HomeWork 3

RISHI KUMAR SONI 1001774020

QUESTION 4.4

```
public void waitC() {
++numWaitingThreads;
threadQueue.VP(mutex);
mutex.P();
--numWaitingThreads;
}
public void signalC() {
if (numWaitingThreads > 0)
    threadQueue.V();
// continue in the monitor; perhaps send more signals }
```

I .

Why is this implementation incorrect

SOLUTION :

There can be many cases in this scenario:

CASE 1 :

WHEN BOTH THE THREADS EXECUTE THE ***WAITC*** OPERATION :

- Suppose we have two threads T1 and T2 respectively . Now T1 execute the waitc Operation(), as a result , the T1 get into the thread Queue and as a result , thus mutex gets unlock for the another thread, so that another thread can access the operations.

Now Thread T2 comes , it executes the same operation, which as a result making the T2 to run the same

Operation i.e wait operation () and as a result the thread T2 will also in get the thread Queue.

IF we don't have any other Threads to execute, then threads T1 and T2 will remain in the thread Queue forever and as a result , Our implementation ***FAILS.***

CASE 2 :

Suppose we have three threads T1, T2 , T3 .

T1 comes and execute the WaitC operation.

After execution, it went to the Queue.

Now Thread T2 comes and execute the SignalC operation.

T2 will be in the monitor section, and it will signal to the thread in the Queue i.e , T1.

T2 exits the sections.

Currently there are two threads in the entry section, T1 AND T3.

Now , any thread can come and starts execution , in our case T3 executes.

T3 executes the SignalC operation, thus it will enter the monito section.

There are no waiting thread in the Queue, then T3 wont signal anyone and it exit the monitor section.

Now, when the T1 execute WaitC operation, it get into the Queue and mutual exclusion will unlock for the another thread.

Thus this situation is same as the CASE 1, so

Thus, this implementation also ***FAILS.***

DeadLock Condition.

CASE 3 :

Because of the IF loop.

Thus, IF loop won't call the thread repeatedly. As the result, the condition cannot check properly and as a result the implementation ***FAILS.***

II. Consider what can happen if two successive signal operations are performed on a condition variable on which one thread is waiting.

SOLUTION:

Considering we have 2 Threads T1 and T2.

T1 executes the SignalC operation, remain in the monitor and finishes It.

Calling the thread from the Queue.

Thus, there is no thread In the Queue. So, no Thread gets awaken.

Now, the thread T2 will execute, the monitor and finishes it.

After the completion. It also the signals to the thread in the Queue.

No, Thread in the Queue found.

In this case, all threads got executed and we achieved our target of Using the thread with its proper execution.

4.11. Below isanimplementation ofthe strategyR<W.2using SCmonitors. This strategy allows concurrent reading and generally gives writers a higher priority than readers. Readers have priority in the following situation: when a writer requests to write, if it is a lead writer (i.e., no other writer is writing or waiting), it waits until all readers that arrived earlier have finished reading.

(a) Is this implementation correct? If not, give a scenario that demonstrates any errors that you found. Methods startRead() and startWrite() contain if-statements that determine whether a waitC() operation is executed.

SOLUTION :

CASE 1:

According to the R < W.2. It states that, all the readers finishes first, then only the writer is able to perform the action.

Suppose we got two writers W1, W2, R1.

Now, Readers execute the startRead(), it enters the Monitor, and execute the code and exit the code.

Suddenly after executing the exit monitor() in startRead, the writers W1 enters and it start executing the startWrite code.

Thus in this case***, if condition becomes true*** , as a result , writer will enter in the Queue and waitC operation will perform.

And after that it exits the monitor from the startWrite section.

After executing the exit monitor, Second writer W2 comes and it start executing the startWriter code and also execute the endWriter operation.

Thus when, it execute the endWriter operation() , ***if condition gets satisfy*** and as a result, W1 wakes up, and it execute the endWriter() operation.

In ,the above scenario , there are no more writers in the queue and as a result the if condition becomes false and as a result, else part is executed. It makes the reader.signalC() to execute.

The initial reader R1 , who is in the queue awakes.

