**Exercise 5 – Reader Writer Problem**

**Notes from Internet – Series-5**

### **Introduction**

In computer science, **concurrency** is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome. This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems.

In more technical terms, concurrency refers to the **decomposability** property of a program, algorithm, or problem into order-independent or partially-ordered components or units.

Because computations in a concurrent system can interact with each other while being executed, the number of possible execution paths in the system can be extremely large, and the resulting outcome can be indeterminate. Concurrent use of shared resources can be a source of indeterminacy leading to issues such as **deadlocks**, and **resource starvation**.

Design of concurrent systems often entails finding reliable techniques for coordinating their execution, data exchange, memory allocation, and execution scheduling to minimize response time and maximise throughput.

## **Process Synchronization**

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be **synchronized** so that their order of execution can be guaranteed.

The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization. There are various synchronization mechanisms that are used to synchronize the processes.

That means, **Process Synchronization** was introduced to handle problems that arose while multiple process executions.

One of the These problems are :-

1. **Critical Section Problem**

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

A critical section have following regions :

1. Entry Section  
   ii. Critical Section  
   iii. Exit Section  
   iv. Remainder Section

A solution to the critical section problem must satisfy the following three conditions:-

a. **Mutual Exclusion**

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

b. **Progress**

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

c. **Bounded Waiting**

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Process Synchronization are handled by two approaches:

1. **Software Approach** –

In Software Approach, Some specific Algorithm approach is used to maintain synchronization of the data. Like in Approach One or Approach Two, for a number of two process, a temporary variable like (turn) or boolean variable (flag) value is used to store the data. When the condition is True then the process in waiting State, known as Busy Waiting State. This does not satisfy all the Critical Section requirements.

Another Software approach known as Peterson’s Solution is best for Synchronization. It uses two variables in the Entry Section so as to maintain consistency, like Flag (boolean variable) and Turn variable(storing the process states).

1. **Hardware Approach** –

The Hardware Approach of synchronization can be done through Lock & Unlock technique.Locking part is done in the Entry Section, so that only one process is allowed to enter into the Critical Section, after it complete its execution, the process is moved to the Exit Section, where Unlock Operation is done so that another process in the Lock Section can repeat this process of Execution.

### **Mutex Locks**

As the hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

### **Semaphores**

It is a significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.  
So it is basically a **synchronizing tool** and is accessed only through two low standard atomic operations, wait and signal designated by **P(S)** and **V(S)** respectively.

In very simple words, semaphore is a **variable** which can hold only a non-negative Integer value, shared between all the threads, with operations wait and signal, which work as follow:

P(S): if S ≥ 1 then S := S - 1

else <block and enqueue the process>;

V(S): if <some process is blocked on the queue>

then <unblock a process>

else S := S + 1;

The classical definitions of wait and signal are:

1. Wait: **Decrements** the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).
2. Signal: **Increments** the value of its argument S, as there is no more process blocked on the queue.

## **Classical Problems of Synchronization :-**

1. **Using Semaphore :-**

Semaphore can be used in other synchronization problems besides Mutual Exclusion.  
Following are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

### Problem :-

* There is a shared resource which should be accessed by multiple processes.
* There are two types of processes in this context.
* They are **reader** and **writer**.
* Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource.
* When a writer is writing data to the resource, no other process can access the resource.
* A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

### **Solution 1 Using Semaphores :-**

Here, readers have **higher priority** than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.  
Here, we use :-

* one mutex **m** and a semaphore **w**.
* An integer variable **read\_count** :- used to maintain the number of readers currently accessing the resource. The variable read\_count is initialized to 0.
* A value of **1** is given initially to **m** and **w**.  
  Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the **read\_count** variable

a. **Writer Process** :

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

while(TRUE)

{

wait(w);

/\* perform the write operation \*/

signal(w);

}

b. **Reader Process** :

1. Reader requests the entry to critical section.
2. If allowed:  
   i. it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **w** semaphore to restrict the entry of writers if any reader is inside.  
   ii.It then, signals mutex as any other reader is allowed to enter while others are already reading.  
   iii. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore **w** as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

while(TRUE)

{

//acquire lock

wait(m);

read\_count++;

if(read\_count == 1)

wait(w);

//release lock

signal(m);

/\* perform the reading operation \*/

// acquire lock

wait(m);

read\_count--;

if(read\_count == 0)

signal(w);

// release lock

signal(m);

}

Thus, the semaphore **w** is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

## **Limitations of Semaphores**

* Priority Inversion is a big limitation of semaphores.
* Their use is not enforced, but is by convention only.
* With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in details in coming lessons.

