"OS-2 PROJECT (READERS-WRITERS PROBLEM)"

ID	NAME
202000825	محمد نجيب على عبد العزيز
202001103	يوسف محمد الحلي عبدالعال
202000821	محمد ممدوح عطية سيد
202000794	محمد علاء الدين محمد علي
202001065	يمنى عبد الناصر السيد
202000696	ماريا عادل فوزي
202000697	ماريا عماد عزيز

“OS-2 PROJECT (READERS-WRITERS PROBLEM)”

- ## Solution pseudocode:

This algorithm solves reader & writer problem

Writing process

```
Do {
    Writer requests for critical section
    If there aren’t any processes held the write semaphore
        Then writer’s request accepted
    Else
        prevent any process’s entering while there is a process is
        going on
    Performs the write process
    After finishing Leave the critical section
    Allowing another process to enter
} While ← true
```

Reading process

```
readCount initially equals 0
Do {
    Reader requests for critical section
    If there isn’t a process is modifying the readCount
        Then increment readCount by 1
    If readCount == 1
        Then writer process can’t hold writer semaphore
        //don’t allow writer process from entering the critical section
    Allow any reader process to enter by mutex semaphore
    Current reader process performs reading process
    Decrement readCount by 1    //reader process wants to leave
    If readCount == 0
        Allowing writer process If exists
    Reader process leaves
} while ← true
```

"OS-2 PROJECT (READERS-WRITERS PROBLEM)"

• Examples of deadlock:

• Shared Data:

```
Semaphore mutex = new Semaphore(1);

Semaphore rw_mutex = new Semaphore(1);

readCount = 0;
```

• The structure of a writer's process:

```
while (true) {
     wait(mutex);
     wait(rw_mutex); // any writers or readers?
                     ...
     /* writing is performed */
                     ...
     Signal(rw_mutex);
     signal(mutex); // enable others
}
```

• The structure of a reader's process:

```
do {
     wait(mutex);// ensure mutual exclusion
     read count++;
     if (read count== 1) // another reader
          wait(rw_mutex); // block writers
     signal(mutex); // release lock for other readers
               ...
     /* reading is performed */
               ...
     wait(mutex); // ensure mutual exclusion
     read count--; // reader done
     if (read count== 0)
          signal(rw_mutex); // enable writers
     signal(mutex); // release lock for other readers

} while (true);
```

When one Reader has acquired rw_mutex and try to acquire mutex

```
if (read count== 1)
     wait(rw_mutex);
signal(mutex);

READING SECTION

wait(mutex); --> HERE
```

While another Writter has acquired mutex and try to acquire rw_mutex deadlock happens

```
wait(mutex);
wait(rw_mutex);--> HERE
```

Deadlock Happens

• How to solve deadlock:

- One way to solve this problem is to reverse the acquire (wait) function of mutex & rw_mutex in writer’s process

```
wait(rw_mutex);
wait(mutex);

WRITING SECTION

signal(mutex);
Signal(rw_mutex);
```

- We can also use only rw_mutex in writer’s process

```
wait(rw_mutex);

WRITING SECTION

Signal(rw_mutex);
```

"OS-2 PROJECT (READERS-WRITERS PROBLEM)"

Examples of starvation:

• *Shared Data:*

```
Semaphore mutex = new Semaphore(1);
Semaphore rw_mutex = new Semaphore(1);
readCount = 0;
```

• *The structure of a writer's process:*

```
while (true) {
     wait(rw_mutex); // any writers or readers?
                    ...
     /* writing is performed */
                    ...
     signal(rw_mutex); // enable others
}
```

• *The structure of a reader's process:*

```
do {
     wait(mutex);// ensure mutual exclusion
     read count++;
     if (read count== 1) // another reader
          wait(rw_mutex); // block writers
     signal(mutex); // release lock for other readers
                ...
     /* reading is performed */
                ...
     wait(mutex); // ensure mutual exclusion
     read count--; // reader done
     if (read count== 0)
          signal(rw_mutex); // enable writers
     signal(mutex); // release lock for other readers
} while (true);
```

"OS-2 PROJECT (READERS-WRITERS PROBLEM)"

First variation:
No reader kept waiting unless the writer has permission to use a shared object (Writer will starve).

Second variation:
Once a writer is ready, it performs the write ASAP. In other words, if a writer is waiting to access the object, no new readers may start reading. (Reader will starve).

How to solve starvation:

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.

• *Shared Data:*

```
Semaphore mutex = new Semaphore(1);

Semaphore rw_mutex = new Semaphore(1);

Semaphore in_mutex = new Semaphore(1);

readCount = 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 must enclosed within in_mutex to ensure mutual exclusion in the whole process and thus, making the algorithm starve-free.

“OS-2 PROJECT (READERS-WRITERS PROBLEM)”

• *The structure of a Reader’s process:*

```
do {
     wait(in_mutex);
     wait(mutex);
     readCount++;
     if (read count== 1)
          wait(rw_mutex);
     signal(mutex);
     signal(in_mutex);
               ...
     /* reading is performed */
               ...
     wait(mutex);
     readCount--;
     if (read count == 0)
          signal(rw_mutex);
     signal(mutex);
} 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.

• *The structure of a writer’s process:*

```
while (true) {
     wait(in_mutex);
     wait(rw_mutex);
                     ...
     /* writing is performed */
                     ...
     signal(rw_mutex);
     Signal(in_mutex);
}
```

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

• Explanation for real world:

We made a banking system:

- ➢ The reader’s process is reading account balances.
- ➢ The writer’s process is updating account balances “withdraw or deposit money”.