Operating system 2 Project – Cover sheet

Project Title :

Readers-Writers Problem

Group# …………………………………………………..

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

ID	Name(Arabic)	Bounce	Minus	Total Grade	Comment
202000031	احمد حمدى محمود الشويمى				
202000145	السيد محمد السيد محمود				
202000226	تغريد ربيع علي محمد				
202000357	زياد محمد احمد علي				
202000228	تقوى السيد محمد فهمى				
202000053	احمد علاء محمد المهدي				
202000335	روان محمد الطيب محمد				
202000348	ريم رضا عبدالحميد				

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		
	Run correctly (correct output)	5		
	Run but with incorrect output	-3		
	Not run at all (error and exceptions)	-8		
	Free from Deadlock	3		

Implementation	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	[illegible]		
	Total Team Grade(after adjustment)	[illegible]		
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

The Problem Statement

A database must be shared by numerous concurrent processes, some of which may simply want to read the database, while others may wish to update (read and write) the database.
We differentiate between these two processes by referring to the former as Readers and the latter as Writers.

- There will be no negative consequences if two readers access the shared data simultaneously.
- If a writer and another thread (either a reader or a writer) access the common data simultaneously, chaos may follow.

To avoid these problems, we need that the writers have exclusive access to the shared database.
This synchronization issue is known as the readers-writers problem.

Parameters of the problem:

-
- Several processes share a single set of data.
- When a writer is ready, it begins writing.
- At any given moment, only one writer may work.
- No other process can read what a process is writing.
- No other process can write if at least one reader is reading.
- Readers may not write and must rely only on what they read.

Reader Writer Solution pseudocode

The Solution:
We will make use of two semaphores and an integer variable:
1. **mutex,** a semaphore (initialized to 1), is used to ensure mutual exclusion when readCount is updated, i.e., when any reader enters or exits from the critical section.
2. **wrt**, a semaphore (initialized to 1) common to both reader and writer processes.
3. **readCount**, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

Functions for semaphore :

- wait(): decrements the value of the semaphore.
- signal(): increments the value of the semaphore.

pseudocode

```
Initialize b to 100
Initialize readersTurn to false
Function startwrite()
```

```
Reapet until the value of writing or readers is greater than 0{
Increase waitingwriters by 1
 Try: {
    (wait() )
 catch( (Argument  InterruptedException ex):
    ( decrease  waitingWriters by 1)
 }
 decrease  waitingWriters by 1

  initailze writing to true
 }
  function write(Argument s){

( decrease  waitingWriters by 1    }

function stopWriting(){
      initialize writing to false
      initialize readersTurn to true;
      (function  notifyAll())
   }
Function startReading() {
 Repeat until writing or waitingWriters is greater than 0 and not readersTurn{
            Try:
            (   function wait() )
           catch( Argument InterruptedException ex): }
Increase readers by 1
Function  stopReading(){
decrease  readers by 1
initialize    readersTurn to false;
if readers equal to 0 then (function notifyAll())
}
  function getTickets()    {    return b}
```

Reader process

1. The reader requests entry to the critical section.
2. If permitted,

- it increments the number of readers within the critical section. If this reader is the first to enter, the wrt semaphore is locked, preventing writers from entering if any other reader is present.
- it then signals mutex, indicating that any new reader may enter while others are currently reading.
- it leaves the critical section after reading. When departing, it checks to see whether there are any more readers within and if there are, it signals the semaphore "wrt," indicating that the writer can now enter the critical region.

3. If it is not permitted, it will continue to wait.

```
do {
  // The reader requests entry to the critical section
  wait(mutex);
  //Now,the number of readers has incremented by 1
  readCount++;
  // there is minimum one reader in the critical section
  // this ensures that no writer can enter if there is even one reader
  // hence readers are given preference here
  if (readCount==1)
  wait(wrt);
  // other readers can enter while the current reader is inside the critical section
  signal(mutex);
  // current reader performs reading
  wait(mutex);   // a reader wants to exit
  readCount--;
  // i.e., no reader is left in the critical section,
  if (readCount == 0)
   signal(wrt);        // writers can enter now
  signal(mutex); // reader exits
} while(true);
```

As a result, the semaphore 'wrt' is queued on both readers and writers, with readers being prioritized if writers are also present.

