Subject Name: CS - 232 [Operating Systems]

ASSIGNMENT - Starve Free Readers-Writers Problem

Submitted By: Abdulahad Khan (17112002)

.

.

UNDER THE GUIDANCE OF:

Prof. Peddoju Sateesh Kumar

Department of Computer Science and Engineering

IIT ROORKEE

Chapter 1

Starve Free Readers-Writers Problem

When multiple process read and write to any shared resource, there are chances of indefinite starvation. So, this problem deals with multiple processes reading and writing to a shared resource synchronously, in such a way that, neither the reader nor writer gets starved for indefinite time.

1.1 Starve Free Readers-Writers Solution:

Normally, the classical solution of the problem leads to starvation of either reader or writer. In this solution, a semaphore is used to handle the processes in FIFO (First In First Out) order, such that it gives us *starve free solution.*

1.2 Semaphore Design:

This semaphore will be used to store the list of blocked processes in a FIFO queue.

- Design of Process Block

```
// Process Block Structure
struct Process{
    int process_pid;
    Process* next;
}
```

- Design of FIFO Queue of Process Nodes

```
// FIFO Queue implementation of Process nodes
struct Queue{
    Process* qFront, qRear;

    int push(int pid){
```

```
        Process* pro = new Process();
        pro->process_pid = pid;
        if(qRear == NULL){
            qFront = qRear = pro;
        }
        else{
            qRear->next = pro;
            qRear = pro;
        }
    }

    int pop(){
        if(qFront == NULL){
            return -1;          // Underflow condition
        }
        int pid = qFront->process_pid;
        qFront = qFront->next;

        if(qFront==NULL) qRear = NULL;
        return pid;
    }
}
```

- Design of Semaphore using FIFO Queue

```
// This is a FIFO semaphore
struct Semaphore{
    int val = 1;                // value
    Queue* q = new Queue();     // this is a FIFO Queue

    void wait(int pid){
        // pid denotes process id

        val--;
        if(val < 0){
            q->push(pid);
            block(pid);
            /* this will block particular process, using the system call
             and will be sent to waiting queue, unless the wake() is called.*/
        }
    }

    void signal(){
        val++;
        if(val<=0){
            int processId = q->pop();
            wake(processId);
            // to wakeup the process with the given process id
        }
```

```
        }
    }
```

1.3 Semaphore Initialization:

```
int reader_count = 0;
// to indicate how many readers are executing critical section

Semaphore next_turn = new Semaphore();
/* semaphore maintaining the order in which the reader
and writer are requesting access to critical section */

Semaphore token = new Semaphore();
// required semaphore to access the critical section

Semaphore r_change = new Semaphore();
// semaphore to change the reader_count
```

1.4 Reader's Process:

```
do{
/* ENTRY SECTION */
    next_turn->wait(process_id);
        // process waiting for its turn to get executed
    r_change->wait(process_id);
        // process requesting access to change reader_count
    reader_count++;
        //updating the number of readers trying to access critical section
    if(reader_count==1)
        // if this is the first reader then request access to critical section
        token->wait();
            //requesting  access to the critical section for readers

    turn->signal();
        //releasing turn so that the next reader or writer can take the token
    r_change->signal();
        //release access to the reader_count
/* CRITICAL SECTION */

/* EXIT SECTION */
    r_change->wait(process_id)
        //requesting access to change reader_count
    reader_count--;
        //a reader has finished executing critical section so reader_count
        // will decrease by 1
    if(reader_count==0)
        //if all the reader have finished executing their critical section
```

```
        token->signal();
            //releasing access to critical section for next reader or writer
    r_change->signal();
        //release access to the reader_count

/* REMAINDER SECTION */

}while(true);
```

1.5 Writer's Process:

```
do{
/* ENTRY SECTION */
  next_turn->wait(process_id);
    //waiting for its turn to get executed
  token->wait(process_id);
    //requesting  access to the critical section
  turn->signal(process_id);
    /*releasing turn so that the next reader or writer
    can take the token and can be serviced*/

/* CRITICAL SECTION */

/* EXIT SECTION */
  token->signal()
    //releasing access to critical section for next reader or writer

/* REMAINDER SECTION */

}while(true);
```

1.6 Conclusion:

Here, Multiple readers can simultaneously read the shared resource. But, Once the writer has entered the waiting list, none of the other process can get access to the resource.

The correctness of solution can be justified based on Mutual Exclusion and Bounded Waiting

The token semaphore is used to make sure, that only a single process has access to the critical section at any point of time, which satisfies Mutual Exclusion. The reader or writer process has to first acquire next-turn semaphore (using FIFO queue for storing blocked processes), then only it can enter the critical section. Thus, it is waiting for finite amount of time, before accessing critical section. Hence it has bounded waiting.

1.7 References:

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne - Operating System Concepts
- https://arxiv.org/abs/1309.4507
- https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem