**Operating system 2 Project – Cover sheet**

Project Title :

Readers-Writers Problem

Group# ……………………………………………………..

Discussion time:- …………………………………………………….. Instructor ……………………………………………………

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Name(Arabic) | Bounce | Minus | Total Grade | Comment |
| 202000092 | احمد هشام احمد حلمي عبدالسلام |  |  |  |  |
| 202000522 | عبدالرحمن علي مصطفى كمال |  |  |  |  |
| 202000620 | عمرو يوسف محمد |  |  |  |  |
| 202001028 | هبه اسماعيل محمد علي |  |  |  |  |
| 202000487 | عائشه محمد احمد عطيه |  |  |  |  |
| 202001064 | يسرا رمضان عبدالوهاب جاد |  |  |  |  |
| 202000677 | كريمه ابوحسيبه محارب عبدالحفيظ |  |  |  |  |
|  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Critrial |  | | | | | | | | Grade | | | Team Grade | Comment |
| Documentation | Solution pseudocode | | | | | | | |  | 1 | |  |  |
| Examples of Deadlock | | | | | | | |  | 1 | |  |  |
| How did solve deadlock | | | | | | | |  | 1 | |  |  |
| Examples of starvation | | | | | | | |  | 1 | |  |  |
| How did solve starvation | | | | | | | |  | 1 | |  |  |
|  | Explanation for real world application and how did apply the problem | | | | | | | |  | 1 | |  |  |
| GitHub | Upload project files | | | | | | | |  | 2 | |  |  |
| Submitted before discussion time (shared GitHub project link with TA and Dr) | | | | | | | |  | 1 | |  |  |
| Only one contribution | | | | | | | | -1 | | |  |  |
| Implementation | Run correctly (correct output) | | | | | | | |  | 5 | |  |  |
| Run but with incorrect output | | | | | | | | -3 | | |  |  |
| Not run at all (error and exceptions) | | | | | | | | -8 | | |  |  |
| Free from Deadlock | | | | | | | |  | 3 | |  |  |
| Free from deadlock in some cases and not free in other cases | | | | | | | | -2 | | |  |  |
| Free from Starvation | | | | | | | |  | 2 | |  |  |
| Free from Starvation in some cases and not free in other cases | | | | | | | | -1 | | |  |  |
| Apply problem to real world application | | | | | | | |  | 6 | |  |  |
| Total |  | Total grade for Team | | | | |  | |  | 25 |  |  |  |
|  | Total Team Grade(after adjustment) | | | | | |  |  | 25 |  |  |  |
| Bounce | Multithreading GUI Based Java Swing | | | | | | | | +5 | | |  |  |
| Multithreading GUI Based Java | | | | | | | |  |  |
| Swing( | | adjustment | | ) | | | |
| Multithreading GUI Based JavaFX | | | | | | | | +10 | | |  |  |
| Multithreading GUI Based | | | | | | | |  |  |
| JavaFX( | | | adjustment | | ) | | |
| Bounce Graphic and animation | | | | | | | | +5 | | |  |  |
| Total with  Bounce |  | Total Team Grade | | |  | | | |  | | |  |  |
|  | Total Team Grade(after adjustment) | | | | | |  |  | | |  |  |

Index of the content

[Readers and Writers Problem: 2](#_Toc121817321)

[**1)** **Solution when Reader has the Priority over Writer pseudocode** 3](#_Toc121817322)

[**Writer process:** 3](#_Toc121817323)

[**Reader process:** 4](#_Toc121817324)

[**Deadlock:** 5](#_Toc121817325)

[**2)Examples of Deadlock** 5](#_Toc121817326)

[**3)Methods for handling deadlock** 6](#_Toc121817327)

[**Starvation** 7](#_Toc121817328)

[**4)Examples of starvation** 7](#_Toc121817329)

[**5)** **How did solve starvation** 7](#_Toc121817330)

[**6) Real-word Example :** 11](#_Toc121817331)

Readers and Writers Problem:

Consider a situation where we have a file shared between many people. 

