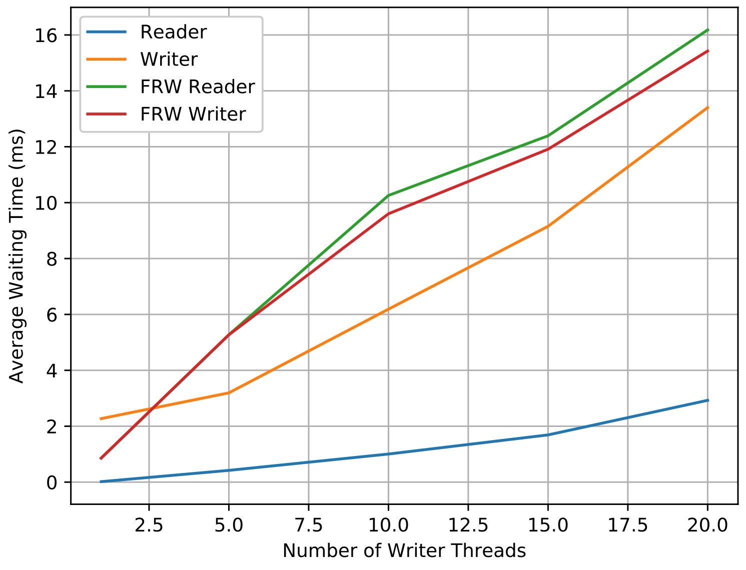
```
Average Time Taken for Writer Threads 7.42405 ms
Average Time taken for Reader Threads 0.80541 ms
Worst Waiting Time for Writer Threads 32.8 ms
Worst Waiting times for Reader Threads 12.405 ms
```

Results & Graphs

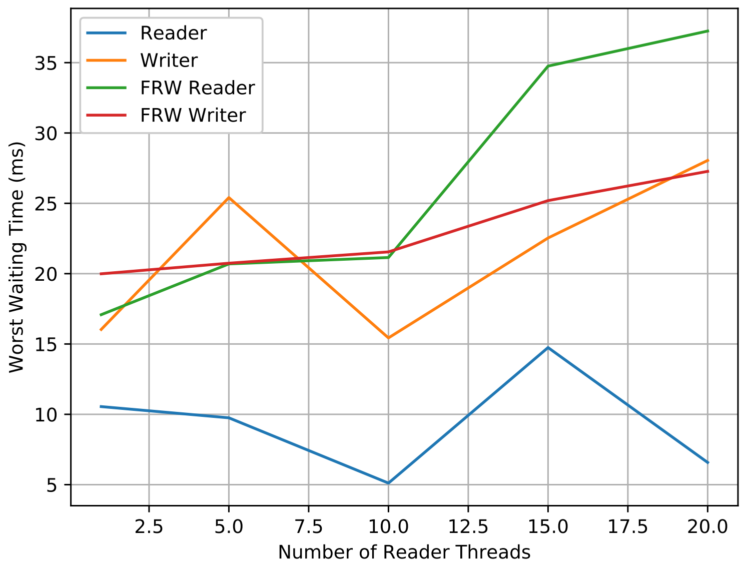
Average Waiting Times with Constant Writers



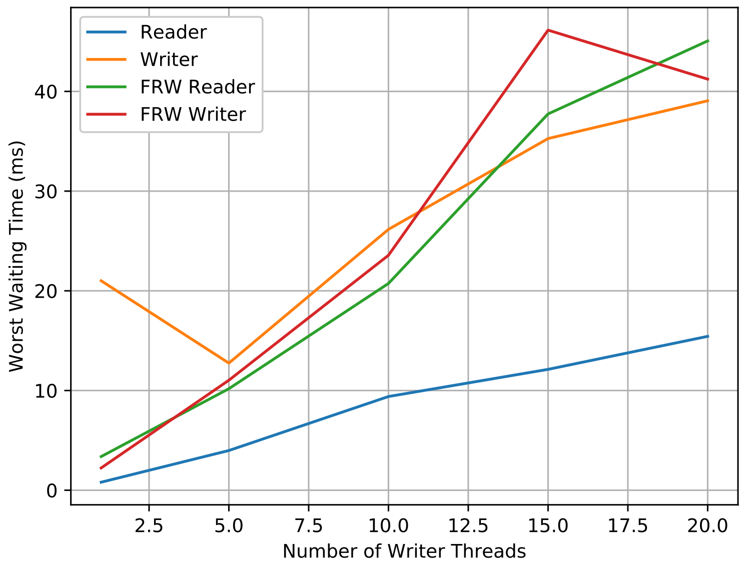
Average Waiting Times with Constant Readers



Worst-case Waiting Times with Constant Writers



Worst-case Waiting Times with Constant Readers



Explaination of Results

1. **Average Waiting Times With Constant Writers**

    (a) With increasing number of Reader threads, all the curves show an increase in the average waiting times except reader threads in normal Readers-Writers.

    (b) In general, the **waiting times for the Fair Readers-Writers Algorithm are more than that for Readers-Writers since it has more instructions**.

    (c) **The difference in the waiting times of the reader and writer threads for Fair Readers-Writers is almost negligible when compared to Normal Readers-Writers**, this shows the effectiveness of the FRW algorithm, and the starvation of writers in RW algorithm.

2. **Average Waiting Times With Constant Readers**

    (a) This graph also tells pretty much the same story, except the fact that all the curves start off from a common low starting point, this is not unusual since we started off with a small number of writer threads and these are the threads that **contribute most to the waiting times through a convoy effect**.

    (b) Once again, the divergence in the waiting times of reader and writer threads of the normal RW algorithm is clearly visible. The writers starve more often than not.

    (c) The fair readers writers show a smooth increase in the waiting times but at the same time have **very low difference in the waiting times of the reader and writer threads**.

3. **Worst Case Waiting Times With Constant Writers & Readers**

    (a) Ideally, with increasing number of threads, the worst waiting time must also increase and this is exactly what we see in the curves of the worst waiting times.

    (b) We can clearly see that the worst waiting times of the writer threads from the normal RW algorithm are in the same range as the fair RW agorithm inspite of the fact that the normal RW algorithm has fewer instructions, this is an **indication of starvation**.

(c) This is also a clear representation of the fact that neither of the threads starve when using the fair RW algorithm.