Task1 [36 Marks]

part a:[10 marks] Write a short text explaining the critical section problem: - To do this write at least two sentences for each of the following:

What is the problem about?

The critical section problem refers to the problem of how to ensure that at most one process is executing its critical section at a given time. Upon fixing the problem more problems may arise(starvation, deadlock and race conditions) and in general, in order for a critical section to actually work as expected, 3 requirements must be satisfied:

1. No mutual exclusion (no more than one task is allowed at the same time.)
2. No stuck(if some processes are trying to get into their critical section, one of them must succeed)
3. No individual starvation(if any process is trying to enter its critical section, it must eventually succeed.)

In order to help achieve the above 3 things, we also need to assume some minimal correctness properties of tasks(make some assumptions), which are covered in the lectures.

With the above mentioned, the problem of critical section is that it is very hard or even unrealistic to make true of the 3 requirements. For example. If we try to avoid race conditions(no mutual exclusion), we risk having deadlocks(stuck).

*Why is the problem relevant?

Sometimes we have to write parallel programs and when we do, we have to solve the critical section problem. Reimplementing the support for the critical section lets us understand how difficult it is so that it stops us from trying.

How did learning about this problem influenced you and your beliefs about concurrent programming?

If more than 2 threads could potentially write to a shared field we could have the data race problem.

If we fix the data race problem by writing code to mutually exclude threads from doing something, we could get deadlocks and starvations.

Previously in the assignments and projects I sort of took the easy way out by just parallelizing tasks that do not communicate with each other and this sort of makes the problem much less difficult since communication is bad in concurrent programming.

Now that we are learning what problems the communications between processes could cause and what problems could arise from solving the critical section problem, we could know better about concurrent programming.

part b:[8 marks]

Make a compact code example showing how a good and professional programmer should solve a critical section problem in Java Explain how your code behaves and why.

Public class Monitor{

conditionVar c1;

conditionVar c2;

public synchronized void method1(){

// because this method is synchronized, no other thread could be running this method at the same time.

//However, just because this thread is the only thread running this method does not mean that this thread is able to access its critical section. It may need to wait for the conditions to be met in the following while loop.

while(some things are not ready){

// the current thread will wait until some conditions are met(for example some data to be produced by other thread) in order to get out of this while loop

waitC(c1); // this is to tell the current thread to wait in a queue.

}

//so now the condition have been satisfied, the current thread can finally enter critical section!

//critical section.

notifyC(c2);// this is to signal the other thread that this thread has done some bit of work in its critical section.

}

public synchronized void method2(){

//similarly with the method1 explanations, this

while(some things are not ready){ //similar to the explanation in method1, the conditions could be some things that are done by method1. So until the thread that is doing method1 has updated some information, the while loop will not exit.

waitC(c2); //tell current thread to wait in a queue.

}

//critical section

notifyC(c1); // this is to signal the other thread that this thread has done some bit of work in its critical section.

}

}

So these 2 methods are communicating back and forth and notifying what they have done with each other and they are sort of waiting each other to have done things so that the critical section problem could be solved.

//in another class, two threads could be running the 2 methods in an infinite loop like this:

```
Thread1 implements Runnable{
@Override
Public void run(){
While(true){
m.method1();
}
}
}
Thread2 implements Runnable{
@Override
Public void run(){
While(true){
m.method2();
}
}
}

Class a{
Monitor m;

Public static void main(String[] args){
Thread1.start();
Thread2.start();
```

}

}

The above code looks more like pseudo java code but I think I have expressed the idea in this way.

part c:[18 marks] Discuss the five attempts to solve the critical section discussed in lectures. For each attempt and for each of the three properties, write at least two sentences justifying why that property holds or not (that is, 5 section with three subsections each).

Essentially I think part of the purpose of the assignment is to help us understand race conditions, deadlocks and starvations. My understanding towards the difference between starvation and deadlock was that starvation can be prevented by aging and in deadlock the resources are kind of blocked.

Attempt 1:

Mutual exclusion: yes. If a thread enters its critical section, other threads cannot enter their critical section at the same time. In this attempt turn is 1 in default and in the while loop turn can only be one of 1 and 2, thus only one thread can enter its critical section at the same time.

No block: yes. No matter what value turn is, one of the threads will eventually succeed in entering the critical section.

No individual starvation: no. if thread 1 runs into infinite loop while doing normal operations, this thread will starve and since this thread cannot enter its critical section, the turn is always equal to 1, stopping the other thread from entering their critical section (the other thread will never get out of the while(turn != 2) loop).

Attempt 2:

Mutual exclusion: No. If 2 threads try to get into their critical section at the same moment, both threads will enter critical section because at this stage both want1 and want2 are false.

