CS4231
Parallel and Distributed Algorithms

Solution for Homework 1

Instructor: Haifeng YU

Homework Assignment

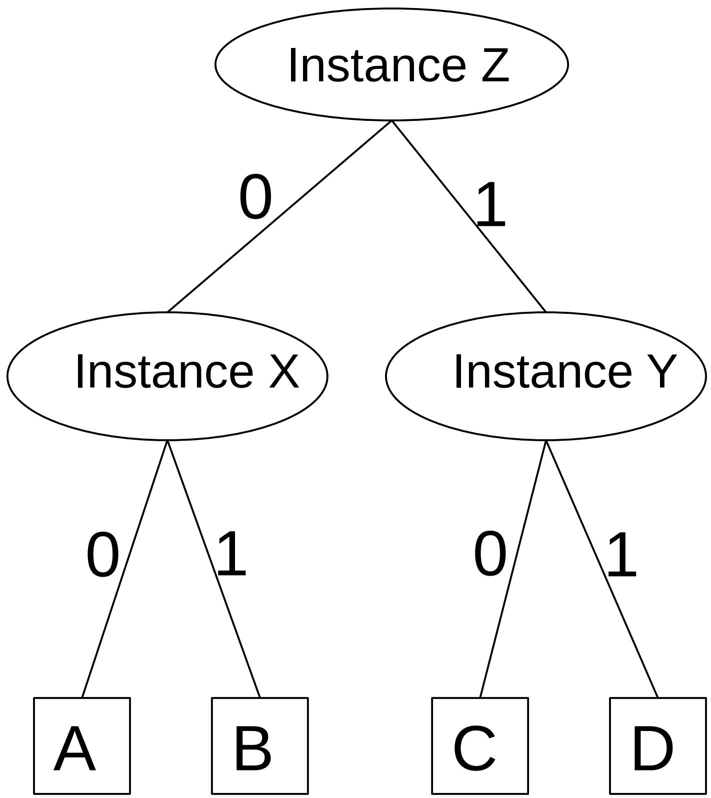
- Devise a mutual exclusion algorithm for *n* processes by using Peterson's 2-mutual-exclusion algorithm

- Page 28:
  - Problem 2.1 (clearly write out a scenario for 2 processes where problem occur as on slide 13)
  - Problem 2.3 (same as above)
  - Problem 2.4 (either give a proof or construct a problematic scenario as above. Do this for all three properties.)

n-process mutual exclusion

- Construct any binary tree (can be viewed as a tournament tree) with n leaves and where each tree node has either 2 or 0 children.
- Each intermediate tree nodes is a logical critical section and runs an instance of Peterson's algorithm
- To enter the critical section, a process needs to enter all the logical critical sections on the path from itself to the tree root.
- Mutual exclusion, progress, and no starvation can then be easily proved…

Need to Release with Proper Ordering



process A	*process B*
X.RequestCS(0){…}	
Z.RequestCS(0){…}	
	X.RequestCS(1) { …
X.ReleaseCS(0){…}	
	… }
	Z.RequestCS(0){ …
Z.ReleaseCS(0){…}	
	… }

Z.RequestCS(0) invoked a 2nd time without anyone invoking Z.ReleaseCS(0)!

2.1(a)

- Initially
  - wantCS[0] = false
  - wantCS[1] = false
- Violates mutual exclusion

process 0	*process 1*
wantCS[0] = true	
turn = 0	
while (wantCS[1] == true && turn == 1) {}	
	wantCS[1] = true
	turn = 1
	while (wantCS[0] == true && turn == 0) {}

2.1(b)

- Initially
  - wantCS[0] = false
  - wantCS[1] = false
- Violates mutual exclusion

process 0	*process 1*
	turn = 0
turn = 1	
wantCS[0] = true	
while (wantCS[1] == true && turn == 1) {}	
	wantCS[1] = true
	while (wantCS[0] == true && turn == 0) {}

2.3

- Initially
  - number[0] = number[1] = 0
- Example for two processes: Violates mutual exclusion

process 0	*process 1*
	if (number[0]>number[1]) number[1] = number[0];
if (number[1]>number[0]) number[0] = number[1];	
	number[1]++; (1)
	while (number[0] != 0 && Smaller(number[0], 0, number[1], 1))
number[0]++; (1)	
while (number[1] != 0 && Smaller(number[1], 1, number[0], 0))	

Dekker's algorithm – Mutual Exclusion

P0

```
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
    }}
CRITICAL SECTION
turn = 1
wantCS[0] = false
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
    }}
CRITICAL SECTION
turn = 0
wantCS[1] = false
```

Prove by contradiction.

