Starve_free_Readers-Writers-Problem

Readers Writers Problem is a classic synchronization problem in Operating Systems. It involves readers and writers trying to read or edit a document simultaneously which can lead to one or more writers simultaneously modifying information could lead to errors in documents. So, here too like other problems, the three conditions should hold to solve the race condition namely, mutual exclusion, progress and bounded waiting. ## Formulation of the problem There is a shared data between many users who perform reading and writing which could make the system faulty due to concurrency problems. So, we need to devise a ensure synchronization. Also, there are various solutions to this code in which the first readers writers solution, consists of starving the writers. The second Readers-Writers Problem gives priority to writers and the readers starve. In this repo, we are going to solve the readers writers problem such that both readers and writers don't starve.

Proposed Solution

I aim to solve starvation through utilising the FIFO order instead of allowing either reader or writer to pass first. For this purpose each process calls wait(enter) but reader processes before writers processes can happen in parallel so for reader modify rdcount and signal(enter) so that next process enters. If it's a writer process it will have to wait until write semaphore is released which is released after all readers before it completed their critical section. If it's a reader then it again modifies the rdcount and signals enter and it too goes into critical section.

The main difference here from Bounded Buffer being that readers can happen in parallel. So, while my writer's code resembles producer-consumer problem reader code is modified to get a starve free efficient solution. My main ideas come from the first and second readers-writers subproblems solved by sir in the class.

Pseudocode

```
typedef struct{
int val;
struct process *list;
}semaphore

wait(semaphore *s, int pid){
s->val--;
if(s->val<0){
add pid to s->list from end;                    //adding a process from the
tail of list
block();
}
}
signal(semaphore *s){
s->val++;
if(s->val<=0){
```

```
remove a process from head of list from s->list;            //removing from
head so that FIFO order is followed
wakeup(prid);
}
}
```

Reader's Code

```
/*Starting Section*/
wait(enter, prid)
rdcount++;
if(rdcount==1) wait(write);
signal(enter)
/*Critical Section*/
//Perform Reading
wait(exit, prid)
rdcount--;
if(rdcount==0) signal(write);
signal(exit)
/*Remainder Section*/
```

Writer's Code

```
/*Starting Section*/
wait(enter, prid)
wait(write, prid)
signal(enter)
/*Critical Section*/
//Perform Writing
signal(write)
//signal(enter) can also be written here instead of writing after wait(write,
prid).
/*Remainder Section*/
```

Verification of solution

Mutual Exclusion

The solution uses 3 semaphores to ensure mutual exclusion between writer-reader, writer-writer and rdcount variable modification. The semaphore enter is used to ensure that only one process enters the rdcount modification code for readers and also stops at the writer's end until all readers before him finish. This is done wityh the help of write semaphore which is signalled only after all readers before writer finish. So, after that either after writer performs writing or after wait(write) we can have the signal(enter) code as anyways it's release before writer writes will make it remain blocked in the if statement of rdcount segment. The exit semaphore ensures mutual exclusivity for decrementing rdcount after finishing read operation.

Progress

Progress condition states that the solution should ensure that either of the two processes executing in the critical section must do so with a non-zero speed. Also it speciifes that if a process is interested and there is no process is in the critical section then there should not be delayed access to the critical section. In our code if one process enters the critical section it will execute with some speed. But to prove the second condition we need to take a closer look at the pseudocode. Here if any process wants to enter it can enter the critical section if no process is inside because the enter semaphor would be 1 in that case. So, no interested process is delayed while accessing the critical section. So we can say that the progress condition holds as well.

Bounded Waiting

Bounded waiting as the name suggests is a condition that the solution should not make any process wait indefinitely and should be completed within a bounded time. This is satisfied since we use a list which is FIFO hence as soon as a process completes critical section the next process takes over. A process takes finite amount of time so, in a FIFO data structure it will have finite processes ahead of it thus giving a bounded waiting time.

No Busy Waiting

Our solution is not only starve free but also eliminates the possibility of deadlocks. The busy waiting condition is caused by the while loop usage in wait semaphore usually, but we have used block and wakeup instead thus eliminating busy waiting too. Also, the semaphore is also allowed to be negative to facilitate addition to list which can be used later to wakeup a process.

No Deadlocks

There is no possibility of deadlock as no two processes will form a cycle in RAG as semaphores used restrict entry into critical section and take care of no deadlocks.

Footnotes

Operating System Concepts by Abraham Silberschatz, Peter B. Galvin, Gerg Gagne

Prof. Sateesh Kumar Peddoju's Process Synchronization Slides