No block: Yes. There is no moment when want2 and want1 are both true before entering while loop. After each critical section, either want1 or want2 is set to false, ensuring no block next time the pre-protocol is executed.

No individual starvation: yes. No individual starvation is ensured because unlike attempt1 where turn is "switched" to 1 or 2 after each critical section, this time either want1 or want2 is set to false after critical section. This makes sure that even if a thread goes into infinite loop in normal operations, another thread can still go into critical section because we eventually set "want" to false after the execution of critical section.

Attempt 2b:

Mutual exclusion: No. It is going to be the same as attempt2 if the while(want2) is executed once because the pre-protocol sets want1 to false.

No block: Yes. No block is ensured because the two while loops cannot be entered simultaneously. Even if want1 is default true, after an execution of the left hand side code, want1 is set back to false, making sure that want1 and want2 cannot both be true before entering the while loops.

No individual starvation: no. Imagine initially if thread2 enters the while loop waiting for want1 to be false but thread1 enters infinite loop in normal operations. Thread2 will never get to do critical section because thread1 will never set want1 to false. Thus no individual starvation is not satisfied.

Attempt 3:

Mutual exclusion: yes. Before entering the while loop, want is always set to true. This makes sure that at least one thread is kept waiting in the while loop while another thread is doing things in the critical section and upon leaving critical section, set the want to be false in order to release the waiting thread.

No block: no. In symmetric execution, want1 and want2 are both set to true and thus two threads both go into infinite while loop and never get out.

No individual starvation: no. Since there exists block, two threads could both go into infinite loop in pre-protocol and be blocked out of critical section. So there exists individual starvation.

Attempt 4:

Mutual exclusion: yes. There is no case when the two while loops are both not entered or they are both exited. No two threads can both not enter the while loop because want1 and want2 are set to true before checking and no two threads can exit while loop in the same time because the last step of while loop always set want1 or want2 to true.

No block: yes. If two threads are trying to get into critical section and they enter their while loops respectively, there are always moments when one of the threads can exit the while loop. More specifically, in the while loops want1 and want2 are set to false and set to true continuously, making the condition of one of while loops possible to be false in some moments. Thus no block is ensured.

No individual starvation: No. It is possible that two threads execute the tasks symmetrically for infinitely long time. This is when the two threads both try to enter the critical section symmetrically and they never get out of their while loops and thus starvation.

Attempt 5:

So if 2 processes attempt to enter critical section at the same time, Dekker's will only allow one of them in, based on whose turn it is. If one process is already in the critical section, the other process will wait for the first process to exit. Dekker's algorithm is sort of like an integration of the previous attempts that satisfy all three requirements with no obvious flaw. However it is slow.

Mutual exclusion: yes. If thread 1 and thread2 symmetrically enter the while loop, only one of them can get out and go on to the critical section because in the while loop, it first checks whose turn it is. If it is not thread1's turn, thread 1 will stop wanting to enter critical section and go into another while loop and busy wait. In this way no threads can enter critical section at the same time.

No block: yes. No block because even if two threads both enter the outer while loop, one of them will eventually exit if its turn comes. This is done by putting the waiting thread into a nested while loop until the other thread finishes critical section and say it does not want anymore and make it the other thread's turn.

No individual starvation: yes. If thread 1 goes into infinite loop in normal operations, thread 2 can still enter critical section. And infinitely many symmetric executions will not cause individual starvation because at the same time, it is only one of the thread's turn. If other threads try to exit they will be put into the while(turn) loop until their turn comes.

Task 2 [32 Marks] Consider the following code which uses the same assumptions and conventions as the code we have seen in the lecture slides. Discuss which of the three properties holds for this code. For each of the three properties, write at least 3 sentences justifying why it holds /does not hold. You are encouraged to label different parts of code and provide examples of possible executions.

```
static volatile int want1 =0;
  static volatile int want2 =0;
//first worker
  ...while(true){
  //normal operations, it can loop forever
  if(want2==-1)//pre-protocol
    {want1=-1;}//pre-protocol
  else{want1=1;}//pre-protocol
  while(want2==want1){}//pre-protocol
  try{
    //critical section operations,
    // termination assured
    }
  finally{want1=0;}//post-protocol
  }
```

```
//second worker
  ...while(true){
  //normal operations, it can loop forever
  if(want1==-1)//pre-protocol
    {want2=1;}//pre-protocol
  else{want2=-1;}//pre-protocol
  while(want1==-want2){}//pre-protocol
  try{
    //critical section operations,
    // termination assured
    }
  finally{want2=0;}//post-protocol
  }
```

all states of want1 and want2 before entering while loop:

1. Want1 = 0 and want2 = 1
2. Want1 = 0 and want2 = -1. This is when initially only thread 2 tries to enter. Pre-protocol sets want2 to -1.
3. Want1 = 0 and want2 = 0
4. Want1 = -1 and want2 = -1. After thread2 has set want2 = -1, there is a moment when thread 1 can set want1 = -1.