Case 1: turn = 0 when P0 and P1 are in critical section.

Then P1 last executed "turn = 0" after P0 last executed "turn = 1". Hence P1 must have seen turn = 0 when it checked turn's value. P1 must have seen wantCS[0] = false when it checks wantCS[0]'s value, since otherwise with turn = 0, P1 will get stuck and can never get into the critical section. Hence P1 must have executed "while (wantCS[0])" before P0 executes its first statement. Then P0 will see wantCS[1] being true, and will not pass its outer while loop and will not enter the critical section.

Dekker's algorithm – Mutual Exclusion

P0

```
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
    }}
CRITICAL SECTION
turn = 1
wantCS[0] = false
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
    }}
CRITICAL SECTION
turn = 0
wantCS[1] = false
```

Prove by contradiction.

Case 2: turn = 1 when P0 and P1 are in critical section.

Symmetric…complete yourself…

Dekker's algorithm – Progress

P0

```
wantCS[0] = true;
while (wantCS[1]) {
   if (turn == 1) {
      wantCS[0] = false;
      while (turn == 1) ;
      wantCS[0] = true;
}}
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
   if (turn == 0) {
      wantCS[1] = false;
      while (turn == 0) ;
      wantCS[1] = true;
}}
```

Prove by contradiction.

Case 1: turn = 0 when neither of them can enter critical section (will the value of turn change?).

On P1: wantCS[1] will become and remain false since turn = 0.

So P0 will enter critical section after wantCS[1] becomes false. Contradiction.

Case 2: turn = 1 when neither of them can enter critical section. Symmetric…

Dekker's algorithm – No starvation

P0

```
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
    }}
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
    }}
```

Prove by contradiction: Suppose P0 (the case for P1 is similar) tries to enter CS and is blocked forever starting from time T1. P0 never modifies turn after T1, while P1 can only set turn to be 0 (when it exits the critical section).

Case 1: turn is always 0 (i.e., 0 at T1 and later potentially set to 0 again by P1).

Case 2: turn is always 1 (i.e., 1 at T1 and never set to 0 by P1)

Case 3: turn is 1 at T1, and later 0 (set by P1) forever.

In any of these cases, the value of turn "stabilizes" eventually (i.e., will not flip-flop forever). Hence we only need to consider fixed value for turn.

Dekker's algorithm – No starvation

Case 1 and 3:
Eventually turn = 0.

Let T2 (where T2 ≥ T1) be the time starting from which turn is always 0.

P0

```
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
}}
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
}}
```

On P0: With P0 being blocked and with turn always being 0 since T2, wantCS[0] will remain true forever since some time T3.

There can be many choices for T3 -- we choose a T3 such that T3 ≥ T2.

Dekker's algorithm – No starvation

Case 1 and 3:
Eventually turn = 0.

From previous slide:
turn = 0 and
wantCS[0] = true
forever since T3.

P0

```
wantCS[0] = true;
while (wantCS[1]) {
  if (turn == 1) {
    wantCS[0] = false;
    while (turn == 1) ;
    wantCS[0] = true;
}}
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
  if (turn == 0) {
    wantCS[1] = false;
    while (turn == 0) ;
    wantCS[1] = true;
}}
```

We want to show that wantCS[1] = false forever since some time T4 ≥ T3.

We check where P1 is at time T3.

- Case 1: At T3, P1 is somewhere in the above code segment. Then P1 must reach "while (turn == 0)" some time after T3, and the claim must hold at that point of time.
- Case 2: At T3, P1 is not in the above code segment, but will enter the above code some time in the future. Proof is the same as the previous case.
- Case 3: At T3, P1 is not in the above code segment, and will never enter it either. Then either wantCS[1] is false at T3, or wantCS[1] will be changed to false when P1 releases CS at time T4. Furthermore, wantCS[1] will remain false after that.

Dekker's algorithm – No starvation

Case 1 and 3:
Eventually turn = 0.

From previous slide:
turn = 0 and
wantCS[0] = true and
wantCS[1] = false
forever since T4.

P0

```
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
}}
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
}}
```

We know that P0 is still blocked at time T4, where T4 ≥ T3 ≥ T2 ≥ T1.

Regardless of where P0 is at time T4, given that turn = 0 and wantCS[1] = false, P0 will no longer block. Contradiction.

We are done with Case 1 and 3, next move on to Case 2…

P0 | P1

Case 2: turn is always 1 since T1.

