Operating Systems–2: Spring 2024

Programming Assignment 4: Implement solutions to Readers-Writers (writer preference) and Fair Readers-Writers problems using Semaphores

Name : Rishitha Surineni
Roll No : CS22BTECH11050

Introduction

This is a C++ program to implement Readers Writers problem in two variants

1) Preference to Writers
2) Fair solution

Readers-Writers (writer preference)

Implementation

Processing the Input:

The Input consists of
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,
μCS,
μRem

This input is read from a file named “inp-params.txt” and it is stored in corresponding variables.

Creating and Initialising Semaphores:

The following semaphores are used they are declared using sem_t data type

```
sem_t lock1;     // for maintaining count of writers
sem_t lock2;     // for maintaining count of readers
```

sem_t writelock; // for accessing the shared resource
sem_t canread; // for checking if readers can acquire the resource
sem_t file; // for accessing the output file
sem_t time_lock; // for calculating average time of each thread

These semaphores are intialised using **sem_init()** function

sem_init(&lock1, 0, 1)
Initialises the semaphore lock1 with 1. The second parameter 0 implies that the semaphore is shared between threads of the same process.

Creating threads:

Threads are created using the **pthread_create()** function.
A loop is run nw times and in this writer threads are created using
pthread_create(&WriterThreads[i], NULL, Writer, NULL);

WriterThreads is an array of data type pthread_t, these serve as identifiers for the threads, **Writer** is a function which is executed by each writer Thread.
A loop is run nr times and in this reader threads are created using
pthread_create(&ReaderThreads[i], NULL, Writer, NULL);

ReaderThreads is an array of data type pthread_t, these serve as identifiers for the threads, **Reader** is a function which is executed by each Reader Thread.

Reader Function:

Each of the reader threads execute this function
The thread id is obtained by using **gettid()** function.
A loop is run kr times where each iteration is a request for read.
First the thread acquires access to the output file to log its request time.
This is done by waiting on the semaphore **file.**
The time of the request is calculated and is logged onto the output file.

sem_wait() function is used for the thread to wait for the acquiring the semaphore and **sem_post()** is used to signal the semaphore
After this the threads waits to acquire canread semaphore (**sem_wait(&canread)**).
This semaphore is to tell if a reader can access the resource or not. If there is any writer waiting to enter into the Critical section then this reader thread keeps on waiting here till there are no writers waiting .This semaphore is used to give priority to writers.

After the reader thread acquires the **canread** semaphore , it waits on **lock2** semaphore to gain exclusive access to the number of readers. The number of readers is incremented and if this reader is the first reader then it has to acquire the **writelock** to gain access to the shared resource. Lock2 is signaled so that any other reader thread can access the **readers** variable.

The semaphore canread is also signaled so any one of the waiting readers can wake up.

If any reader is already executing its critical section and there is no writer waiting then writelock is already acquired and this incoming reader need not acquire it again as multiple readers can execute critical section at the same time. So this reader executes its critical section.

In the Critical section , the thread gains access to the output file and logs its entry time into it. The time difference between the entry into critical section and the time at which request was placed is calculated.
For simulating a critical section, the threads sleeps for time **randCST**
This delay value is exponentially distributed with an average of μCS milliseconds.
For this implementation random header of c++ is used and a generator is written which when called produces the value according to the given distribution.

Again the semaphore lock2 is acquired to get access to the number of readers .This value is decremented. If there are no more readers in critical section then writelock semaphore is signaled to allow writers to enter into critical section.

The exit time is logged into the output file and then remainder section executes.

To simulate Remainder section the thread is put to sleep for time randRemTime which is calculated same as the above randCST.

After the kr iterations the **average of these time differences** of the kr iterations of a reader thread is calculated and it is stored in a vector named **Readers.** As this Readers vector is a shared resource for all the reader threads another semaphore **time_lock** is used to make sure only one of the reader threads access this vector at a particular instant.

Writer Function:

Each of the writer threads execute this function.
In this function the thread id is obtained by gettid() function.
A loop is run kw times and in each iteration a request for write is made and serviced.
The semaphore file is acquired and then the request time is logged into the output file.Then semaphore lock1 is acquired to access the number of writers variables which is a shared variable among all the writer threads.

After lock1 is acquired the writers variable is incremented . If this writers is the only writer waiting to enter into critical section then sem_wait(&canread) is done so as to not grant access to any new reader which put request after the writer. This is to make sure the writers are given more priority.

After this the writer thread waits on the semaphore writelock. This is to gain access to the shared resource and enter the critical section.

In the critical section the entry time is logged onto the output file and the time difference between the entry and request is calculated.
For simulating a critical section, the threads sleeps for time **randCST**
This thread then signals writelock semaphore.

