Operating system 2 Project – Cover sheet

Project Title :

Readers-Writers Problem

Group# …………………………………………………..

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

ID	Name(Arabic)	Bounce	Minus	Total Grade	Comment
202000713	مايكل صفوت				
202000698	ماريان معوض عزيز				
202000705	مازن السيد كامل السيد				
202000689	مارفيلين يوسف انور				
202000489	عبدالعزيز حسن عبدالعزيز				
202000703	ماريو عماد حكيم				
202000169	انطوانيت عماد صالح				
202000797	محمد علاء الدين حسين				

Critirial		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:

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.

```
do {
    // writer requests for critical section
    wait(wrt);

    // performs the write

    // leaves the critical section
    signal(wrt);

} 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 **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. 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 (readcnt==1)
      wait(wrt);

   // 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

   readcnt--;

   // that is, no reader is left in the critical section,
   if (readcnt == 0)
       signal(wrt);         // writers can enter

   signal(mutex); // reader leaves

} while(true);
```

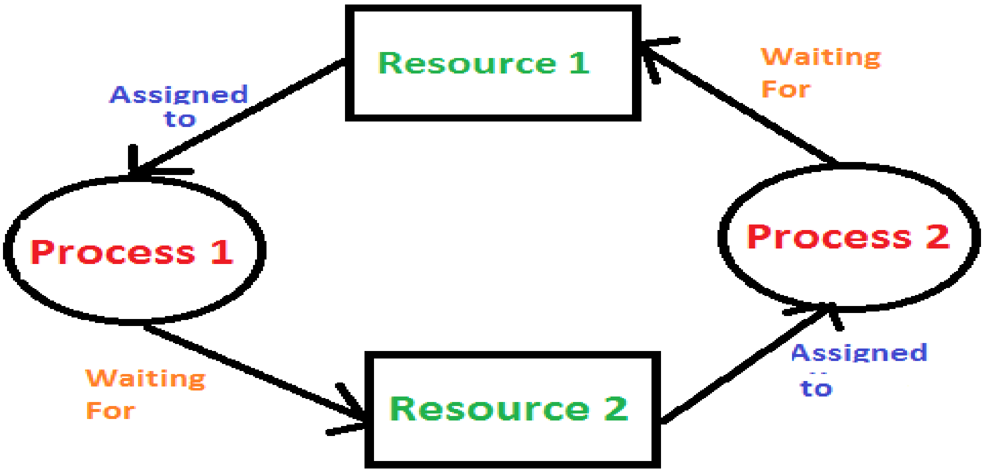
Deadlock:

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).

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.



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.

```
Semaphore *in_mutex = new Semaphore(1);
Semaphore *out_mutex = new Semaphore(1);
Semaphore *write_sem = new Semaphore(0);

int readers_started = 0; // Number of readers who have already started reading
int readers_completed = 0; // Number of readers who have completed reading
// The above variables will be changed by different semaphores
bool writer_waiting = false; // Indicates if a writer is waiting
```

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.

Implementation: Reader

```
do{
    // Entry Section
    in_mutex->wait(processID);
    readers_started++;
    in_mutex->signal();

    /**
     *
     * Critical Section
     *
     */

    // Exit Section
    out_mutex->wait(processID);
    readers_completed++;
    if(writer_waiting && readers_started == readers_completed){
        write_sem->signal();
    }
    out_mutex->signal();

    // Remainder section

}while(true);
```

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.

Implementation: Writer

```
do{
    // Entry Section
    in_mutex->wait(processID);
    out_mutex->wait(processID);
    if(readers_started == readers_completed){
        out_mutex->signal();
    }else{
        writer_waiting = true;
        out_mutex->signal();
        writer_sem->wait();
        writer_waiting = false;
    }
    /**
     *
     * Critical Section
     *
     */

   // Exit Section
   in_mutex->signal();

   // Remainder Section

}while(true);
```

<u>6) Real-word Example :</u>