A Report

On

Starve Free Reader Writer Problem

Submitted in requirement for the course

OPERATING SYSTEM (CSN-232)

of Bachelor of Technology in Computer Science and Engineering

by

Aryan Jain

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE
ROORKEE- 247667 (INDIA)

May 2021

Contents

Reader-Writer Problem

1.1 Introduction

A classical Reader Writer problem is a situation where multiple processes can access and modify a shared file or data structure concurrently. In such a situation in order to avoid the **race condition** only one writer should be allowed to access the critical section in any point of time and also when no writer is active any number of reader can access the critical section. So to solve this synchronization problem Semaphores are used

1.2 Starve Free Solution

The classical solution of the problem results in starvation of either reader or writer. In this solution, I have tried to propose a starve free solution. While proposing the solution only one assumption is made i.e. Semaphore preserves the first in first out(FIFO) order when locking and releasing the processes(Semaphore uses a FIFO queue to maintain the list of blocked processes).

1.2.1 Initialization

In this solution 3 semaphores are used. **turn** semaphore is used to specific whose chance is it next to enter the critical section. The process holding this semaphore gets the next chance to enter the critical section. **rwt** semaphore is the semaphore required to access the critical section. **r_mutex** semaphore is required to change the **read_count** variable which maintain the number of active reader.

Pseudo Code	**Comments**
read_count = Integer(0);	Integer representing the number of reader executing critical section
turn = Semaphore(1);	semaphore representing the order in which the writer and reader are requesting access to critical section
rwt = Semaphore(1);	semaphore required to access the critical section
r_mutex = Seamaphore(1);	seamphore required to change the read_count variable

1.2.2 Reader's Code

Pseudo Code	**Comments**
do {	

<ENTRY SECTION>
wait(turn);
wait(r_mutex);
read_count++;
if(read_count == 1)
wait(rwt);
signal(turn);
 	
signal(r_mutex);
	
	waiting for its turn to get executed
	requesting access to change read_count
	update the number of readers trying to access critical section
	if I am the first reader then request access to critical section
	requesting access to the critical section for readers
	releasing turn so that the next reader or writer can take the token and can be serviced
	release access to the read_count

<CRITICAL SECTION>	

<EXIT SECTION>
wait(r_mutex);
read_count–;
if(read_count==0)
signal(rwt);
signal(r_mutex);
	
	requesting access to change read_count
	a reader has finished executing critical section so read_count decrease by 1
	if all the reader have finished executing their critical section
	releasing access to critical section for next reader or writer
	release access to the read_count

<REMAINDER SECTION>	
} while(true);	

1.2.3 Writer's Code

Pseudo Code	**Comments**
do {	

<ENTRY SECTION>
wait(turn);
wait(rwt);
signal(turn);
	
	waiting for its turn to get executed
	requesting access to the critical section
	releasing turn so that the next reader or writer can take the token and can be serviced

<CRITICAL SECTION>	
<EXIT SECTION> **signal(rwt);**	 releasing access to critical section for next reader or writer
<REMAINDER SECTION>	
} while(true);	

1.3 Correctness of Solution

1.3.1 Mutual Exclusion

The **rwt** 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 **rwt** mutex lock to access the critical section thus ensuring mutual exclusion between the readers and writers.

1.3.2 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.

1.3.3 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.