OPERATING SYSTEMS II

PROGRAMMING ASSIGNMENT 4

by: Devraj, ES22BTECH11011

READERS-WRITERS PROBLEM

Common things in the main programs

The variables ***nw***, ***nr***, ***kw***, ***kr***, ***u_CS*** and ***u_rem*** are globally defined.

wait_readers and ***wait_writers*** are dynamically allocated 2-D arrays for storing the wait times of the reader and writer threads respectively.

log_sem is a semaphore used to ensure that only one thread writes to the log file so that it is readable, otherwise a thread could start writing into the file while another thread is writing and the result would be unreadable.

exponential_distribution: ***CS*** and ***rem*** are defined globally and are given their respective values in the main function.

writer() function to simulate writing to a file and ***reader()*** function to simulate reading from a file. Both use the ***sleep_for()*** function to give the impression that some work is being done, while the thread is actually sleeping. ***randCSTime*** and ***randRemTime*** are variables that store the randomly generated sleep durations.

The structure arguments is used to pass the ***thread_no*** (index in the thread array) to the ***reader()*** and ***writer()*** functions.

The function ***max()*** returns the maximum among two double values.

nw writer threads are created first and then the ***nr*** reader threads are created.

The outputs are printed onto files except the worst-case times which are printed onto the terminal.

Writers Preference solution

Four semaphores are used: ***read_sem***, ***write_sem***, ***try_read_sem*** and ***CS_sem***

read_sem is used to protect the variable ***reader_count*** and ensure mutual exclusion for the entry and exit sections the reader function.

Similarly, ***write_sem*** is used to protect the variable ***writer_count*** and ensure mutual exclusion for the entry and exit sections of the writer function.

try_read_sem is used to indicate that a reader is trying to enter the critical section.

CS_sem is there to protect the critical section for both the reader and writer threads.

Entry section of ***reader()***:

Whenever a reader wishes to enter the CS it first checks if the ***try_read_sem*** is free indicating that there are no other readers or writers currently holding it. Then it checks ***read_sem*** to increment ***reader_count***. If it is the first reader then it can request to acquire ***CS_sem***, otherwise it can just access the CS without requesting access since some other reader is currently holding it.

Exit section of ***reader()***:

After the thread has finished its work in the CS (sleeping), it requests to hold ***read_sem*** and decrements ***reader_count*** to indicate its leave. If it is the last reader leaving, it releases ***CS_sem*** so that a writer thread can access it.

Entry section of ***writer()***:

Whenever a writer wishes to enter the CS, it first requests access to ***write_sem*** to increment ***writer_count***. If it is the first writer then it holds ***try_read_sem*** to ensure no new readers can access the CS and then it requests for ***CS_sem***, otherwise it can just go on and request for ***CS_sem*** since no reader threads are currently holding it.

When a writer thread holds ***try_read_sem***, it means that it will wait for all the reader threads that are currently waiting on ***CS_sem*** to finish and no more reader threads can request access for ***CS_sem***.

Exit section of ***writer()***:

After the writer thread has finished its work in the CS, it releases ***CS_sem*** for other writers. Then it holds ***write_sem*** to decrement ***writer_count*** to indicate its leave. If it is the last writer leaving, it releases ***try_read_sem*** to let readers in, otherwise it does not release ***try_read_sem*** since there are other writers waiting for their turn.

This solution ensures that writers do not starve by blocking new readers from entering when a writer is waiting to access the critical section.

Fair solution

Three semaphores used: ***read_sem***, ***CS_sem*** and ***queue_sem***.

read_sem is there to ensure mutual exclusion among readers. It does this by protecting the variable ***reader_count***. It ensures mutual exclusion in the entry and exit sections of the ***reader()*** function.

CS_sem is there to protect the critical section for both the reader and writer threads.

queue_sem is there to ensure fairness in acquiring the critical section. It allows threads to queue up in order of arrival and enter the critical section in a first come, first served manner.

Entry section of ***reader()***:

Whenever a reader wishes to enter the CS, it must first request for ***queue_sem***. Then it requests for ***read_sem*** to enter its entry section and increment ***reader_count*** to indicate its arrival. If it is the first reader to arrive, it requests for ***CS_sem*** and then continues on to its CS, otherwise it does not need to request for ***CS_sem*** since some other reader would be holding it.

Exit section of ***reader()***:

After the reader has finished its work in CS, it requests for ***read_sem*** to decrement ***reader_count*** to indicate its leave. If it is the last reader to leave, it releases ***CS_sem*** for the writers, otherwise it doesn't so that other readers can access the CS.