Again the lock1 semaphore is acquired to gain access to writers variable. The writers variable is decremented and if there are no more writers waiting to access the resource then the semaphore canread is signaled to allow readers to enter the critical section.
This ensures that as long as there are some writers waiting the readers can't execute their critical section implying the priority is given for writers over readers.

The exit time is logged into the output file and then remainder section executes.

To simulate Remainder section the thread is put to sleep for time randRemTime which is calculated same as the above randCST.

After the kw iterations the **average of these time differences** of the kw iterations of a reader thread is calculated and it is stored in a vector named **Writers.** As this Writers vector is a shared resource for all the writer threads another semaphore **time_lock** is used to make sure only one of the writer threads access this vector at a particular instant.

Joining the threads :

The writer threads are joined by using the function pthread_join() in a loop .
And similarly for the readers.
After all the threads are done executing, the average of waiting time for the reader threads and the writer threads is calculated by running for loops on Readers vector and Writers vector.In the similar way worst waiting times are calculated.
The average times are written into an output file.

Output :

Two output files will be created .

1) "RW-log.txt" which contains the request time,entry time,exit time of all the iterations of the threads.
2) "Average-RW.txt" which contains the average waiting time of each Reader thread and of each Writer Thread and also average of all the reader threads and the writer threads.

Readers-Writers (Fair Solution)

Implementation

Processing the Input:

The Input consists of
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,
μCS,
μRem

This input is read from a file named “inp-params.txt” and it is stored in corresponding variables.

Creating and Initialising Semaphores:

The following semaphores are used they are declared using sem_t data type

```
sem_t lock2;     // for maintaining count of readers
sem_t writelock; // for accessing the shared resource
sem_t queue;     // for maintaining queue of processes which made request
sem_t file;      // for accessing the output file
sem_t time_lock; // for calculating average time of each thread
```

These semaphores are intialised using **sem_init()** function

sem_init(&lock2, 0, 1)
Initialises the semaphore lock1 with 1. The second parameter 0 implies that the semaphore is shared between threads of the same process.

Creating threads:

Threads are created using the **pthread_create()** function.
A loop is run nw times and in this writer threads are created using
pthread_create(&WriterThreads[i], NULL, Writer, NULL);

WriterThreads is an array of data type pthread_t, these serve as identifiers for the threads,
Writer is a function which is executed by each writer Thread.
A loop is run nr times and in this reader threads are created using
pthread_create(&ReaderThreads[i], NULL, Writer, NULL);

ReaderThreads is an array of data type pthread_t, these serve as identifiers for the threads,

Reader is a function which is executed by each Reader Thread.

Reader Function:

Each of the reader threads execute this function
The thread id is obtained by using **gettid()** function.
A loop is run kr times where each iteration is a request for read.

First the thread acquires access to the output file to log its request time.
This is done by waiting on the semaphore **file.**
The time of the request is calculated and is logged onto the output file.

sem_wait() function is used for the thread to wait for the acquiring the semaphore and **sem_post()** is used to signal the semaphore

After this the threads waits to acquire queue semaphore (**sem_wait(&queue)**).
This semaphore is used to maintain the list of processes whose request is to be serviced.
As this implementation aims in creating a fair solution which doesn't cause starvation or readers or writers, every reader threads when makes a request has to wait on the semaphore queue to add itself to the list of the processes. When this semaphore is acquired, the threads waits on lock2 semaphore to gain access on readers counter. This readers is incremented and if this reader is the first one then it acquires writelock . If there is already a reader which is executing its CS then there is no need to acquire the writelock again as multiple readers can execute CS at once.After this it signals lock2 and also queue semaphore. This would allow the next thread in the queue to wake up.

In the critical section the entry time is logged onto the output file and the difference between the entry time and request time is calculated. These differences over the kw iterations are used to find average waiting time.

For simulating a critical section, the threads sleeps for time **randCST**
This delay value is exponentially distributed with an average of μCS milliseconds.
For this implementation random header of c++ is used and a generator is written which when called produces the value according to the given distribution.

Again the semaphore lock2 is acquired to get access to the number of readers .This value is decremented. If there are no more readers in critical section then writelock semaphore is signaled.

The exit time is logged into the output file and then remainder section executes.

To simulate Remainder section the thread is put to sleep for time randRemTime which is calculated same as the above randCST.

After the kr iterations the **average of the time differences** of the kr iterations of a reader thread is calculated and it is stored in a vector named **Readers.** As this Readers vector is a

shared resource for all the reader threads another semaphore **time_lock** is used to make sure only one of the reader threads access this vector at a particular instant.

Writer Function:

Each of the writer threads execute this function.
In this function the thread id is obtained by gettid() function.
A loop is run kw times and in each iteration a request for write is made and serviced.
The semaphore file is acquired and then the request time is logged into the output file.