## **C++** Implementation Of reader writer problem using semaphore

#include<semaphore.h>

#include<stdio.h>

#include<pthread.h>

# include<bits/stdc++.h>

using namespace std;

void \*reader(void \*);

void \*writer(void \*);

int readcount=0,writecount=0,sh\_var=5,bsize[5];

sem\_t x,y,z,rsem,wsem;

pthread\_t r[3],w[2];

void \*reader(void \*i)

{

cout << "\n-------------------------";

cout << "\n\n reader-" << i << " is reading";

sem\_wait(&z);

sem\_wait(&rsem);

sem\_wait(&x);

readcount++;

if(readcount==1)

sem\_wait(&wsem);

sem\_post(&x);

sem\_post(&rsem);

sem\_post(&z);

cout << "\nupdated value :" << sh\_var;

sem\_wait(&x);

readcount--;

if(readcount==0)

sem\_post(&wsem);

sem\_post(&x);

}

void \*writer(void \*i)

{

cout << "\n\n writer-" << i << "is writing";

sem\_wait(&y);

writecount++;

if(writecount==1)

sem\_wait(&rsem);

sem\_post(&y);

sem\_wait(&wsem);

sh\_var=sh\_var+5;

sem\_post(&wsem);

sem\_wait(&y);

writecount--;

if(writecount==0)

sem\_post(&rsem);

sem\_post(&y);

}

int main()

{

sem\_init(&x,0,1);

sem\_init(&wsem,0,1);

sem\_init(&y,0,1);

sem\_init(&z,0,1);

sem\_init(&rsem,0,1);

pthread\_create(&r[0],NULL,(void \*)reader,(void \*)0);

pthread\_create(&w[0],NULL,(void \*)writer,(void \*)0);

pthread\_create(&r[1],NULL,(void \*)reader,(void \*)1);

pthread\_create(&r[2],NULL,(void \*)reader,(void \*)2);

pthread\_create(&r[3],NULL,(void \*)reader,(void \*)3);

pthread\_create(&w[1],NULL,(void \*)writer,(void \*)3);

pthread\_create(&r[4],NULL,(void \*)reader,(void \*)4);

pthread\_join(r[0],NULL);

pthread\_join(w[0],NULL);

pthread\_join(r[1],NULL);

pthread\_join(r[2],NULL);

pthread\_join(r[3],NULL);

pthread\_join(w[1],NULL);

pthread\_join(r[4],NULL);

return(0);

}

### Output :

student@sh-4.4-desktop:~$ gcc rw1.c -lpthread

student@sh-4.4-desktop:~$ ./a.out

-------------------------

reader-0 is reading

updated value : 5

writer-0 is writing

-------------------------

reader-1 is reading

updated value : 10

-------------------------

reader-2 is reading

updated value : 10

-------------------------

reader-3 is reading

updated value : 10

writer-3 is writing

-------------------------

reader-4 is reading

### **Using Monitor :-**

Monitors provide a structured concurrent programming primitive, which is used by processes to ensure exclusive access to resources, and for synchronizing and communicating among users. A monitor module encapsulates both a resource definition and operations/ procedures that exclusively manipulate it. Those procedures are the gateway to the shared resource and called by the processes to access the resource. Only one call to a monitor procedure can be active at a time and this protects data inside the monitor from simultaneous access by multiple users. Thus mutual exclusion is enforced among tasks using a monitor. Processes that attempt monitor entry while the monitor is occupied are blocked on a monitor entry queue.

Monitor is one of the ways to achieve Process synchronization. Monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can’t access the internal variable of monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

### Syntax :-

< Monitor-Name > : monitor

begin

Declaration of data local to the monitor.

.