Let T2 (where T2 < T1) to be the time when turn was last assigned a value of 1 --- by P0 or by initialization. Now the value of turn is always 1 since T2.

If T2 was initialization time…

```
P0:
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
}}
CRITICAL SECTION
turn = 1;
wantCS[0] = false;
```

```
P1:
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
}}
CRITICAL SECTION
turn = 0;
wantCS[1] = false;
```

We claim that P1 never ever sets wantCS[1] to be true, since initialization time.

- Note that wantCS[1] only becomes true when P1 tries to enter the critical section. If P1 ever tries to enter, then P1 must be able to eventually enter. Otherwise since P0 is forever blocked after T1, P1's not being able to enter would violate the progress property (which we already proved). If P1 enters, P1 will eventually exit and set turn to be 0, contradicting with turn always being 1 since initialization time.

Now since wantCS[1] is always false since initialization time, P0 would not block at time T1 (where T1 > T2). Contradiction.

Case 2: turn is always 1 since T1.

Let T2 (where T2 < T1) to be the time when turn is last assigned a value of 1 --- by P0 or by initialization.

If T2 was the last time when P0 releases CS and sets turn to be 1…

P0	P1	
wantCS[0] = true;	wantCS[1] = true;	
while (wantCS[1]) {	while (wantCS[0]) {	
if (turn == 1) {	if (turn == 0) {	
wantCS[0] = false;	wantCS[1] = false;	
while (turn == 1) ;	while (turn == 0) ;	A
wantCS[0] = true;	wantCS[1] = true;	
}}	}}	
CRITICAL SECTION	CRITICAL SECTION	
turn = 1;	turn = 0;	B
wantCS[0] = false;	wantCS[1] = false;	C

We want to find out where P1 can potentially be, at time T2…

At T2, P1 must not be at A:

- Otherwise P1 will later change turn to 0, which contracts with the fact that turn is always 1 after T2.

Case 2: turn is always 1 since T1.

Let T2 (where T2 < T1) to be the time when turn is last assigned a value of 1 --- by P0 or by initialization.

If T2 was the last time when P0 releases CS and sets turn to be 1…

P0

```
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
}}
CRITICAL SECTION
turn = 1;
wantCS[0] = false;
```

P1

```
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;          A
        wantCS[1] = true;
}}
CRITICAL SECTION
turn = 0;
                                     B
wantCS[1] = false;
                                     C
```

At T2, P1 must not be at B: Prove by contradiction and assume P1 is at B at time T2.

- P0 and P1 are releasing CS at T2. Consider when they requested CS this round…
- Let S0 be when P0 passes "while(wantCS[1])": P0 sees wantCS[1] being false at time S0
- Let S1 be when P1 passes "while(wantCS[0])": P1 sees wantCS[0] being false at time S1
- If S0 $\leq$ S1 < T2: Between S0 and T2, wantCS[0] is always true. Contradict with wantCS[0] being false at S1.
- If S1 $\leq$ S0 < T2: Between S1 and T2, wantCS[1] is always true. Contradict with wantCS[1] being false at S0.

P0 | P1

Case 2: turn is always 1 since T1.

Let T2 (where T2 < T1) to be the time when turn is last assigned a value of 1 --- by P0 or by initialization.

If T2 was the last time when P0 releases CS and sets turn to be 1…

```
P0:
wantCS[0] = true;
while (wantCS[1]) {
    if (turn == 1) {
        wantCS[0] = false;
        while (turn == 1) ;
        wantCS[0] = true;
}}
CRITICAL SECTION
turn = 1;
wantCS[0] = false;

P1:
wantCS[1] = true;
while (wantCS[0]) {
    if (turn == 0) {
        wantCS[1] = false;
        while (turn == 0) ;
        wantCS[1] = true;
}}
CRITICAL SECTION
turn = 0;
wantCS[1] = false;
```

A

B

C

Hence at T2, P1 must be at C!

- In turn, at T2, wantCS[1] must be false.
- After T2, wantCS[1] will remain false forever: To see why, note that wantCS[1] only becomes true when P1 tries to enter the critical section. If P1 is trying to enter, then by the earlier progress property (which we already proved), either P0 or P1 will be able to enter. If P0 enters, it contradicts to P0 being blocked forever. If P1 enters, P1 will eventually exit and set turn to be 0, contradicting with turn always being 1.

Hence when P0 later (anytime after T2) tries to request CS, P0 will never block. This contradict with the fact that P0 is blocked forever starting from T1 where T1 > T2.