After this the writer thread waits on queue semaphore .
After this the writer thread waits on the semaphore writelock. This is to gain access to the shared resource and enter the critical section.
The queue semaphore is signaled which allows the next thread in the queue to wake up.

In the critical section the entry time is logged onto the output file and the time difference between the entry and request is calculated.
For simulating a critical section, the threads sleeps for time **randCST**
This thread then signals writelock semaphore.

The exit time is logged into the output file and then remainder section executes.

To simulate Remainder section the thread is put to sleep for time randRemTime which is calculated same as the above randCST.

After the kw iterations the **average of these time differences** of the kw iterations of a reader thread is calculated and it is stored in a vector named **Writers.** As this Writers vector is a shared resource for all the writer threads another semaphore **time_lock** is used to make sure only one of the writer threads access this vector at a particular instant.

Joining the threads :

The writer threads are joined by using the function pthread_join() in a loop .
And similarly for the readers.
After all the threads are done executing, the average of waiting time for the reader threads and the writer threads is calculated by running for loops on Readers vector and Writers vector.In the similar way worst waiting times are calculated.
The average times are written into an output file.

Output :

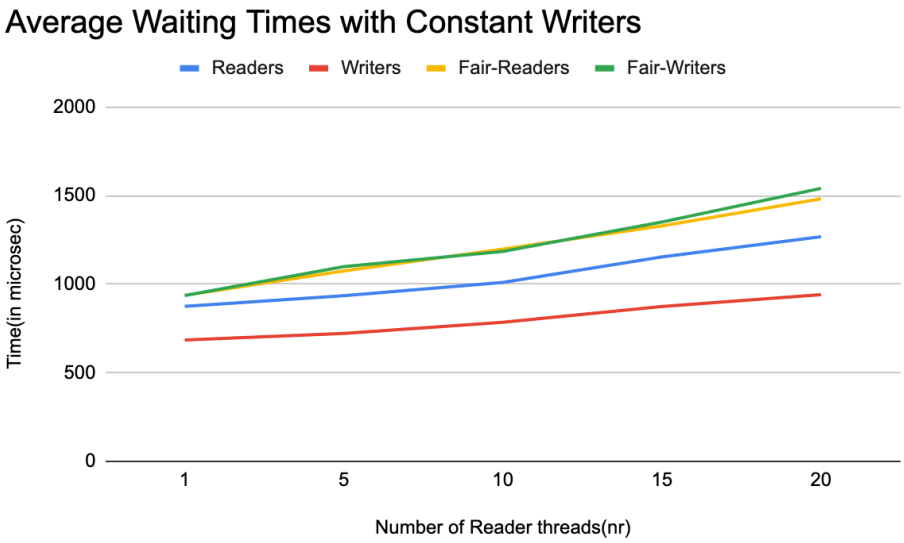
Two output files will be created .

1) “FairRW-log.txt” which contains the request time,entry time,exit time of all the iterations of the threads.

2) “Average-FairRW.txt” which contains the average waiting time of each Reader thread and of each Writer Thread and also average of all the reader threads and the writer threads.

Results

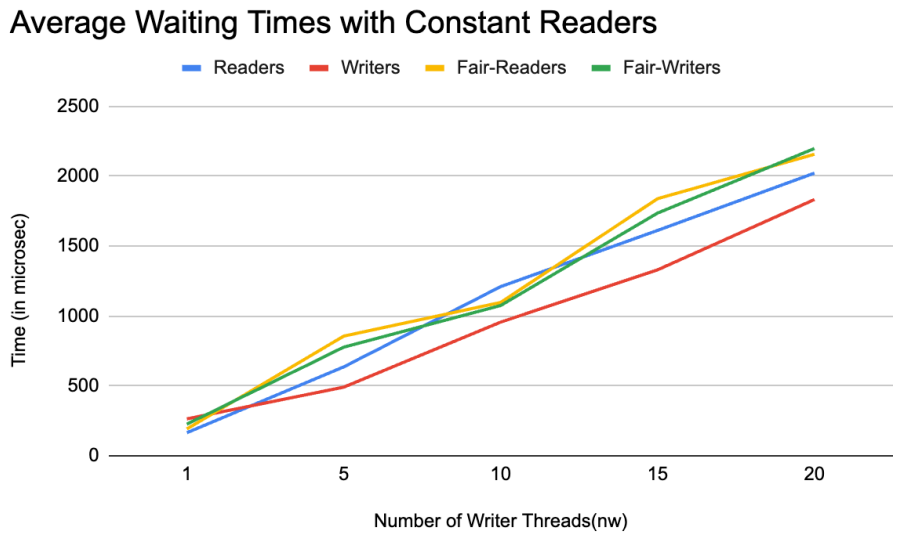
Experiment-1



In this experiment the value of nr is varied from 1 to 20 in increments of 5 keppind the other parameters constant as
nw=10
kr=10
kw=10
μCS=10
μRem=5

From the graph it can be seen that,
In the writers preference implementation the average waiting time of readers is greater than that of writes as writers are given more priority.
In the fair solution implementation both the readers and writers have almost same average time this is because they are given equal priority and starvation does not occur.
It can be observed that average waiting time of fair readers and fair writers is greater than readers in writer preference. One reason for this could be that each thread has to wait on the queue semaphore.