Writer process

1. The writer requests entry to the critical section.
2. If permitted, i.e., wait() returns a true value, it enters and performs the write.

3. If it is not allowed, it will continue to wait.
4. It gets out of the critical section.

```
do {
   // the writer requests entry to the critical section
   wait(wrt);
   // performs the write
   // exits the critical section
   signal(wrt);
} while(true);
```
.

• Starvation

- A thread may wait indefinitely because other threads keep coming
- in and getting the requested resources before this thread does. Note that
- resource is being actively used and the thread will stop waiting if other
- threads stop coming in.

Example of starvation:

1. Reader Threads are given preference over Writer Threads. Hence, if lot of Readers are coming in compared to few Writers, Writer Threads will get starved.
2. Starvation of Writer Threads can result in Reader Threads reading old(stale) data.

1. Writer Threads are given preference over Reader Threads. Hence, lot of Readers get starved when there are lot of writes..
2. Starvation of Reader Threads can result in inconsistent reads.

Avoiding starvation:

Switch priorities so that every thread has a chance to have high priority. o E.g., Readers give priority to waiting writers, but active writers give priority to waiting readers. When both are waiting, they will end up alternating. o Raise priority if a thread has been waiting too long. o Use mutex order among competing requests

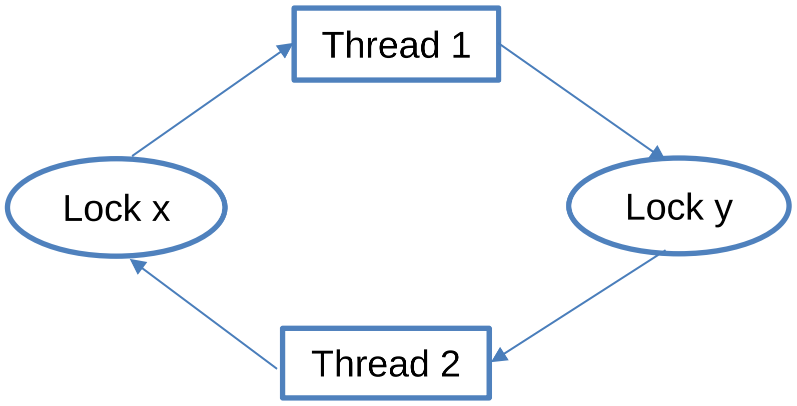
• Deadlocks

- A group of threads are waiting for resources held by others in the
    - **group. None of them will ever make progress.**

Preventing deadlocks:

To understand how to prevent deadlocks, one must figure out necessary conditions for a deadlock to occur. There are four necessary conditions, all of which must hold, for a deadlock to occur. If any one condition fails, then you cannot have a deadlock.

1) Limited (shared) resource
i.e., not enough to serve all threads simultaneously. Otherwise there's no waiting.
2) **No preemption:** Preemption means you forcefully take away the resource from someone. (Hard to do that with locks.)
3) **Hold and wait:** Threads in a deadlock hold a resource, and wait for another resource
4) Circular wait:



- thread 1 is waiting for resource y
- resource y is held by thread B
- thread 2 is waiting for resource x
- resource x is held by thread A

Real-word Example :

Readers try to view tickets while Writers try to book a ticket, the Reader and Writer Threads in the code are made to read and book the tickets. controller class of Java is used to implement the solutions.

In reality, multiple readers and writers try to read and book tickets of all the flights and low level synchronization primitives
like Semaphores etc.. are not used. Instead the Application itself stores data in an RDBMS like Oracle
which helps in concurrent transactions automatically. The application needn't bother to write the locking mechanism .
Oracle never blocks reads. By a method of SCN and rollback/undo, it provides consistent Reads at any point in time .
Multiple Writes are prevented using Row-Level L0w