Entry section of ***writer()***:

First the writer requests for queue_sem and after it acquires it, it requests for CS_sem.

So, a writer can request access to the CS only if it is currently holding queue_sem.

Exit section of ***writer()***:

To exit the CS, a writer simply releases CS_sem so the next thread in queue can request access for it.

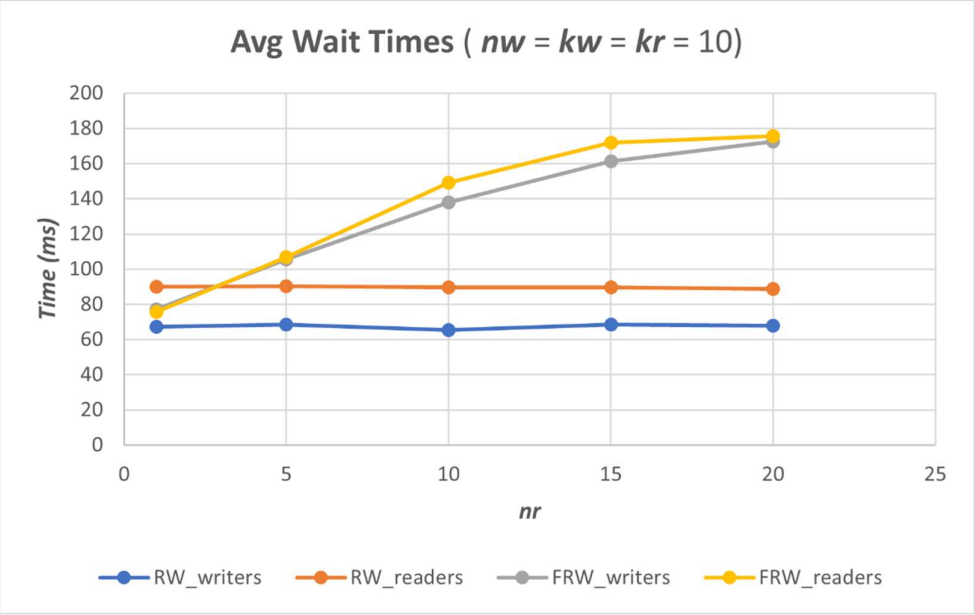
By implementing a queueing mechanism using semaphores and maintaining mutual exclusion within the critical section, this solution ensures fairness in accessing shared resources for both readers and writers in the readers-writers problem. Each thread is given a fair chance to access the critical section without being starved indefinitely.

EXPERIMENTS

Experiment 1

Average Waiting Times with Constant Writers

Avg Wait Times (*nw* = *kw* = *kr* = 10)				
nr	***RW_writers***	***RW_readers***	***FRW_writers***	***FRW_readers***
1	67.2	89.81	77.22	75.62
5	68.38	90.39	105.46	106.89
10	65.44	89.71	138.03	148.99
15	68.42	89.73	161.18	171.86
20	67.94	88.63	172.4	175.73



As we can see, the average wait times for the writers-preference solution remain almost the same for any number of readers (***nr***). While the average wait times for the fair solution seem to be increasing with the number of readers.

This behaviour can be explained as follows:

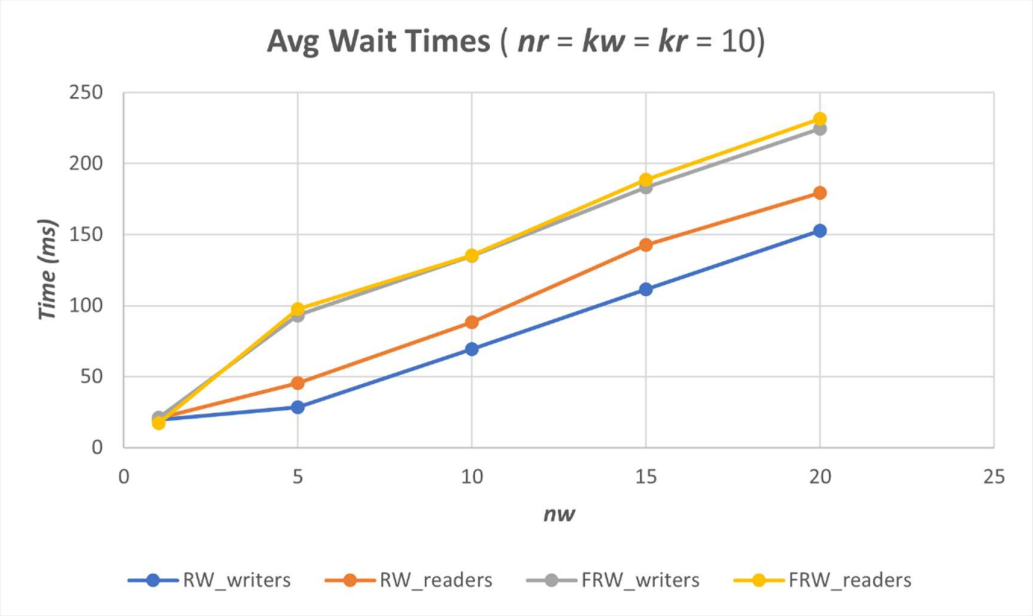
Since in the writer-preference solution the writers can block new incoming threads from entering and whatever be the number of readers, once a writer requests for the CS no new readers can try to access the CS. Also, since preference is given to the writers, their average time is less compared to that of the readers.