.

procedure < Name > ( < formal parameters > );

begin

procedure body

end;

Declaration of other procedures

.

.

begin

Initialization of local data of the monitor

end;

end.

*Note that a monitor is not a process, but a static module of data and procedure declarations. The actual processes which use the monitor need to be programmed separately.*

## **Limitations of Monitors**

Since only one process can be active within a monitor at a time, the absence of concurrency is the major weakness in monitors and this leads to weakening of encapsulation. For example, in the Readers-Writers problem, the protected resource (a file, database, etc.) is separated from the monitor and so, there is a possibility that some malicious Reader or Writer could simply access the database directly without getting a permission from the monitor to do so.

Another problem is the possibility of deadlock in the case of nested monitor calls. For example, consider a process calling a monitor that in turn calls another lower-level monitor procedure. If a wait is executed in the last monitor called, the mutual exclusion will be relinquished by the process. However, mutual exclusion will not be relinquished by processes in monitors from which nested calls have been made. Processes that attempt to invoke procedures in these monitors will become blocked. If those blocked processes are the only ones which can cause signaling to occur in the lower level monitor, then deadlock occurs

## **Implementation of Reader-Writer Problem**

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <semaphore.h>

# include<bits/stdc++.h>

using namespace std;

// Define the data we need to create the monitor

struct monitor\_DataType {

sem\_t OKtoRead;

sem\_t OKtoWrite;

int readerCount;

int isBusyWriting;

// The read-queue

int readRequested;

};

struct monitor\_DataType monitor\_data;

// Function that will block until write can start

void monitor\_StartWrite() {

if(monitor\_data.isBusyWriting || monitor\_data.readerCount != 0){

sem\_wait(&(monitor\_data.OKtoWrite));

}

monitor\_data.isBusyWriting++; // Using 1 as true

}

// Function to signal reading is complete

void monitor\_EndWrite() {

monitor\_data.isBusyWriting--;

if(monitor\_data.readRequested){

sem\_post(&(monitor\_data.OKtoRead));

} else {

sem\_post(&(monitor\_data.OKtoWrite));

}

}

// Function that will block until read can start

void monitor\_StartRead() {

if(monitor\_data.isBusyWriting){

monitor\_data.readRequested++;

sem\_wait(&(monitor\_data.OKtoRead));

monitor\_data.readRequested--;

}

monitor\_data.readerCount++;

sem\_post(&(monitor\_data.OKtoRead));

}

// Function to signal reading is complete

void monitor\_EndRead() {

monitor\_data.readerCount--;

if(monitor\_data.readerCount == 0){

sem\_post(&(monitor\_data.OKtoWrite));

}

}

// intialize the monitor

// return's 0 on success, just like sem\_init()

int monitor\_Initialized(){

int returnValue = 1;

// Initialize the structure

monitor\_data.readerCount = 0;

monitor\_data.isBusyWriting = 0;

monitor\_data.readRequested = 0;

// initialize the semaphores

if(sem\_init(&(monitor\_data.OKtoWrite), 0, 1) == 0 &&

sem\_init(&(monitor\_data.OKtoRead), 0, 1) == 0){

returnValue = 0;

} else {

cout&lt;&lt;"Unable to initialize semaphores\n";

}

return returnValue;

}

// Destroys the semphores.

void monitor\_Destroy(){

sem\_destroy(&(monitor\_data.OKtoWrite));

sem\_destroy(&(monitor\_data.OKtoRead));

}

int main() {

if(monitor\_Initialized() == 0){

cout << "Initialized\n";

monitor\_StartWrite();

cout << "Writing stuffs...\n";

monitor\_EndWrite();

monitor\_StartRead();

cout << "Reading stuffs...\n";

monitor\_EndRead();

monitor\_Destroy();

}

return 0;

}

### Output :

**Initialized**

**Writing stuffs...**

**Reading stuffs...**

**Refer**

[GitHub - codophobia/readers-writers-solution-in-c: First readers writers solution using semaphore and mutex in c](https://github.com/codophobia/readers-writers-solution-in-c)

[GitHub - 28Ritu/Reader-Writer-Problem: C Code for readers-writers problem in OS.](https://github.com/28Ritu/Reader-Writer-Problem)