**Solution pseudocode:**

**Definition of readers and writers problem :**

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

Solution: allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

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

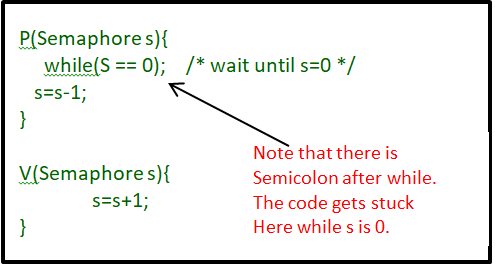
Here priority means, no reader should wait if the share is currently opened for reading Three variables are used: mutex,rw\_mutex , read\_count to implement solution

semaphore mutex, ,rw\_mutex; // semaphore mutex is used to ensure mutual exclusion when read\_count is updated i.e. when any reader enters or exit from the critical section and semaphore ,rw\_mutex is used by both readers and writers

**Functions for semaphore :**

– wait( ) : decrements the semaphore value.

– signal( ) : increments the semaphore value.



**Writer Process:**

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

**Pseudocode :**

do {

// writer requests for critical section

wait(rw\_mutex);

// performs the write

// leaves the critical section

signal(rw\_mutex);

} while(true);

**Reader process:**

1. Reader requests the entry to critical section.
2. If allowed:

* it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the rw\_mutex semaphore to restrict the entry of writers if any reader is inside.
* It then, signals mutex as any other reader is allowed to enter while others are already reading.
* After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “rw\_mutex” as now, writer can enter the critical section.

1. If not allowed, it keeps on waiting.

do {

// Reader wants to enter the critical section

wait(mutex);

// The number of readers has now increased by 1

readcnt++;

// there is atleast one reader in the critical section

// this ensure no writer can enter if there is even one reader

// thus we give preference to readers here

if (read\_count==1)

wait(rw\_mutex);

// other readers can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex); // a reader wants to leave

read\_count--;

// that is, no reader is left in the critical section,

if (read\_count == 0)

signal(rw\_mutex); // writers can enter

signal(mutex); // reader leaves­­­­­

} while(true);

Thus, the semaphore ‘rw\_mutex ‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

-------------------------------------

-------------------------------------

**Examples of Deadlock and How did solve deadlock :**

**Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.**

**Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.**



**How did solve deadlock :**

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

if (readCount == 0 ) {

Write.rw\_mutex.release();

}

mutex.release();

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

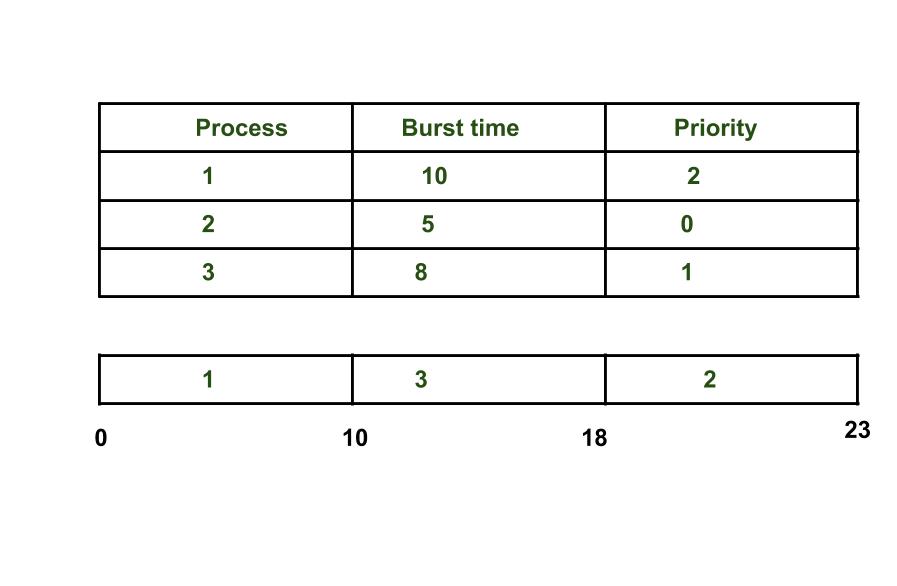
if (readCount == 1 ) {

Write.rw\_mutex.release(); }

mutex.release();

**Examples of starvation and How did solve starvation:**

**Starvation** or indefinite blocking is a phenomenon associated with the Priority scheduling algorithms. A process that is present in the ready state and has low priority keeps waiting for the CPU allocation because some other process with higher priority comes with due respect time. Higher-priority processes can prevent a low-priority process from getting the CPU.



As we see in the above example process having higher priority than other processes getting CPU earlier. We can think of a scenario in which only one process is having very low-priority and we are giving other process with high-priority, this can lead indefinitely waiting for the process for CPU which is having low-priority, this leads to Starvation.

When many 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

**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 mutex = new semaphore (1 ) ;

Public volatile static int readCount = 0 ;

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

}

**Differences between Starvation and Deadlock**

|  |  |
| --- | --- |
| Starvation | Deadlock |
| 1. It is called *lived stock*. | 1. It is called circular waiting. |
| 2. Starvation occurs when a process requires a resource for execution that it is never allowed. | 2. A deadlock occurs when two or more processes need some resource to complete the execution held by the other process. |
| 3. Starvation is more of a scheduler issue. | 3. Deadlock is more of a process design/distributed design issue |

**Real-world Examples:**

A banking system that read account balances versus update

Writer process

we allow the user to Update account balances for users

Reader process

we allow the user to read account balances

----------------------------------------------------------------------

Another example

A sheet with grades for students in a subject

Writer process

we allow the DR \ TA to Update grades for student

Reader process

we allow the DR \ TA and student to read grades

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Full Name** | **المستوى الدراسي** | **Department** |
| **202000555** | **عبدالله نبيل ابراهيم عيد** | **Level 3** | **IS** |
| **202000863** | **مروان احمد عبدالتواب جاب الله** | **Level 3** | **IS** |
| **202000199** | **بسمله سامح عبدالله عبدالغني** | **Level 3** | **IS** |
| **202000964** | **مي ممدوح قناوي حسين** | **Level 3** | **IS** |
| **202000345** | **رؤى محمود عبدالجواد محمود** | **Level 3** | **IS** |
| **202000363** | **زينب حسام الدين مصطفى محمد** | **Level 3** | **IS** |
| **202000889** | **مصطفي احمد حسين ابراهيم الخشاب** | **Level 3** | **IS** |
| **202001097** | **يوسف عماد الدين علي سلام** | **Level 3** | **IS** |