Virginia Tech
ECE Department
Name(s): Navneet Shankar
Hokie Id(s): 906719000
email(s): navneets099@vt.edu

Fall 2024
Homework 1 Part - 1

5510 Computer Vision

Answer to Question 1

In the given program we have two processes P0 and P1 and 3 flags,$a = 1, b = 1$ and $k = 0$. P0 changes a to 1 and then waits in a while loop as long as k is 1. In P1, the inverse happens where b is set to 0, and the process waits in a while loop as long as k is 0. In P0 if the condition for k fails the Critical Section executes a is set to 1 again and in P1 b is set to 1. The two processes in this program as mutually exclusive of each other, i.e when one of the processes are executing their critical section the other process is locked out in the while loop, and can only enter the while loop when the 1st process completes the critical section and sets the flag back to 1.

Proof of Mutual Exclusion:
We start with the assumption that both threads are in Critical Section simultaneously :

If P0 is in CS , it would have broken out of the innermost while loop when $b \neq 0$, after which k is set to 0 , this would break the middle while loop of P1.

When PI enters the CS both the inner loops break, $=> a = 1, k = 1$, but here k becomes 1 when it is simultaneously 0, this is not possible since the same shared variable cannot have multiple values, therefore the two processes are mutually exclusive .

Answer to Question 2

The given lock emulates a queue at a restaurant where Tokens are distributed.When any process/thread calls the Lock it is assigned a Token in form of myTurn , through AtomicFetchAndIncrement . Every time the Lock is called, this token gets incremented. Through this way multiple threads would be waiting in the lock until the while loop condition breaks. 'nowServing' is a variable that holds the token number of the Thread/Process that has the right to enter the Critical Section, i.e when nowServing = myTurn. Once a process exits the Critical Section, nowserving gets incremented, so that the next thread can enter the CS. AtomicFetchAndIncrement occurs in a single clock cycle/instruction, therefore even if a race condition occurs the lock ensures that two threads will never have the same myTurn value, and a sequential order is always maintained by the threads in FIFO order, therefore at no point will any thread be permanently in the lock while other threads enter the CS as threads with a higher myTurn will only enter afterwards. Therefore this lock is starvation free.

The Doorway section of the program is at the lock , where myTurn is assigned , all threads get assigned their token when the Lock is called at this line of the program..

```
myTurn = AtomicFetchAndIncrement ( nextTurn );
```

The Waiting Section is in the while loop inside the Lock , the threads are locked in an infinite while loop until the nowServing gets the value of the respective myTurn,

```
while ( myTurn != nowServing ) { };
```

These two sections are segregated because the assignment of Tokens is a bounded execution while the waiting section is unbounded. FIFO order is maintained as by segregating them, the algorithm ensures threads enter the CS only according to the order of myTurn. It also enables multiple threads to acquire a lock, as if they were together the threads would have to wait.

Answer to Question 3

(a) For a thread to enter the critical section $x = i, y = 1$ are both necessary conditions. When 2 threads enter simultaneously , one thread will end up setting x last. The other thread will enter the critical section as both the while loop conditions get broken, while the last thread will be in the second if loop from line number 8. The last thread resets y to 0 while the first one will enter the CS,thus mutual exclusion is guaranteed as only one thread is allowed in the Critical Section at a time. Since one thread is always going to set x later than the other , there is no possibility of a deadlock since at least one thread will make progress as if one is stuck the other resets x and y which causes the condition to break.

(b) Starvation freedom is not satisfied in this algorithm, if the first thread sets x=1, y=1 , it will enter the CS, while the other will be in the first if loop. When the CS is complete , y are reset to 0 and the 2nd thread exits the wait and sets x=2. But the 1st thread still resets x =0, in that time the 2nd thread will find x!=2 and reset y=0 , thread 1 enters again repeating the same cycle. So one thread my never enter the Critical Section.

(c) The Algorithm does not guarantee deadlock-freedom for three threads. Lets say Thread 1 sets x=1, thread 2 sets x=2 thread 3 sets x=3 and goes past the first if condition and sets y =1, threads 1 and 2 are stuck at line 4 and thread 3 will be stuck at line 8 in the second if condition, thus causing a deadlock.

(d)The algorithm maintains mutual exclusion for three threads similar to the two thread case, If three threads call the lock only one will set y=1 and proceed to CS, other threads will exit the first if condition and reset y=0, ensuring that only one thread is in Critical Section at a time.

Answer to Question 4

For the first modification of line 15, if we use a less than comparison alone:

```
while ((k! = i)( flag[k] && (label[k] < label[i]))) {};
```

The above would violate mutual exclusion, if the two threads get the same label value they can enter the CS at the same time.In the original case there would be further selection based on the Thread IDs.

If we utilize a less than or equal to condition :

```
while ((k! = i)( flag[k] && (label[k] <= label[i]))) {};
```

The algorithm would not be deadlock free in this case, when multiple threads with the same label call the lock, all the threads will have to wait in the while loop indefinitely , when usually one of these threads based on thread id would exit into the Critical Section.

Answer to Question 5

In Figure 2 , mutual exclusion is not satisfied as both threads can enter the critical section simultaneously if both have set flag to false and then true with a very short delay.

Deadlock freedom is satisfied as during the random delay period , the other thread can reset the flag after the Critical section .

Starvation freedom is not satisfied as the random delay can cause one thread to never enter the critical section.

In figure 3 Mutual Exclusion is satisfied as only one thread can set turn at last.

It has deadlock freedom as well since the turn variable is always set to the other threads value when a thread exits the CS, so two threads will never be indefinitely in the waiting section.

Starvation freedom on the other hand is not satisfied as if one thread keeps resetting turn before the other can enter the Critical Section due to a constant delay between the threads, the other thread will never enter the Critical Section.

Answer to Question 6

(a) Petersons algorithm cannot be trivially modified to make it reentrant, when the Peterson lock is called the current thread is made a victim and the other thread is released into the Critical Section, there is no way of knowing whicih thread acquired the lock previously.

(b) The Bakery algorithm cannot be made reentrant as it relies on a labeling system which follows a hierarchy based on label number and thread Id, and one Thread ID cannot have multiple Labels .

(c) The Filter Lock cannot be trivially modified into a reentrant Lock , as it cannot map multiple Entries by the same thread ID.

(d) The Above lock (fig 3) algorithm can be modified by adding a condition to check Flag[i] which gives an indication of whether the Thread acquired the lock or not.

Answer to Programming Comparison Questions

FILTER Task :edu.vt.ece.Test2.main()
Average time per thread is 4ms
Average time per thread is 1ms
BAKERY Task :edu.vt.ece.Test2.main()
Average time per thread is 29ms
Average time per thread is 20ms