And in the fair solution, there is a queue for access of the CS, i.e. both readers and writers are contesting for access. Hence, as the number of readers increases, it increases the average size of queue for both reader and writer threads and they have to wait longer. Also, the times for readers and writers are almost the same since they have equal preference.

Experiment 2

Average Waiting Times with Constant Readers

Avg Wait Times (nr = kw = kr = 10)				
nw	***RW_writers***	***RW_readers***	***FRW_writers***	***FRW_readers***
1	19.67	20.83	21.16	17.27
5	28.57	45.32	93.06	97.56
10	69.18	88.42	135	135.16
15	111.308	142.61	183.35	188.64
20	152.77	179.487	224.3	231.339



In this experiment, we observe that the average wait times for all the cases increase with increase in the number of writers (***nw***).

This behaviour can be explained as follows:

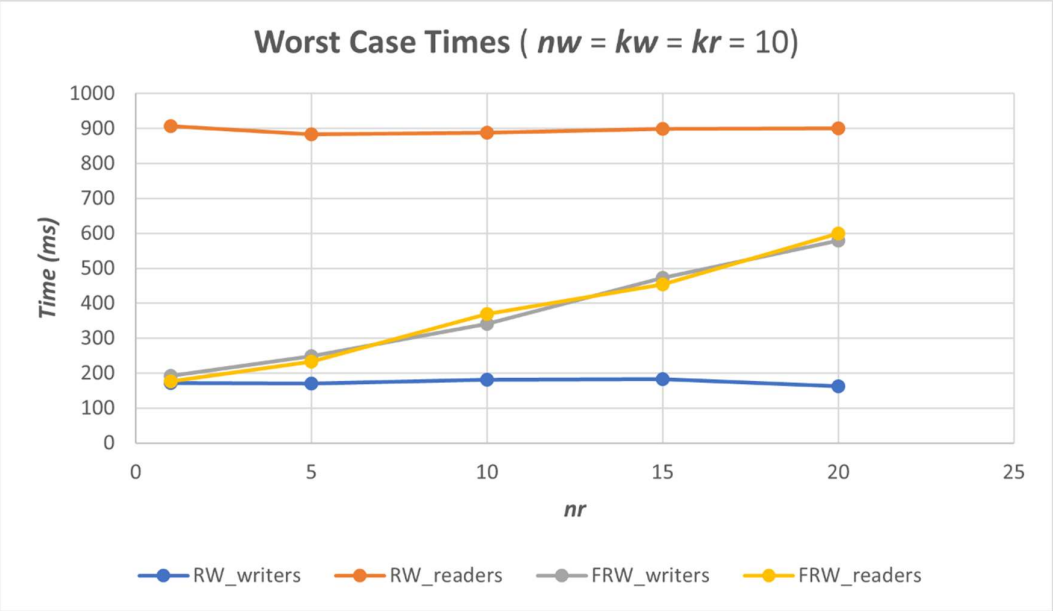
For the writers-preference solution, since the writers are given preference over the readers, the real competition is between the writers to access the CS. Hence, as the number of writers increases the contention for access to the CS also increases. Similarly, for the reader threads, they have to wait for more writers before they can request for access to the CS.

For the fair solution, as the number of writer threads increases the average size of the waiting queue also increases leading to longer wait times for both readers and writers.