* If one of the person tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
* However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem,** Problem parameters: 

* One set of data is shared among a number of processes
* Once a writer is ready, it performs its write. Only one writer may write at a time
* If a process is writing, no other process can read it
* If at least one reader is reading, no other process can write
* Readers may not write and only read

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution 

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt;  //    **readcnt** tells the number of processes performing read in the critical section, initially 0

**Functions for semaphore :**

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

1. **Solution when Reader has the Priority over Writer pseudocode**

**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.



**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 **wrt** 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 “wrt” as now, writer can enter the critical section.
3. Graphical user interface, text, application

   Description automatically generatedIf not allowed, it keeps on waiting.

**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.

**2)Examples of Deadlock**

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).

Diagram

Description automatically generatedFor 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.

**Deadlock can arise if**the **following four conditions hold simultaneously (Necessary Conditions)**  
***Mutual Exclusion:*** Two or more resources are non-shareable (Only one process can use at a time)   
***Hold and Wait:***A process is holding at least one resource and waiting for resources.   
***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.   
***Circular Wait:*** A set of processes are waiting for each other in circular form.

**3)Methods for handling deadlock**

There are three ways to handle deadlock   
1) Deadlock **prevention** or **avoidance**: The idea is to not let the system into a deadlock state.   
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.   
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use **Banker’s algorithm** in order to avoid deadlock.

2) Deadlock **detection and recovery**: Let deadlock occur, then do preemption to handle it once occurred.

3) **Ignore the problem altogether**: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

**BUT IN OUR WREADER WRITER PROBLEM DEADLOCK PROBLEM WILL NEVER HAPPEN BECOUSE OUR PROBLEM DOSE NOT SATISFY ANY CONDITION OF THE FOUR CONDITIONS**

**Mutual Exclusion: our problem has one shared resource, and this is enough to prevent the deadlock problem from occurring**

**Starvation**

**4)Examples of starvation**

The readers-writers problem has several variations, all involving priorities.

* The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. (Priority for readers)

In this case, **writers may starve**.

* The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading. (Priority for writers)

