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.

Constraints:

- Any number of Readers can access the database when there are no active writers.
- Writers can access database when there are no readers and other writers (i.e, only one at a time).
- At any time, Only a single thread can manipulate state variables.

Implementation of Queue:
A queue is used to maintain the blocked processes in their order of arrival. The first process to enter is waken up first so as to ensure that no process waits for too long. Besides, this helps to give a fair chance to both readers and writers to execute without any starvation.

```
class Queue{
   /* Queue implements the FIFO behavior
   where addition of new nodes takes place at the tail of the queue
   while deletion of node takes place at the head of the queue*/
   Node* Front; // front pointer to acess the first node of the queue; required during pop
operation
   Node* Rear; //rear pointer to acess the last node of the queue; required during pushing a
new node at the end of the queue
```

```
    public:
    // Implementation of push() that adds node at the queue's tail
            void push(pid_t val){
        Node* n = new Node();  // create a new node -> to be later pushed in the queue
        n->pid = val;       // initialize the pid value of the node with function's argument

        if(Front == NULL) Front = n; // if queue is empty, point the front to the newly added node
        else Rear->next = n; //else update the next pointer of rear to point to newly added node

        Rear = n; // update the rear pointer
    }

    // Implementation of pop() that takes out the node from the queue's front
    pid_t pop(){
        if(Front == NULL) return -1; // if queue is empty, underflow error as no node can be
popped out
        else{
            pid_t pop_id = Front->pid; // get the pid of the node to be popped

            Front = Front->next;  // update front pointer to point to the next node; indicating front
node has been popped

            if(Front == NULL) Rear = NULL;  // if after pop, queue is empty , update rear

            return pop_id;
        }
    }
};
```

Semaphore:

Semaphores are used as signalling mechanism between multiple co-operating processes to access the critical section. It has 2 atomic operations wait() and signal().

```
// Implementation of Semaphore (counting semaphore that can take any  int value)
class Semaphore{
    int value = 1;  // the shared variable to keep track of processes, the negative value
indicates no. of suspended processes
    Queue blocked = Queue(); // queue of blocked process waiting for wake up by some other
process

    public:
    /* wait() operation is used by a process whenever it wants to either access the critical
section or some shared resource,
    it checks the availability of resource and if possible then access it and block other
processes from accessing the same by decrementing the value
    else it itself gets blocked and move to blocked queue where it waits for some wake up call
by some other process
```

```
    */
    void wait(pid_t pid){
        value = value-1;
        if(value < 0){
            blocked.push(pid); // add this process to the blocked queue
            wait(pid); // put this process in wait state until waked up by some other process
        }
    }
    /*signal() operation is used by a process to wake up some sleeping process from the
blocked queue,
    it implies that now the resource has been used up by the previous process and is
available to be accessed by some of the sleeping processes
    */
    void signal(){
        value++;
        if(value <= 0){
            pid_t pid = blocked.pop(); // pop the front process from the blocked queue
            wakeup(pid);  // wakeup the popped process as it can again enter the ready state
and can try for its execution
        }
    }
};
```

Global Variables:

- 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:

First the reader checks for its turn using order_mutex, if some writer is in the critical section then a reader won't be allowed until it is done. Once the reader enters the entry section, it uses readers_mutex to ensure mutual exclusion i.e, only one reader can be in the entry section and can modify readers_count at a time. If it is the first reader then it waits for the access permission for the shared resource and after getting the access locks it for the writers. If multiple readers come then they need not to re-lock the resource again. Then the reader enters the critical section to perform the read operation from the shared resource. Multiple readers can be in the critical section to read the resource at the same time. When a reder is in exit section, again mutual exclusion is ensured as only single process can modify readers_count. If it is the last reader then, access to the resource is released so that any waiting writer can access it. Else, the resource won't be unlocked till reader's complete their operation. Finally, the readers_mutex is released as now any process can modify it.

```
void *reader(pid_t pid){
   /*         < ENTRY section>              */
  order_mutex.wait(pid); //check for its turn to get executed, if can't access the resorces
currently then wait till other processes are serviced
  readers_mutex.wait(pid); // requesting access to modify readers count, only one can modify
it at a time, mutual exclusion achieved

  readers_count++; // increment number of readers

  if(readers_count==1) access_mutex.wait(pid);
  // if it is the first reader then wait till other writers release the resource semaphore

  order_mutex.signal(pid); // release the semaphore to get the next reader/ writer to be
serviced
  readers_mutex.signal(pid); // release access to modify readers count


 /*        <CRITICAL SECTION>          */
  printf("Data read by the reader%d is %d\n",*((int *) pid),data);

  /*       <EXIT SECTION>             */

  readers_mutex.wait(pid); // again request access to readers count, onlu one can modify it at
a time
  readers_count--; //  decrement the count as a reader is leaving

  if(readers_count==0) access_mutex.signal(pid);
  // if it is the last reader then release the access to the critical section after reading is done
so that it can be used by any waiting writer

  readers_mutex.signal(pid); // release access to modify readers count
}
```

Writer’s Code:
Once the writer enters in the entry section it waits till other requests are serviced. Then it gets access to the resource and locks it until writing is done. It releases the order_mutex to allow other requests to be serviced meanwhile. After the writing operation is completed, it enters in exit section where it unlocks the resource for other processes.

```
void *writer(pid_t pid){
  /*       <ENTRY SECTION>     */
  order_mutex.wait(pid); //Wait in queue till other requests are serviced
  access_mutex.wait(pid); // get the access to the writer for accessing the critical section and block the readers from accessing it
  order_mutex.signal(pid); // release the semaphore to get the next reader/ writer to be serviced

  /*       CRITICAL SECTION       */
  data = data* 2 + 5; //modify the value of data during writing

  /*       EXIT section           */
  access_mutex.signal(pid); //release the access to the critical section so that it can be used by any waiting reader
}
```

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.