#project 4

Readers and writers problem

1 - Solution pseudocode

We need to solve the problem with 3 classes free from Starvation and deadlock

First class : Global variables shared with other 2 clsses

1 – semaphore mutex

2- semaphore rw\_mutex

3- semaphore in\_mutex

4- initialize count = 0

Second Class Reader

1. Do{
2. Entry section
3. In\_mutex.wait(pid)
4. Mutex.wait(pid)
5. Counter ++
6. If (counter == 1):
7. Rw.mutex.wait(pid)
8. Mutex.signal()
9. In\_mutex.signal()
10. critical section
11. Mutex.wait(pid)
12. Counter --
13. If(counter==0):
14. Rw\_mutex.signal()
15. Mutex.signal()
16. Reminder section
17. } while(true)

Third class Writer

1. Do {
2. Entry section
3. In\_mutex.wait(pid)
4. Rw\_mutex.wait(pid)
5. Critical section
6. Rw\_mutex.signal()
7. In\_mutex.signal()
8. Reminder section
9. } while(true)

After implement these classes the problem is solved

2- Examples of Deadlock

Deadlock example for readers and writer problem

Boolean empty=true;

Public synchronized string read(){

While(empty);

Empty=true;

Return message;

}

Public synchronized void write(string message){

While(!empty);

This.message =message;

Empty=false;

}

\*When we started the threads from the Main class, both the threads called

the run() method.

Note that both threads are sharing a common message object. Now the Reader

thread called the synchronized read() method and hence acquired the lock of the

message object.

As initially the boolean empty flag was set to true, the Reader thread keeps

executing while loop infinitely.

Also, the Writer thread won't be able to execute the write() method as the lock of

the message object is already acquired by the Reader thread.

3-How did solve deadlock

Solving deadlock example

class Message{

String message;

boolean empty = true;

public synchronized String read() {

while (empty) {

try {

wait();

} catch (InterruptedException e) {

System.out.println(Thread.currentThread().getName() + "Interrupted.");

}

}

empty = true;

notifyAll();

return message;

}

public synchronized void write(String message) {

while (!empty) {

try {

wait();

} catch (InterruptedException e) {

System.out.println(Thread.currentThread().getName() + "Interrupted.");

}

}

this.message = message;

empty = false;

notifyAll();

}}

Reader thread waits until Writer invokes the notify()method or

the notifyAll() method for 'message' object. Reader thread

releases ownership of lock and waits until Writer thread notifies

Reader thread waiting on this object's lock to wake up either

through a call to the notify method or the notifyAll method.

Wakes up all threads that are waiting on 'message' object's monitor(lock).

This thread(Reader) releases the lock for 'message' object.

Writer thread waits until Reader invokes the notifyAll() method for

'message' object. Writer thread releases ownership of lock and

waits until Reader thread notifies Writer thread waiting on his

object's lock to wake up

Wakes up all threads that are waiting on 'message' object's

4,5-Examples of starvation and solution

he Reader-Writer's problem deals with synchronizing multiple processes which are categorized into 2 types namely:

* **Readers -** They read data from a shared memory location
* **Writers -** They write data to the shared memory location

Classical solution (Writers Starve)

This is the solution to the first reader-writer's problem. Here, there is a chance that the writers starve. The global variables (that are shared across all the readers and writers) are as shown below.

Global Variables

Semaphore \*mutex = new Semaphore(1);

Semaphore \*rw\_mutex = new Semaphore(1);

int counter = 0;

There are 2 mutex locks implemented using Semaphores namely mutex and rw\_mutex. mutex ensures the mutual exclusion of readers while accessing the variable counter and rw\_mutex ensures that all the writers get access to the shared memory resource exclusively. The implementation of the reader is shown below

Implementation: Reader

do{

// Entry Section

mutex->wait(processID);

counter++;

if(counter == 1) rw\_mutex->wait(processID);

mutex->signal();

/\*\*

\*

\* Critical Section

\*

\*/

// Exit Section

mutex->wait(processID);

counter--;

if(counter == 0) rw\_mutex->signal();

mutex->signal();

// Remainder Section

}while(true);

Here, if a reader is waiting for a writer process to signal the rw\_mutex, all the other readers are waiting on mutex. After the writer process signals the mutex, all the readers can simultaneously perform the read operations. Till all the readers are done, all the writers are paused on rw\_mutex, thus, causing starvation.