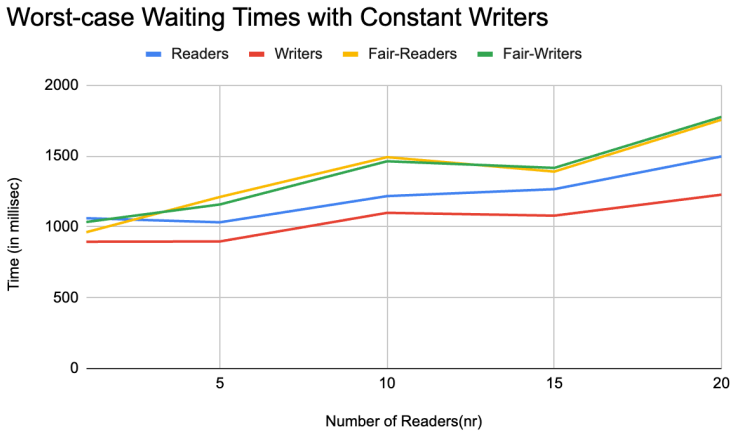
Experiment-2



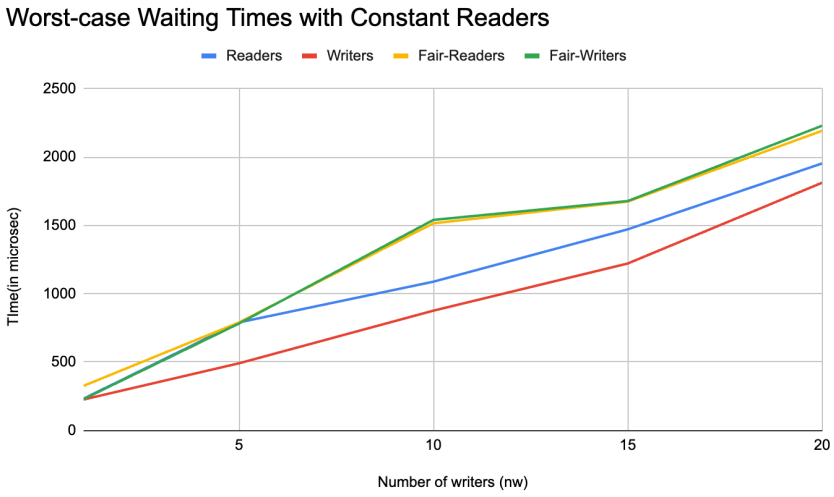
In this experiment the value of nw is varied from 1 to 20 in increments of 5 keppind the other parameters constant as
nr=10
kr=10
kw=10
μCS=10
μRem=5
From the graph it can be seen that,
In the writers preference implementation the average waiting time of readers is greater than that of writes as writers are given more priority.
In the fair solution implementation both the readers and writers have almost same average time this is because they are given equal priority and starvation does not occur.
The Four curves are pretty close to each other and intersect at some places.

Experiment-3
In this experiment the value of nr is varied from 1 to 20 in increments of 5 keppind the other parameters constant as
nw=10
kr=10
kw=10
μCS=10
μRem=5
Graph is plotted between worst case waiting time and number of readers.
The trend in this graph is similar to that observed in the case of experiment 1(Average waiting time with constant writers). The reasons for this trend are the same.

In the writers preference implementation the average waiting time of readers is greater than that of writes as writers are given more priority.
In the fair solution implementation both the readers and writers have almost same average time this is because they are given equal priority and starvation does not occur.



Experiment-4



In this experiment the value of nw is varied from 1 to 20 in increments of 5 keppind the other parameters constant as
nr=10
kr=10
kw=10
μCS=10
μRem=5
The trend observed in this experiment is similar to that observed in experiment 2
In the writers preference implementation the average waiting time of readers is greater than that of writes as writers are given more priority.

In the fair solution implementation both the readers and writers have almost same average time this is because they are given equal priority and starvation does not occur.

Complications

The time taken for computation changes from one execution to the other.This is because of the processor availability and time taken for content switch. Hence to find the time taken I have averaged over 5 executions and taken that time.