HomeWork 2

RISHI KUMAR SONI 1001774020

Section 3.5.5 shows how to use binary semaphores to implement P() and V() operations for a countingSemaphore class. Suppose that binary semaphores are redefined so that a V() operation never blocks the calling thread. That is, if the value of a binary semaphore is 1 when a V() operation is called, the value is not modified and the calling thread simply completes the operations and returns.

(a) Are the implementations of P() and V() still correct if these new binary semaphores are used? Explain.

(b) If not, suppose that we replace the statements

mutex.V(); delayQ.P();

with the single statement

delayQ.VP(mutex);

Is this new implementation correct? Explain.

<u>SOLUTION :</u>

a .

If we consider that the value of the binary Semaphore values is **always 1**, when it execute the V operation.

The permit value will always be 1(given according to the question).

Then **if condition** of the V operation (Exit Section) doesn't execute and as a result, the threads will not get suspended or get in the queue. And it will never execute the delayQ.v();

Thus it directly exit the critical section , without waking up the other threads.

Rather they Exit the critical Section, without waking up or notify the queue threads.

It would be incorrect to implement with the fix value, the reason being the delay.V never get execute and as a result , once the Thread is added in the waiting list or Thread is in the sleep mode, it will remain forever.

Also, it violates the binary semaphore condition.

Its an Incorrect Implementation.

b.

delayQ.VP(mutex) = Exit the critical Section.

So, we can conclude that whenever the value of the permit is less then 0 , it will exit the critical section.

Thus V operation is completed in this section only.

Thus this condition fails .

Concurreny will be failed.

3.6. In Section 3.5.4 a semaphore-based solution to strategy R<W.2 for the readers and writers problem is given. (a) Below is a revised version of the read operation, in which semaphore mutex r is deleted and semaphore readers w que is used to provide mutual exclusion for accessing activeReaders and for synchronization between readers and writers. Does the revised solution still implement strategy R<W.2? Explain.

```
Read()
 { readers_w_que.P(); // readers may be blocked by a writer
 ++activeReaderCount;
if (activeReaderCount = 1)
 writer_que.P()
EXERCISES 169
readers_w_que.V();
 /* read x */
 readers_w_que.P();
--activeReaderCount;
 if (activeReaderCount = 0)
writer_que.V();
readers_w_que.V();
}
```

(b) Below is a revised version of the read operation in which the last two statements have been reversed. Does the revised solution still implement strategy R<W.2? Explain.

```
Read()
 { readers_w_que.P();
 mutex_r.lock();
++activeReaders;
if (activeReaders == 1) writers_que.P();
 mutex_r.unlock();
 readers_w_que.V();
/* read x */
mutex_r.lock();
 --activeReaders;
mutex_r.V(); // ***** these two statements
 if (activeReaders == 0)
 writers_que.V(); // ***** were reversed
 }
```

<u>SOLUTION :</u>

a.

Deleting the mutex r , means we have multiple readers in the critical Section. Which fails the race condition.

Also , if we don't have the mutex_r , then the reader wont lock the critical section. And there are chances for the writers to influence the condition.

This strategy Fails.

It will not implement R< W2.

b.

If we reverse the statement, then this statement works for the readers but it wont work for the writer.

The reason being that if we have more then one writers in the queue and if we don’t have the mutex.lock and mutex.unlock for the writers then, there will be more then one writer in the critical section and this process never ends.

So, For this strategy for the writer Fails but for the readers it will work.

3.10. Solution 4 of the dining philosophers problem allows a philosopher to starve. Show an execution sequence in which a philosopher starves.

<u>Solution :</u>

```
philosopher(int i /* 0..n-1 */)

{

 while (true) {

 /* think */

 mutex.P()

state[i] = hungry;

test(i); // performs self[i].V() if philosopher i can eat mutex.V();

 self[i].P(); // self[i].P() will not block if self[i].V() was

 /* eat */ // performed during call to test(i)

 mutex.P();

state[i] = thinking;

 test((n + i - 1) %n); // unblock left neighbor if she is hungry and can eat

test((i +1) %n); // unblock right neighbor if she is hungry and can eat
mutex.V(); } }

void test(int k /* 0..n-1 */) {

 // philosopher i calls test(i) to check whether she can eat.

 // philosopher i calls test ((n + i - 1) % n) when she is finished eating to

// unblock a hungry left neighbor.

 // philosopher i calls test((i + 1) % n) when she is finished eating to

// unblock a hungry right neighbor.

 if ((state[k] == hungry) && (state[(k+n-1) % n)] != eating) && (state[(k+1) % n] != eating))

{ state[k] = eating;

self[k].V(); // unblock philosopher i's neighbor, or guarantee } // that philosopher i will not block on
self[i].P().

}
```

	P1	P2	P3	P4	P5
State	E	T	E	T	T
Semaphore	0	0	0	0	0

Suppose one process comes, P1

When the process after completing the ***take_forks ()*** process but at the same time suddenly another process comes P3 but this process is not the neighbhor of the previous one, so this process will be ***independent*** and it will run ***take_fork().***

It State changed to Eating.

After completing the second one , third process ***P2 OR P4 OR P5*** will come and as a result it will wait until the other two process get completed.

This cause the current thread to go in ***the blocked state. Because it's neighbor state is still in the Hungry state.***

It will remain in the blocked thread until, other two process completed it's put fork method.

This causes the thread to ***Starve and Deadlock.***

3.11. Below is a proposed implementation of operations P() and V() in class BinarySemaphore. Is this implementation correct? Explain.

```
public synchronized void P()

{

while (value==0)

try { wait();

}

 Catch

 ( InterruptedException e)

{ } value=0; notify();

} public synchronized void V()

{

 while (value==1)

try

{ wait();

}

catch

(

InterruptedException e) {

 }
 value=1;
notify();
 }
```

SOLUTION :

Thus there is no critical section, so processes can’t enter into the critical section.

It will only remain in the blocked state and it will get notify according to the value .

If the value = 0 ,

Process will be in the block state and if the value is not 0 , then it will notify in the void P() method.

Elsewhere ,

In the Void V , if the value = 1 then the process value in the blocked states

Else

The value turn its to 1 and it will notify to the other process to wake up and again try for the critical section.

This implementation is not correct .