Operating Systems - 2
Spring 2019

Instructor - Dr. Sathya Peri
Jatin Chauhan - CS17BTECH11019

Design of Program:

The program has been designed to solve the problem of race condition in reader_writer situation, with proper synchronization using **semaphores in both - fair and unfair situation.**
The libraries used in the implementation are -

<pthread.h> - for creating threads
<**semaphore.h>** - for creating semaphores
<random> - To create random number generating engines.
<unistd.h> - To calculate current system time.
<fstream> - Creating file streams.

The reader-writer problem is the situation of race condition when some threads are writing to file(s) and some other threads are trying to read from the file(s). To ensure proper synchronization, one has to ensure that no other thread is accessing the file while any writer thread is writing. Also no writer thread should be allowed to access the file when one or many reader threads are accessing the file. Therefore the use of proper synchronization in inevitable, lest the data should be misread or may corrupt in case of serious operations.

Implementation of Program:

The implementation of both the programs follow the same structure, except a minor change in **fair and unfair** case (discussed later).

Threads are created at the start of main function, along with their attributes. Various reader and writer threads are created, as the count is taken in form of argument. The *func_writer* function operates over writer threads, while the *func_reader* operates over reader threads.

Semaphores are created to ensure synchronization in both the cases. In unfair situation 2 semaphores - **rw_mutex and mutex** are created. The **rw_mutex** is to alter between the writer and the reader threads. The **mutex** semaphore is to ensure synchronization among multiple reader threads. In case of **fair** implementation another semaphore - **input** is created to ensure fairness among the threads trying to enter their CS.

At the start of the writer thread function, the time is saved in a variable which is determined as the time for the request of **CS** and is written to the log file. The thread then waits for semaphore **rw_mutex** to be acquired. After acquiring, it writes the time in the log file and performs the write operation. Thereafter it releases the semaphore and sleeps for some time to determine some operation of remainder section.
In the case of fair implementation, it has an additional semaphore named - **input**. The thread first acquires this semaphore and within this it acquires the **rw_mutex** one. The **input** semaphore is used to ensure fairness, for once a writer thread requests to enter the CS, this semaphore ensures that after finishing the remaining reader threads this specific writer thread will gain access to its CS. Here it is assumed that the semaphore waiting data structure ensures fairness among its entries. At the end of the CS, the writer thread first releases the **rw_mutex,** and thereafter it releases the **input** semaphore.

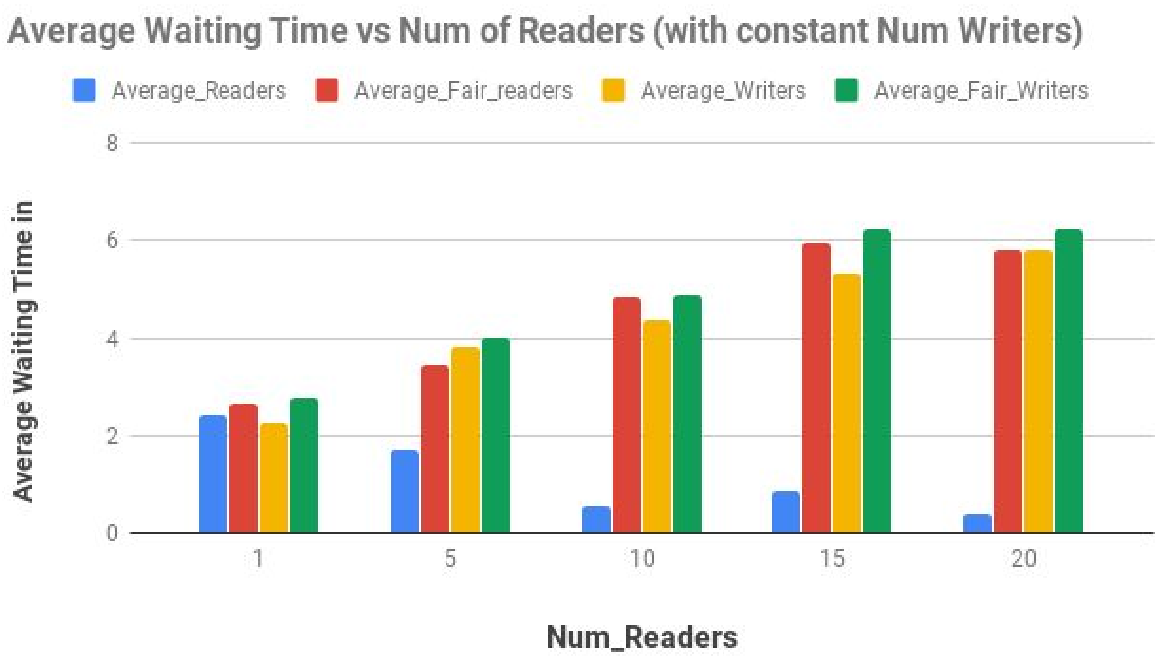
In the reader function, again the start time is saved in some variable and written to log file. It then tries to acquire the **mutex** semaphore. This is to allow multiple reader threads to gain access to their CS simultaneously. After it has acquired **mutex**, it checks if this is the first reader thread for now to ask for entry in its CS. If so, it increments the **READ_COUNT**

variable and tries to acquire **rw_mutex**, otherwise it directly increments the **READ_COUNT** and releases the mutex. After acquiring **rw_mutex,** it releases the **mutex** and enters it CS. After finishing the read operation, it again tries to acquire the **mutex** and then decrements the **READ_COUNT** variable. If it becomes zero, it releases the **rw_mutex.** Finally, it releases the **mutex** and enters the remainder section.
In case of fair implementation, it first tries to acquire the **input** semaphore, within which it acquires the **mutex** and after releasing the **mutex,** it releases the **input** semaphore. Rest is same.

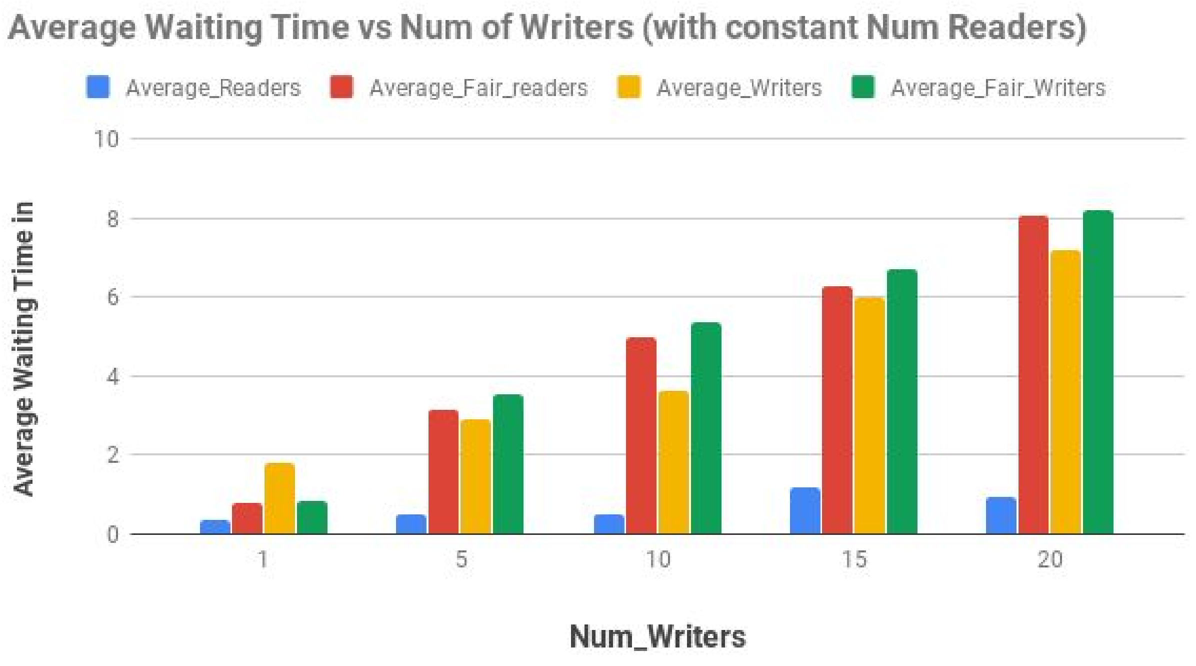
Graph:

Multiple graphs for average waiting time and worst case waiting time are plotted for various cases - varying number of readers while fixing the number of writers and varying number of writers while fixing the number of readers. In all the cases the fixed values are all 10, and the exponential distribution parameters are fixed to 1. The remaining value is varied from 1 to 20 with the interval of 5 after 1.
In all the graphs, the y-axis represents time values in milliseconds and the x-axis represents the variable.

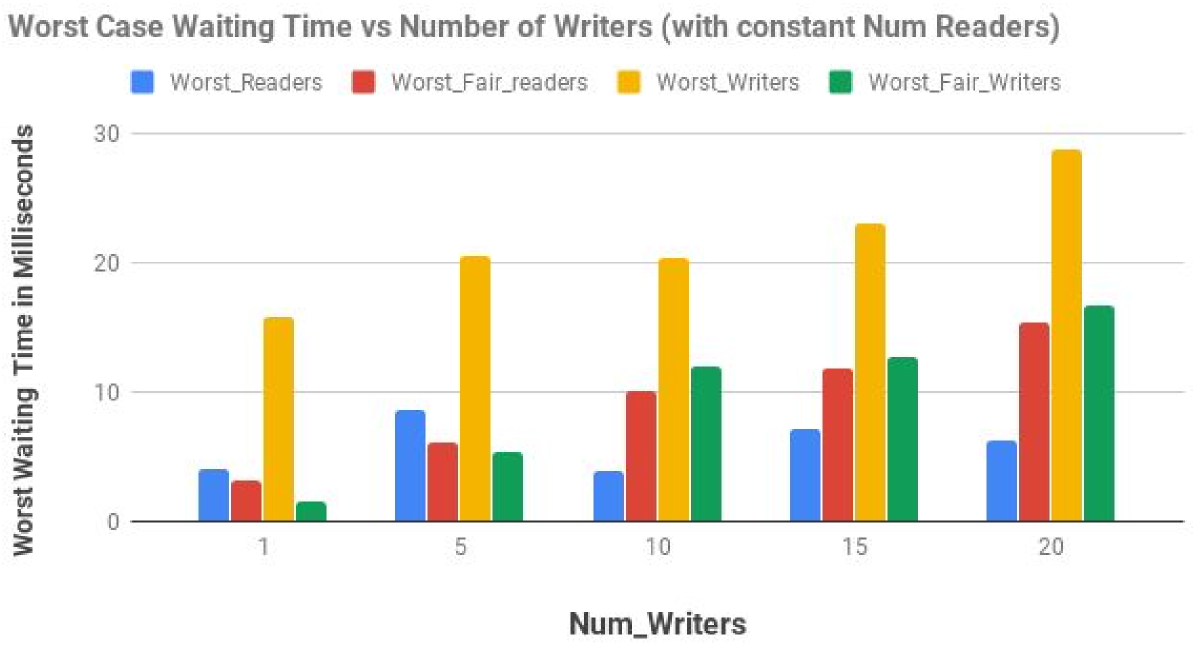
1) Average Waiting Times with Constant Writers:



