pseudocode with explanation of the Solution :

Diagram

Description automatically generated

**Definition of Readers and Writers problem :**

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time.

**NOTE :** Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.

**Overview explanation :**

* If two or more than two readers want to access the file at the same point in time there will be no problem.
* However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems.
* the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no reader is allowed to read the file at that point of time and if one writer is updating a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

**Possible Cases of the problem :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Case** | **Process 1** | **Process** | **Allowed / Not Allowed** |
| Case 1 | Writing | Writing | Not Allowed |
| Case 2 | Reading | Writing | Not Allowed |
| Case 3 | Writing | Reading | Not Allowed |
| Case 4 | Reading | Reading | Allowed |

**Solution for the problem :**

Using Semaphore algorithm

**Definition of semaphore algorithm :** Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

* **Wait**

The wait operation decrements the value of its argument S , if it is positive. If S is negative or zero, then no operation is performed.

* **Signal**

The signal operation increments the value of its argument S.

Advantages of Semaphores

* Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
* There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
* Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

* Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
* Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
* Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

**Examples when the deadlock can occur and how we solved it :**

Suppose that thread1 and thread2 are two threads that are in a deadlock. The thread thread1 holds the lock for the resource R1 and waits for resource R2 that is acquired by thread thread2.

At the same time, thread thread2 holds the lock for the resource R2 and waits for R1 resource that is acquired by thread thread1. But thread2 cannot release the lock for resource R2 until it gets hold of resource R1.

Since both threads are waiting for each other to unlock resources R1 and R2, therefore, these mutually exclusive conditions are called deadlock in Java.

Let’s understand the concepts of deadlock with realtime examples.

**How we solved it :**

// leave the lock to writer -and how allow to enter after Reader finish, so any deadlock problems are solved.

if (readCount == 0 ) {

Writer.writeLock.release();

}

readLock.release();

// tack the lock Writer and prevent form enter with Reader, so any deadlock problems are solved.

if (readCount == 1 ) {

Writer.writeLock.acquire(); }

readLock.release();

**Examples when Starvation can occur and how we solved it :**

When two threads are having different priority then the thread having higher priority will get the first chance to execute. Low priority thread has to wait until completing all high priority threads. It may have to wait for a long time period for its execution but waiting will end at a certain point. This situation is an example of starvation.

**Another example :**

Assume there are 10,000 threads running. Among them, one thread is having priority=1 (lowest priority), and the remaining all threads are having a priority greater than 1 (priority > 1). In this case, the thread having priority = 1 has to wait till the completion of all remaining threads having high priority

**How we solved it :**

Public class Reader implements Runnable {

// the threads are placed into a FIFO queue when blocked , so any starvation problems are solved .

Public static semaphore readLock = new semaphore (1 ,true) ;

Public volatile static int readCount = 0 ;

// every thread will read/write the value in main memory only. (atomic)

}

**Explanation for real world application and how did apply the problem :**

* We Have Implemented a demo about airport system which allow the users to (book/ return) any no. of tickets they want according to the no. of tickets available for their flight.
* Writer process explanation : We allow the user to write the no. of tickets he wants to (book/return) for his flight.
* Reader process explanation : we allow the user to show the no. of available tickets to facilitate the booking transaction.

**NOTES :**

Any change the user will made (book/return) no. of tickets, our system will be updated simultaneously to show the right no. of available tickets after the changing process.