STARVE-FREE READERS WRITER PROBLEM

OPERATING SYSTEMS

CSN - 232

JINENDRA VERMA
19114038
SUB BATCH O2

What is the reader writer problem?

A reader writer problem is a situation when multiple processes are trying to access(read) and edit(write) the same data structure or shared file simultaneously. In the classical solution of this problem, there is starvation for either reader or writer. So in a starve free solution of the problem to avoid this starvation, only one writer is allowed to access the critical section at any point of time and when there is no writer active then any number of readers can access the critical section.
In our solution we have used three semaphores namely **chance_queue,req** and **r_mutex.** chance_queue represents the chance of the next process to enter the critical section, req is the semaphore required to access the critical section and r_mutex is required to change the read_count variable.

Code explanation:

FIFO

```
struct Node{
    Node* next;
    pid_t value;
};
```

A queue node having a pointer to the next node and value of pid_t data type(a signed integer type which is capable of representing a process ID).

```
class Queue{
    Node* Front, *Rear;

    public:
```

```
    void push(pid_t val){
        Node* new_process = new Node();             //new node(process)
        new_process->value = val;

        if(Rear != NULL){
            Rear->next = new_process;               //if queue is empty
        }else{
            Front = new_process;
        }

        Rear = new_process;                         //pointing rear to new
node
    }
pid_t pop(){

        if(Front == NULL){
            return -1;
        }
        else{
            pid_t process_id = Front->value;
            Front = Front->next;
            if(Front == NULL) Rear = NULL;

            return process_id;
        }
    }
};
```

A class queue for queue implementation having two nodes front and rear for pop operating and node addition respectively. Push and pop functions for adding a new node at the rear side and popping a node from the front side.

SEMAPHORE IMPLEMENTATION:

```
class Semaphore{
    int value = 1;
```

```
    Queue processes = Queue();          //processes that are to be
executed

    public:

    void wait(pid_t process_id){
        value--;
        if(value < 0){
            processes.push(process_id);
            wait(process_id);
        }
    }

    void signal()
    {
        value++;
        if(value <= 0){
            pid_t pid = processes.pop();
            wakeup(pid);
        }
    }
};
```

A class semaphore having two functions wait and signal for adding a process at queue,decrementing the value and blocking it till it gets the turn(wait) and for getting the next process from the queue and wake up for its operation and turn(signal).

DECLARATION:

```
//Declare Semaphores
struct Semaphore chance_queue;
struct Semaphore req;
struct Semaphore r_mutex;

int read_count = 0;
int data = 1;
```

Declaring above mentioned three semaphores chance_queue,req and r_mutex. A read counter and data as 0 and 1 resp.

READER FUNCTION:

```
void *reader(pid_t process_id){

    //  ENTRY SECTION
    chance_queue.wait(process_id);
    r_mutex.wait(process_id);

    read_count++;

    if(read_count == 1){
        req.wait(process_id);
    }

    chance_queue.signal();
    r_mutex.signal();

    //  CRITICAL SECTION
    printf("Readed data is: %d",data);

    //  EXIT SECTION
    r_mutex.wait(process_id);
    read_count--;

    if(read_count == 0)
        req.signal();

    r_mutex.signal();

}
```

This method allows multiple readers to read at the same time. First reader waits to acquire chance_queue semaphore or waits for its turn to get executed. Then it requests access to change the read_count variable and then updates the number of readers. If the current reader is the first reader

under execution then the reader waits for other writers to release resource semaphore and acquires it. After the Entry section the three semaphores are released for other processes.
In the critical section, Reader performs read operation on data from memory.
In the Exit section, the reader again waits till other readers have updated the read_count variable and then updates it. If there are no active readers remaining then *req* is released and after it r_mutex semaphore is also released for other processes.

WRITER'S CODE:

```
void *writer(pid_t process_id){
    //  ENTRY SECTION
    chance_queue.wait(process_id);
    req.wait(process_id);
    chance_queue.signal();

    //  CRITICAL SECTION
    data += 2;

    //  EXIT SECTION
    req.signal();
}
```

First the writer waits for its turn.Now it waits for its turn to modify *data* variable by getting *req* Semaphore. Finally, in Entry Section chance_queue Semaphore is released for other processes and the writer enters the Critical Section.
Inside the Critical Section, Writer performs any kind of modification to *data* variable in memory. Here I have added 2 in *data* variable. In the End, Writer releases *req* for other Reader/Writers.