Experiment 3

Worst Case Waiting Times with Constant Writers

Worst Case Times (*nw* = *kw* = *kr* = 10)				
nr	***RW_writers***	***RW_readers***	***FRW_writers***	***FRW_readers***
1	172.41	906.562	191.61	176.17
5	170.364	882.8	248.05	233.47
10	181.59	887.28	341.22	368.92
15	182.15	898.26	473.16	454.25
20	163.19	899.72	578.77	599.7



In this experiment, the worst-case times for the writers-preference solution appear to remain constant while those for the fair solution appear to increase with increase in the number of readers (***nr***). This graph also shows the starvation that the readers face in the writers-preference solution.

This behaviour can be explained as follows:

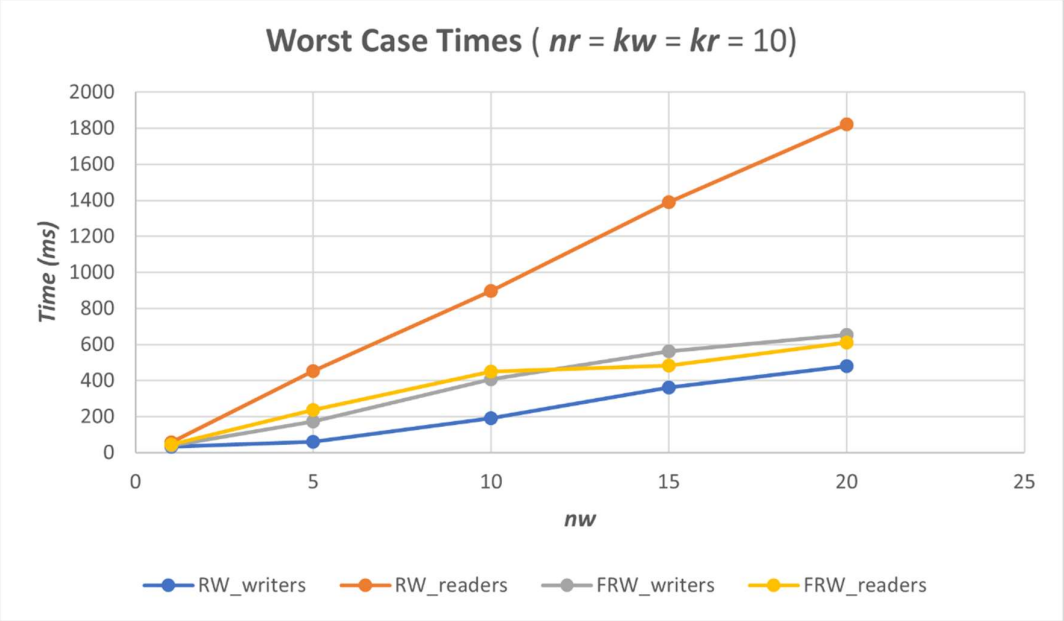
For the writers-preference solution, worst case time doesn't depend on the number of readers since as soon as a writer requests access to the CS, no new readers may further request for it. So, the worst-case wait times for a writer depends on the number of writers that are holding the CS before it. Similarly, for the readers, their worst-case wait times depends on the number of writers that have requested for the CS.

For the fair solution, the worst-case wait times depend on the largest size of queue that can form while executing since both writers and readers have to for those that are ahead of them in the queue. So, since the number of readers is increasing, the largest size of queue will also increase, thus increasing the wait time.

Experiment 4

Worst Case Waiting Times with Constant Readers

Worst Case Times (*nr* = *kw* = *kr* = 10)				
nw	***RW_writers***	***RW_readers***	***FRW_writers***	***FRW_readers***
1	34.42	56.92	39.84	44.07
5	61.76	453.2	173.05	238.29
10	191.46	898.41	406.87	448.64
15	362.91	1391.24	561.46	483.34
20	481.83	1821.62	653.23	611.63



In this experiment, the worst-case wait times for all cases appear to increase with increase in the number of writers (***nw***). This graph also shows that starvation for the readers in the writers-preference solution increases with increase in number of writers.

This behaviour can be explained as follows:

For the writers-preference solution, the writers really only have to wait for the other writers that have requested for access to the CS. So, as the writers increase in number the contention for access to the CS also increases among them and wait times increase. The readers have to wait for the writers to finish their work in the CS, hence as the number of writers increases, the wait times for the readers also increases.

For the fair solution, the worst-case wait times depend on the largest size of queue that can form while executing since both writers and readers have to for those that are ahead of them in the queue. So, since the number of readers is increasing, the largest size of queue will also increase, thus increasing the wait time.