After waking up the reader R1 ,

the end of the Else statement takes place, which makes the writer endWriter () to exits the monitor.

Which ***FAILS*** our ***R < W2*** condition.

The reason being all the writer executed firstly but in above case Readers have the highest priority.

CASE2 :

It is using the SC monitors. And it also including ***the if statement*** which basically means, that the condition won't get check repeatedly.

Which also causes the condition, to ***FAIL.***

b. If these if-statements are changed to while-loops, is the implementation correct? If not, give a scenario that demonstrates any errors that you found.

SOLUTION:

Yes, this implementation is correct.

If we use while loop in the endWrite() operation, the writer will check the reader exists in the queue or not.

If it exists then, it will signal to the reader from the queue.

All the reader's perform the endReader operation, after completion of the endReader only writers exits the monitor , which is true and this implementation is correct.

(c) If you believe that the implementations in parts (a) and (b) are incorrect, show how to modify the implementation below to correct it.

SOLUTION:

We can modify the solution, by the implementing the while loop instead of the if else loop.

Replacing ,

```
if (!(writerQ.empty()))
writerQ.signalC();
else {
waitForReaders = readerQ.length();
for (int i = 1; i <= waitForReaders; i++)
readerQ.signalC();
}
```

By

```
While(writeQ.empty())
{
waitForReaders = readerQ.length();
for(int I = 1 ; i<=waitForReaders;i++)
readersQ.signalC();     // SIGNAL ALL THE SLEEPING READERS
}
exitMonitor()          // Exit when all the Readers finish the endOperation.
```

NOTE :

Considering that after completing the while loop, only then exit Monitor works.

4.18

(a)Assume that the monitor is still of type SU; does the revised solution still implement strategy R>W.1?

SOLUTION :

When we apply the ***Signaling Urgent discipline***, then it work properly.

Thus When we have Thread T1 which try to execute the StartRead Operation , it will first execute the reader.wait( ) operation. In which the Thread get into the Queue. After that, the execution of the readerQ.signal () takes place.

Which awakes the thread, which just entered into the Queue.

The awake Thread will be in the ***Monitor section.***

As a result of this the reader doesn't wait in the Queue and reader get the Signal immediately which make it to keep on executing.

Yes, it implements the strategy.

(C)Assume that the monitor is still of type SC; does the revised solution still implement strategy R>W.1?

SOLUTION :

When we use the Signaling Continuous discipline, then it won't create any error and our implementation will run smooth and correctly.

Yes, it implements the strategy.

By implementing

public void startRead {

if (writing)

readerQ.wait();

readerQ.signal();

++readerCount; }

The only error can happen, in ***the IF Statement***.

Because it won't check condition, repeatedly.

But this error won't happen all the time.

It ***FAILS.***

4.25

(a) Give a scenario for Solution 1 that illustrates how a philosopher can starve.

SOLUTION:

Suppose we have 3 Philosopher P0, P1, P2 and 3 folks F0, F1 , F2.

Step 1 : P0 starts it's execution and access the pickup operation ,then test operation execute and P0 is in the eating state.

Step 2 : P1 try to execute the pickup state and run the test , but it denied because the P0, already eating.

Step 3 : P2 try to execute the pickup state and run the test operation, but denied because its one of the folk is in the P0.

Step 4 : P0 execute the putDown operation () . it's state is changed to the thinking state.

Test test(i-1)%n will check it's neighbor philosopher. Thus philosopher P1 neighbor P0 changed it's state to Thinking and P2 is already in the Hungry state so it's satisfy the condition.

Now P1 starts eating.

Step 5 : Now test(i+1) % n will execute , thus it implies to the P2 philosopher.

P2 try to execute the operation , but p1 is already is in the eating state and folks are occupied so P2 can't access .

Thus, P2 is in the ***Starving state.***

It is DeadLock.

(b) ***Show how this scenario is avoided in Solution 2.***

SOLUTION:

By using the Starving priority.

In solution 2, this condition is avoided by declaring the ***starving variable***, which give highest priority to the starving thread.

It first check is there any starving thread or not, if have they It will execute otherwise it will work normally.

We added the else condition in the solution2, that check the starving priority.