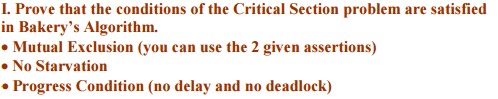
CS715--Homework#3—YUQIAN ZHANG



Solution:

Based on the original code:

While(true) {

choosing[i] = 1;

number[i] = 1 + max(number[1], …, number[N]);

choosing[i] = 0;

for (int j = 1; j <= N; j++) {

while (choosing[j] = 1) {}

while(number[j] <>0 and (number[j],j) < (number[i], i)) {}

}

Critical Section

number[i] = 0;

Remainder

}

Assertion 1: if processes i and k are in the bakery, and i entered the bakery before k entered the doorway, then number[i] < number[k] (1)

Assertion 2: if process i executes its Critical Section, and process k is in the bakery (k !=i) then ( number[i], i ) < (number[k], k ) (2)

1. Mutual Exclusion:

Assume a process Pi is already in the CS. And Pk attempts to enter CS. Although Pk has no problem entering the doorway code, it will be blocked in the bakery as following reasons:

1. If there is other Process just arriving in the doorway and be in computing its number[i], Pk will busy waiting in the first while loop.
2. If the first while loop is not satisfied, the second while loop will block Pk anyway. Because of the Assertion 1&2: number [i]<number[k] and when Pi is in CS, Pk in bakery,( number[i],i) < (number[k],k). The blocking conditions satisfied and will keep blocking Pk until Pi in CS gets out to the exiting code. At that time, the condition will change and allow Pk break out of busy waiting and go into CS.

This means that Pk cannot catch-up with Pi in CS. Therefore Mutual Exclusion satisfied.

1. No Starvation:

P1: in the CS, number[1]=1;

P2: got number[2]=2 and blocking in the second while loop,

waits for breaking condition: number[1]=0 or number[1]>=number[2]

P3: got number[3]=3 and blocking in the second while loop,

waits for number[1] & number[2] to break condition

P1: gets out of CS and number[1]sets to 0

Let’s say P2 does not check in time before P3 checks. Thus, P3 breaks the condition first and go into CS ahead of P2. Here are several situations:

* 1. If after number[1] sets to 0, P1 is still in the Remainder section, it’s fine. P2 can still check it and break out of while loop anyway.
  2. If P1 already gone through its remainder and go back to the top of while true loop and reset its value. P2 will break its blocking condition [ (number[j],j) < (number[i], i)) ] anyway. Because at that time the new value of number[1] will be the greatest among all current values. This will break P2’s busy waiting condition anyway.

Thus, no process will be postponed infinitely long.

1. No delay and no deadlock

No delay: If CS is empty and there are processes that attempt to enter CS, one of these process will be able to use the CS right away.

Proof: Keep a process in the remainder, let's say P1, so number[1] = 0 is always satisfied.

P2,P3,P4 attempt to enter CS. They can use CS as many times it needs. Because number[1] =0 is true, and P2 P3 P4 can go into CS mutual exclusively many times.

No deadlock: decision as to which process will use the CS next is taken in finite time.

Proof: process cannot block in the entry section while the CS is empty. Let’s say P1 is in CS, and P2,P3,P4 is blocking in the entry section. So if CS is empty, number[1] will immediately set to 0, which will unblock waiting process in the entry section at once. Thus. When CS is empty, process cannot block in the entry section.



Solution:

The function of choosing[i] array helps to check if a Process is in computation or let’s say when the Process is done with computation. It checks the P[i]’s computation status. It eliminates it, Mutual Exclusion will be violated.

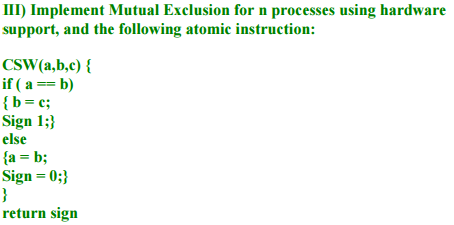
Sequence: P1,P2,P3 computing at the same time referencing the same minimum value 0

P1:number[1]=1

P2:number[2]=1

P3:number[3]=1

Let’s say P1 in CS at first. And P2 attempts to enter CS, because number[1]=number[2]=1, which breaks the blocking condition. So, P2 will also be able to enter CS, with P1 is in it now. Thus, mutual exclusion is violated if we eliminates choosing[] array.



Solution:

Using the same concept of original Test&Test and Set:

Original set: a=1;b=0;c=0;sign=0;

While( (sign)||csw(a,b,c)){}

CS

a=1;

sign=0;

}

Implementing Mutual Exclusion for n processes using above hardware support:

P1: attempts to enter CS, since sign=0, go to check the second condition: csw(a,b,c). Because a!=b, thus a=b=0 and sign=0. P1 breaks out of while loop and go into CS successfully.

At that time all shared variable conditions are: a=0;b=0;c=0;sign=0

P2: attempts to enter CS as well. When checking the first condition in while loop, it continues since sign=0. And then continue to check the second condition: csw(a,b,c). Because a==b, so b=c=0; sign=1, return 1(true), P2 now in busy waiting.

At that time all shared variable conditions are: a=0;b=0;c=0;sign=1

P3: attempts to enter CS at the same time. When checking the while loop condition, it blocks due to the first while loop condition since sign=1.

At that time all shared variable conditions are: a=0;b=0;c=0;sign=1

Thus, following processes want to access Critical Section will be block in the same way until P1 (or currently CS using Process) gets out of CS and entering exiting code to reassign the values.

P1: go out of CS entering exiting code: reassign values: a=1;sign=0. Release Processes blocking on those conditions.

P2: originally blocks on csw(a,b,c) now gets out of it and get the chance to enter the CS because of a=1;sign=0. When it is in the CS, all shared variable conditions are: a=0;b=0;c=0;sign=0 again. This make sure P2 will be in the CS mutual exclusively, other processes are still blocked.

Thus, when every time a process released from CS, it will have a=1, sign=0 reassigned, and gives opportunity for others busy waiting process to check.