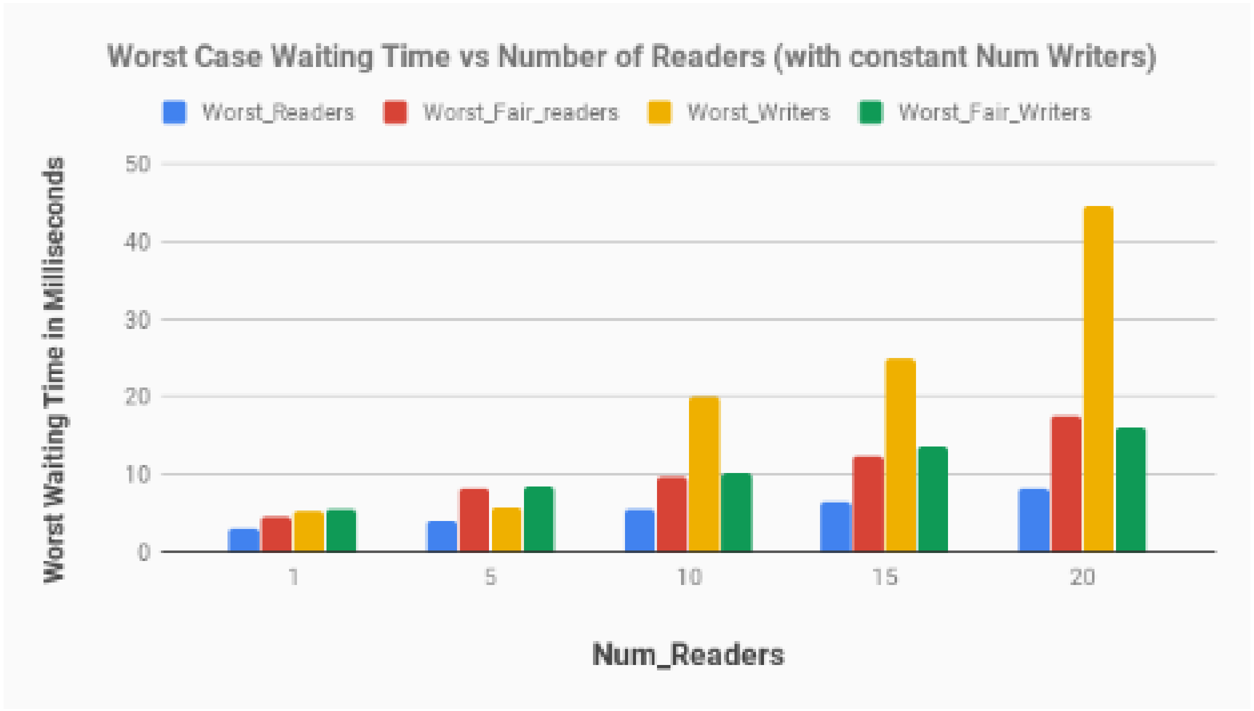
2) Average Waiting Times with Constant Readers:



3) Worst-case Waiting Times with Constant Readers:



4) Worst-case Waiting Times with Constant Writers:



As evident from the graphs of **average waiting times** , reader threads have much less average waiting time as compared to writer threads in **unfair** situation. However, as expected in the **fair** situation the time difference is very less, as both the reader and writer threads follow a fair data structure in the semaphore waiting method. Also, as the number of reader and writer threads increase the difference in average waiting times in **unfair** situation increases. The average waiting time values in the case of constant writers is less than that of constant readers.

The **worst-case waiting time** follow a similar pattern and the worst-case time in **unfair** situation is much larger for the writer threads as compared to reader threads. Again, the difference in **fair** situation is very less. Also, worst-case time in constant writers is higher than in case of constant readers, which should be, since, with the increase in the number of readers, the writer threads might have to wait more (in case of **unfair** situation).