**Operating system 2 Project – Cover sheet**

Project Title :

**Readers-Writers Problem**

Group# ……………………………………………………..

Discussion time:- … 8:50:00 AM ….. Instructor …… Abdelrahman ……………………

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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: 3](#_Toc121817321)

[**1)** **Solution when Reader has the Priority over Writer pseudocode** 4](#_Toc121817322)

[**Writer process:** 4](#_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** 8](#_Toc121817329)

[**5)** **How did solve starvation** 9](#_Toc121817330)

[**6) Real-word Example :** 14](#_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 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**

\*Deadlock will happen if:

* Graphical user interface, text, application, email

  Description automatically generatedThe programmer wrote this case in his code

And reader process entered first, so a reader will enter and will find nothing to read; so it will wait in the critical section, and the writer wanted to enter but cannot enter because there is a reader in the critical section.

So the writer will wait for the reader to exit, and the reader

will wait to find anything to read hence deadlock will occur.

* To solve this case we will write the program as follows

Graphical user interface, text, application

Description automatically generated with medium confidence

* We have shared **resources** when many threads of execution try to access the same shared **resources**at a time. There are N readers to read data and K Writers to write data to shared **resources**.
* To solve this case we will use resource ordering techniques to avoid deadlock

**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 (Free from Starvation and Deadlock)**

* **Conroller Class Pseudocode:**

Variables are used: b, writing, waitingWriters, readers, readersTurn

We have 6 methods: startWriting(), write(String s), stopWriting(), startReading(), stopReading(), getBalance()

* **startWriting() Pseudocode:**

WHILE writing is true OR readers > 0

INCREMENT waitingWriters

BEGIN

CALL wait()

EXCEPTION

WHEN InterruptedException ex

DECREMENT waitingWriters

END

ENDWHILE

DECREMENT waitingWriters

SET waiting true

END

* **write(String s) Pseudocode:**

APPEND b by 100

END

* **stopWriting() Pseudocode:**

SET writing false

SET readersTurn true

CALL notifyAll()

END

* **startReading() Pseudocode:**

WHILE writing is true OR waitingWriters > 0 AND NOT(readerTurn)

BEGIN

CALL wait()

EXCEPTION

WHEN InterruptedException ex

END

ENDWHILE

INCREMENT readers

END

* **stopReading() Pseudocode:**

DECREMENT readers

SET readersTurn false

IF readers is 0 THEN

CALL notifyAll()

ENDIF

END

* **getBalance() Pseudocode:**

RETURN b

END

* **Controller Class** Text

  Description automatically generated with medium confidence
* Text

  Description automatically generated**Reader Class**
* Text

  Description automatically generated**Writer Class**

**6) Real-word Example :**

#### “Bank System”

#### Let's imagine that you are a very dedicated and serious investor. You invested in many places and they all add your ROI (return on investment) at the end of every month at midnight, each as their own thread. In the case that two writer threads attempt to update the balance at the exact time, there is a chance that both of them add to the exact same balance instead of adding and then updating, for example: (Current balance is $200 and P1 and P2 add $100 to the $200 at the same time, resulting to the final balance being $300 instead of $400)

#### But what it will happen is that P1 will call startWrite() method and then enter the critical section to adding 200 to original balance (100), then P2 will call the same method but this time cannot enter the critical section to add 200 to balance, it will be waiting P1 to exit then it will enter and will add 200 to 200+100=300, so the final true balance is $500

#### Our code ensures that:

#### NO data inconsistency which appears if there Race-Condition problem

#### NO starvation will occur

#### NO deadlock will occur