Implementation: Writer

do{

// Entry Section

rw\_mutex->wait(processID);

/\*\*

\*

\* Critical Section

\*

\*/

// Exit Section

rw\_mutex->signal();

// Remainder Section

}while(true);

Commonly Used Solution (Starve Free)

In the above solution, writers were starving as there was nothing stopping the readers entering continuously and blocking the resource for the writers. In this solution, we introduce another mutex lock implemented using a semaphore in\_mutex. The process having access to this mutex lock can enter the workflow described in the solution above and thus, have access to the resource. This implements a check to the readers that come after the writers as all the processes are pushed into the FIFO queue of the semaphore in\_mutex. Thus, this algorithm is starve-free.

Global Variables

Semaphore \*mutex = new Semaphore(1);

Semaphore \*rw\_mutex = new Semaphore(1);

Semaphore \*in\_mutex = new Semaphore(1);

int counter = 0;

The rest of the variables are same as the first solution with an addition of in\_mutex. Now, both the reader and the writer implementations have to enclosed within in\_mutex to ensure mutual exclusion in the whole process and thus, making the algorithm starve-free.

Implementation: Reader

do{

// Entry Section

in\_mutex->wait(processID);

mutex->wait(processID);

counter++;

if(counter == 1) rw\_mutex->wait(processID);

mutex->signal();

in\_mutex->signal();

/\*\*

\*

\* Critical Section

\*

\*/

// Exit Section

mutex->wait(processID);

counter--;

if(counter == 0) rw\_mutex->signal();

mutex->signal();

// Remainder Section

}while(true);

Initially, wait() function is called for in\_mutex. If a reader is waiting for a writer process, the reader is queued in the FIFO queue of the in\_mutex (rather than mutex) with the fellow writers. Thus, in\_mutex acts as a medium which ensures that all the processes have the same priority irrespective of their type being reader or writer.

Implementation: Writer

do{

// Entry Section

in\_mutex->wait(processID);

rw\_mutex->wait(processID);

/\*\*

\*

\* Critical Section

\*

\*/

// Exit Section

rw\_mutex->signal();

in\_mutex->signal();

// Remainder Section

}while(true);

Like the readers, the writers are also queued in the FIFO queue of in\_mutex by calling wait() for in\_mutex.

Hence, all the processes requiring access to the resources can be scheduled in a FCFS manner. In this particular solution, we have to use 2 semaphores. An optimized solution is explained below.

Optimized Solution (Starve Free)

Global Variables

Global variables shared across all the processes. They are as follows

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

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);

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);

6- Explanation for real world application and how did apply the problem

**An explanation for Reader-writer problem :**

**⦁ We Have Implemented a demo about airport system which allow the users to (book/ return) any no. of tickets they want according to the no. of tickets available for their flight.**

**⦁ Writer process explanation : We allow the user to write the no. of tickets he wants to (book/return) for his flight.**

**⦁ Reader process explanation : we allow the user to show the no. of available tickets to facilitate the booking transaction.**

**NOTES :**

**Any change the user will made (book/return) no. of tickets, our system will be updated simultaneously to show the right no. of available tickets after the changing process.**

* **The Reader-Writer problem is useful for modeling processes which are competing for a limited shared resources**
* **The readers–writers problems are examples of a common computing problem in concurrency There are at least three variations of the problems, which deal with situations in which many concurrent threads of execution try to access the same shared resource at one time.**

**- Reader-Writer Real-World examples :**

**. an airline reservation system consisting of a huge data base with many processes that read and write the data**

**. Banking System that read account balances versus updates**