5. Want1 = -1 and want2 = 1.
6. Want1 = -1 and want2 = 0.
7. Want1 = 1 and want2 = -1. This is when initially thread 1 and thread 2 both try to enter. Pre-protocols set want1 = 1 and want2 = -1.
8. Want1 = 1 and want2 = 0. This is, for example initially, only thread 1 tries to enter and the pre-protocol sets want1 to 1 and want2 remains 0.
9. Want1 = 1 and want2 = 1.

Mutual exclusion: yes. In order for 2 threads to both enter critical section at the same time, the want1 and want2 values should satisfy 2 conditions at the same time: 1. (want2 != want1) 2. (want1 != -want2).

Using the listed states above, we can see that the only possible combinations of want1 and want2 are:

1. Want1 = 0, want2 = 1
2. Want1 = 1, want2 = 0
3. Want1 = -1, want2 = 0
4. Want1 = 0, want2 = -1

None of above can happen since if two threads are symmetrically executed, and they both reach the last pre-protocol, which is “while(blabla)”, at this stage neither want1 nor want2 can be 0 anymore.

So yes mutual exclusion is ensured.

No block: yes!

Using the listed states, possible combinations that could cause block are:

1. Want1 = want2 = 0;

This is also unlikely because as illustrated in the mutual exclusion part, if two threads are symmetrically executing, want1 and want2 can neither be 0 before they check condition of while loop.

No individual starvation: yes. In the case of one of the threads go into infinite loop in normal operation, the other thread can never get into the while loop in pre-protocol because the condition will never be satisfied (want1 or want2 will always be set to 0 after a thread has done critical section.).

In the case of symmetric execution, there is no way that 2 threads could both be blocked out of critical section, one of the thread will enter.

Thus no individual starvation is satisfied.

Task 3 [32 Marks] Consider the following code which uses the same assumptions and conventions as the code we have seen in the lecture slides. Discuss which of the three properties holds for this code. For each of the three properties, write at least 3 sentences justifying why it holds /does not hold. You are encouraged to label different parts of code and provide examples of possible executions.

```
static volatile boolean want1 =false;
static volatile boolean want2 =false;
static volatile int turn =1;
//first worker
...while(true){
//normal operations, it can loop forever
want1=true;
if(want2){//pre-protocol
  if(turn==2){
    want1=false;
    while(turn!=1){}//pre-protocol
    want1=true;
  }
  while(want2){}//pre-protocol
}//pre-protocol
try{
  //critical section operations,
  // termination assured
  }
finally{//post-protocol
  want1=false;//post-protocol
  turn=2;//post-protocol
  }//post-protocol
}
```

```
//second worker
...while(true){
//normal operations, it can loop forever
want2=true;
if(want1){//pre-protocol
  if(turn==1){
    want2=false;
    while(turn!=2){}//pre-protocol
    want2=true;
  }
  while(want1){}//pre-protocol
}//pre-protocol
try{
  //critical section operations,
  // termination assured
  }
finally{//post-protocol
  want2=false;//post-protocol
  turn=1;//post-protocol
  }//post-protocol
}
}
```

So this looks like a variation of Dekker's algorithm with the outer while loop changed into an if. A while(want) statement is added in the bottom of the outer if statement.

Mutual exclusion: yes.

Suppose thread 1 and thread 2 are symmetrically executing the block of code. Both want1 and want2 are set to true at the same time and thus they both enter the outer if. Since turn can only be 1 or 2, at this stage only one of the 2 threads go into the inner if.

Assume the turn is 1.

Thread 1 goes into the inner if. It goes into the while loop if turn is not 2 and the thread cannot exit the while loop until the other thread finishes its work and set the turn to 2.

Thread2 does not go into the inner loop. Instead, it skips the while loop and do the critical section operations and finally set turn to 2 and want2 to false. Thread 1 can exit its while loop(stop waiting for thread 2) only after thread2 has set turn to 2.

In this way only one thread can do critical section and the other thread just waits for it to finish. Thus mutual exclusion is ensured.

No block: yes.

The only “blocking” that can happen in this implementation is when both threads go into the “while loop” symmetrically. Since the while loop is wrapped inside an if(turn == bla) and the if statement can only be entered by one thread at the same time (turn can only be 1 or 2), no block is ensured.

No individual starvation: yes. In the case of one of the threads go into infinite loop, thread1 will never set want1 = true, thus thread 2 can skip the outer if statement and go straight into critical section.

In the case of symmetric execution, only one of the threads can enter the inner if and thus there is no change the two threads both enter infinite while loop in pre-protocol.

Thus no individual starvation is ensured.