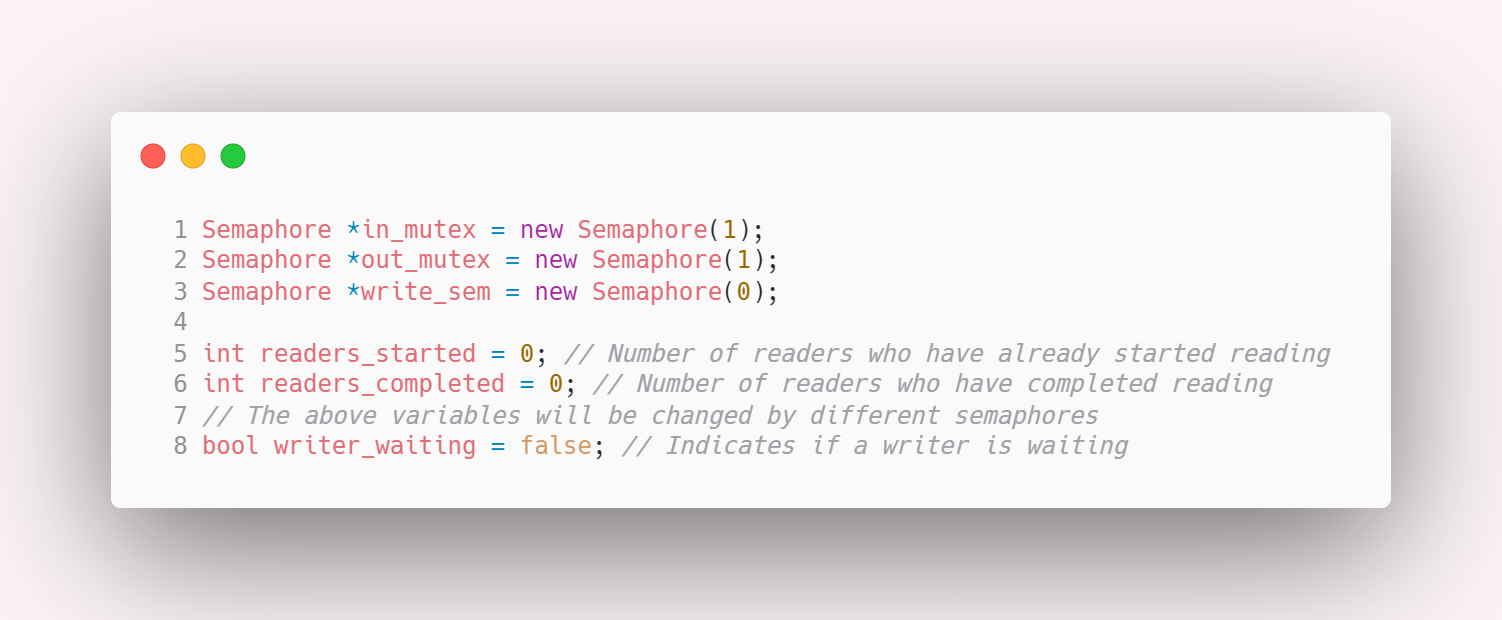
In this case, **readers may starve.**

**5)** **How did solve starvation**

Optimized Solution (Starve Free)

**Logic**

This solution with an optimization in the implementation of reader. Here too, the queue of in\_mutex serves as a common waiting queue for the readers and the writers.

**Global Variables** Global variables shared across all the processes.

##### **Logic for the readers**

In the previous solution, we had to enclose the entry part inside two mutex locks. But here, only one lock is sufficient.

Hence, the process need not be blocked twice.

This saves a great deal of time as blocking a process causes a lot of additional temporal overhead.

Here, initially we have the in\_mutex.

Again, all the readers and writers have to queue in this mutex to ensure equal priority.

Once a reader acquires the in\_mutex (after a writer completes its execution or after a fellow reader signals the mutex), it shows its presence by increasing the variable readers\_started and then immediately signals the in\_mutex.

The only thing that can keep a reader waiting, in this algorithm, is the wait for in\_mutex.

Hence, the reader directly proceeds to its critical section.

Note that all the readers can read at the same time as only writers have a critical section in between the wait() and signal() methods of in\_mutex. After the reader executes its critical section, the reader has to demark that it has completed its critical section execution and does not need the resource anymore.

So, it waits for the out\_mutex and once it acquires it, it increments the variable readers\_completed, thus, announcing its completion of resource usage.

Further, it goes to check if any writer is waiting by checking the variable writer\_waiting.

If yes, it checks if any fellow readers are executing in their critical sections. If not, it signals the writer to start its execution by calling signal() on the semaphore write\_sem.

After this, it signals the out\_mutex to release the variable readers\_completed for other readers and continues to its remainder section.

Text

Description automatically generated**Implementation: Reader**

##### **Logic for the writers**

Firstly, the writers wait on the in\_mutex with all the readers. After acquiring the in\_mutex, the writer goes on to wait on the out\_mutex. Now, after acquiring the out\_mutex (which is just a means to introduce mutual exclusion for the variable readers\_completed), it compares the variables readers\_started and readers\_completed. If they are equal, it means all the readers that had started their reading have completed it. That is, no reader is executing in its critical section currently. If that is the case, the writer simply signals the out\_mutex, thus, releasing its control over the variable readers\_completed and continues with its critical section. Note that any other reader or writer cannot execute in their critical sections as in\_mutex is not signalled yet. If the variables readers\_started and readers\_completed are not equal i.e. there is are reader processes executing their critical sections, then, the writer changes the variable writer\_waiting to true to state its presence and then, signals the out\_mutex. Also, since the resource is busy, the writer waits in writer\_sem for all the readers to complete their execution. Once it acquires writer\_sem, it changes the variable writer\_waiting back to false and proceeds to its critical section. Once the writer completes the critical section, it signals the in\_mutex to state that it does not need the resource anymore. Now, the process next in queue of in\_mutex can proceed.

#### Text Description automatically generatedImplementation: Writer

**6) Real-word Example :**