Starve Free Reader Writer Problem

by

Prakhar Agarwal
17118050

A Report
Submitted in requirement for the course
OPERATING SYSTEM (CSN-232)

Indian Institute of Technology, Roorkee
May, 2021

Chapter 1

Starve Free Readers-Writers Problem

The Reader-Writer problem is an example of a synchronization problem. There are two processes reader and writer that are sharing common resources e.g file at a time. The former one only reads the database, whereas the latter one wants to update (that is, to read and write) the database.
The problem is when one process starts writing the database and another process starts reading the database at the same time then the system will go into an inconsistent state. Another problem arises when one process starts writing the database and another process also starts writing. As both processes want to update the value of the data present in the file at the same time it will lead to inconsistency in the database.
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

1.1 The First Readers- Writers Problem

This is the reader's priority problem. In this, there is a stream of readers accessing the resource and lock all the writers who are trying to access it. It can cause starvation as the writers are waiting to modify the resource and new readers arrive all the time.
Consider the case with the help of an example; It is the case when one or more processes is reading the file and another process tries to update the data in the file at the same time.

Initial value of semaphore r_mutex = 1 and variable readers_count = 0

.Suppose two processes p0 and p1 are in a system, p0 wants to read while p1 wants to write. p0 enters first into the reader code, the moment p0 enters, wait(r_mutex); will decrease semaphore r_mutex by 1, now r_mutex = 0 and readers_count will be incremented to 1. If readers_count = 1 then wait(write_mutex); will set the value of semaphore write_mutex to 0 blocking all the writers to access the critical resource and signal(r_mutex); will increase r_mutex by 1, now r_mutex =1 allowing other readers to enter. Now any writer p1 wants to enter into its code then: p1 does sem_wait(write_mutex) since the write_mutex value is already 0, hence P1 can never write anything, till P0 is there.

1.2 The Second Readers–Writers Problem

The first problem has a drawback suppose Reader R1 is in the critical section and Reader R2 also wants to access the critical section and it is in race with Writer W1 , if Reader R2 access the resource ahead of Writer W1 and this cycle continues then Writer W1 would starve. This problem is called Writers prefernce problem. Initial value of semaphore write_mutex = 1 and variable readers_count = 0 Suppose two processes p0 and p1 are in a system, p0 wants to write while p1 wants to read. p0 does sem_wait(write_mutex) since the write_mutex value is 1 , it will decrement by 1 and set to 0 and p1 continues writing the file.... Now p1 enters and does wait(r_mutex) which will set r_mutex to 0 and readers_count incremented by 1. If readers_count = 1 then wait(write_mutex); will not allow the process to move further since write_mutex is set to 0 by p0 process.The writer will not be able to access the resource until the current reader has released the resource, which only occurs after the reader is finished with the resource in the critical section.

1.3 The Third Readers- Writers Problem

We have seen in first problem that Writer may starve and in second problem that reader may starve So the below solution is proposed to solve the problem of reader-writer starvation.

1.3.1 Solution

The synchronisation problem can be solved with the help of semaphore variables.

1) Semaphore 'write_mutex'

It is used by the process that is writing in the file and it ensures that no

other process should enter the critical section at that instant of time.

2) Semaphore 'r_mutex'
It is used to achieve mutual exclusion during changing the variable that is storing the count of the processes that are reading a particular file.

3) Semaphore 'seq_mutex'
It check for the turn to get executed, puts the blocked process in a fifo queue if it's not its turn currently

They are used to achieve mutual exclusion and initially their value is set to 1. readers_count variable is used to count the number of processes that are reading a particular file. Initially the value is initialized to 0. The wait() function is used to reduce the value of a semaphore variable by one. The signal() function is used to increase the value of a semaphore variable by one. All readers and writers will be granted access to the resource in their order of arrival (FIFOorder). 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.

1.3.2 Pseudo Code

Reader's Code:

```
void *reader(void *id)
{

Entry Section

wait(seq_mutex);
wait(r_mutex);
readers_count++;
if(readers_count==1)
wait(write_mutex);
signal(seq_mutex);
signal(r_mutex);

CRITICAL SECTION

Exit Section
```

```
wait(r_mutex);
readers_count–;
if(readers_count==0)
signal(write_mutex);
signal(r_mutex);
}
```

Writer's Code

```
void *writer(void *id)
{
sem_wait(seq_mutex);
sem_wait(write_mutex);
sem_post(seq_mutex);

CRITICAL SECTION

sem_post(write_mutex);
return id;
}
```

1.3.3 Correctness of Solution

1.Mutual Exclusion
The seq_mutex semaphore ensures that only a single writer can access the critical section at any moment of time thus ensuring mutual exclusion between the writers and also when the first reader try to access the critical section it has to acquire the seq_mutex lock to access the critical section thus ensuring mutual exclusion between the readers and writers.

2.Bounded Waiting
Before accessing the critical section any reader or writer have to first acquire the turn semaphore which uses a FIFO queue for the blocked processes. Thus as the queue uses a FIFO policy, every process has to wait for a finite amount of time before it can access the critical section thus meeting the requirement of bounded waiting.

3.Progress Requirement
The code is structured so that there are no chances for deadlock and also

the readers and writers takes a finite amount of time to pass through the critical section and also at the end of each reader writer code they release the semaphore for other processes to enter into critical section.

1.4 Reference

- https://afteracademy.com/blog/the-reader-writer-problem-in-operating-system
- https://afteracademy.com/blog/what-is-process-synchronization-in-operating-system
- https://www.javatpoint.com/os-readers-writers-problem