Reading Assignment:
Starve Free Readers-Writers Solution

Submitted
Under the Operating System course (CSN-232)

By

Aishwarya
(19114005)
(CSE)

Readers-Writers Problem

This is a classical problem of process synchronization and mutual exclusion where a shared file is being accessed by multiple processes (amongst which some want to read it called as readers and others want to write to it called as writers) concurrently.

In order to avoid any type of inconsistency generation or race condition, we need to ensure following 2 conditions:

- While a writer is in critical section no other reader/ writer should be allowed to access the critical section
- When a reader is present in critical section no writer should be granted access to enter there, while allowing multiple readers to access the critical section at a time (as this won't cause any problem)

Solution:

This synchronization problem can be solved by using Semaphores (integer variables that aid in achieving mutual exclusion condition) . The classical solution provided for this problem namely reader's preference and writer's preference cause the starvation of writers and readers respectively. Hence, in this assignment, I made an attempt to provide a starve free solution where neither readers nor writers starve and are given opportunities as soon as they arrive given that they follow other conditions as well.

Starve free solution:

All readers and writers will be granted access to the resource in their order of arrival (FIFO order). If a writer arrives while readers are accessing the resource, it will wait until those readers free the resource, and then modify it as soon as the resource is released by the reader. New readers arriving in the meantime will have to wait.

Initialization:

- Readers_count : initialized as 0 // indicates number of readers currently accessing the critical section
- Data: initialized as 1 // can be initialized by any value , this is the value modified by writer during write process
- Order_mutex: Semaphore used for maintaining fifo order of read/write requests made for accessing critical section
- Access_mutex: Semaphore used for requesting exclusive access to modify reader_count variable
- Readers_mutex: Semaphore used for avoiding conflicting accesses to entry and exit section (aiding in achieving mutual exclusion condition that only one reader can enter/ exit the critical section at a time) as multiple readers can simultaneously access the resource/ critical section.

We want the first reader to get access to the resource to lock it so that no writer can access it at the same time. Similarly, when a reader is done with the resource, it needs to release the lock on the resource if there are no more readers currently accessing it.

Pseudo-code:

Reader's Code:

```
do
{
<ENTRY SECTION>
wait(order_mutex);              //check for the turn to get executed, puts the blocked process
                                in a fifo queue if it's not its turn currently

wait(readers_mutex);            // requesting access to modify readers count

Readers_count++;                //increment the count of readers accessing the critical section
if(readers_count==1)            // if it is the first reader then get the access to the reader for
wait(access_mutex);             accessing the critical section and block the writer from
                                accessing it

signal(order_mutex);             // release the semaphore to get the next reader/ writer to be
                                serviced

signal(readers_mutex);           // release access to modify readers count

<CRITICAL SECTION>

 <EXIT SECTION>
  wait(readers_mutex);          // again request access to readers count
  readers_count--;              //  decrement the count as a reader is leaving
   if(readers_count==0)         // if it is the last reader then release the access to the critical
      signal(access_mutex);     //section so that it can be used by any waiting writer

  signal(readers_mutex);         // release access to modify readers count
}
while(true);
```

Writer's Code:

```
do
{
<ENTRY SECTION>
  wait(order_mutex);            //check for the turn to get executed, puts the blocked process
                                in a fifo queue if it's not its turn currently
  wait(access_mutex);           // get the access to the writer for accessing the critical section
                                 and block the readers from accessing it
signal(order_mutex);             // release the semaphore to get the next reader/ writer to be
                                serviced

<CRITICAL SECTION>

<EXIT SECTION>
    signal(access_mutex);       //release the access to the critical section so that it can be
```

used by any waiting reader

}

while(true);

Correctness of Solution:

Mutual Exclusion:

The access_mutex semaphore ensure that only a single writer can access the critical section at any moment, aiding in mutual exclusion amongst writers. As, if any other writer try to enter then the value of access_mutex won't allow it to do so.

Besides, when the first reader enter in critical section it calls wait(access_mutex) that aids in mutual exclusion between readers and writers as at that moment it blocks the writer.

Bounded Wait:

Before accessing the critical section both reader and writer use order_mutex which checks for the turn of the request made. If currently the request can't be granted then it blocks that thread and puts it in the FIFO queue. This ensures that when a thread needs to be released,it is done in FIFO sequence so that all get an opportunity of being executed after a finite amount of time and no one starves for an indefinite long time. This meets the requirement for bounded wait conditions.

Progress Requirement:

The code is structured so that there won't be any chances of occurrence of deadlock. Both the reader and writer take a finite amount of time in critical section execution and during exit release the waiting semaphore so that they can release the blocked threads/ processes for re- cheking and entering in critical section. Thus, no process hinders the execution of the other when there is no reader/ writer in the critical section.

Below Image shows the output of the solution.c code's successful execution:

```
helloubuntu@helloubuntu-VirtualBox:~/Desktop$ g++ solution.c -lpthread
-o c_output && ./c_output && rm c_output
Data written by the writer1 is 5
Data read by the reader3 is 5
Data read by the reader3 is 5
Data written by the writer4 is 15
Data read by the reader5 is 15
Data read by the reader6 is 15
Data written by the writer8 is 35
Data written by the writer8 is 75
Data written by the writer2 is 155
Data written by the writer2 is 315
Data written by the writer2 is 635
Data read by the reader4 is 635
Data read by the reader4 is 635
Data written by the writer4 is 1275
Data read by the reader4 is 1275
Data written by the writer4 is 2555
Data read by the reader4 is 2555
Data written by the writer5 is 5115
Data read by the reader5 is 5115
Data read by the reader5 is 5115
```