Critical Sections – Mutual exclusion

Software solutions

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Software solution: no special instructions

- The software solutions to the CS problem are based on the use of **shared (global) variables**
  - Available on systems with shared memory
- We will analyze the solution with only two P (or T)
  - They are named threads P_i (T_i) and P_j (T_j)
    - Give **i** then **j=i-1**, and vice versa
- The proposed solution is not easily extended to more than two threads

In addition, we suppose the existence of two logical values TRUE (1) and FALSE (0)

Mutual exclusion: Solution 1

- Shared variables
  - int flag[2] = {FALSE, FALSE};

P_i / T_i

```
while (TRUE) {
  while (flag[j]);
  flag[i] = TRUE;
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  while (flag[i]);
  flag[j] = TRUE;
  CS
  flag[j] = FALSE;
 non critical section
}
```

Mutual exclusion
Deadlock
Starvation
Symmetry
?

Mutual exclusion: Solution 1

- Shared variables
  - int flag[2] = {FALSE, FALSE};

P_i / T_i

```
while (TRUE) {
  while (flag[j]);
  flag[i] = TRUE;
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  while (flag[i]);
  flag[j] = TRUE;
  CS
  flag[j] = FALSE;
  non critical section
}
```

- Mutual exclusion **not** granted
  - T_i and T_j can access to their CS at the same time

Mutual exclusion: Solution 1

- Solution 1
  - A shared vector of flags "busy CS"
  - A thread tests the other thread "busy CS" flag and sets its own
- It **does not guarantee** mutual exclusion in CS
- The technique fails because
  - The lock variable is controlled and changed by two separate statements
  - A context switching may occur between the two statements (they **are not** executed as single, **atomic** instruction)

Mutual exclusion: Solution 1

- The flag "Busy CS" variable is usually named **lock** variable
  - It serves to protect the CS
- Even if the solution were correct, the cycles testing the flag is a **busy form of waiting**
  - Waste of CPU time
  - Acceptable only if the busy wait is very short
- This lock mechanism, which uses the busy form of waiting, is called **spin lock**

Mutual exclusion: Solution 2

- Shared variables
  - int flag[2] = {FALSE, FALSE};

Exchanges test and set statements

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  while (flag[j]);
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  flag[j] = TRUE;
  while (flag[i]);
  CS
  flag[j] = FALSE;
  non critical section
}
```

Mutual exclusion
Deadlock
Starvation
Symmetry
?

Mutual exclusion: Solution 2

- Shared variables
  - int flag[2] = {FALSE, FALSE};

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  while (flag[j]);
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  flag[j] = TRUE;
  while (flag[i]);
  CS
  flag[j] = FALSE;
  non critical section
}
```

- Possible **deadlock** (or better **livelock**)
  - Both threads can set their flag to TRUE, and wait forever

Mutual exclusion: Solution 2

- Solution 2 tries to solve the problem of solution 1 with a symmetric approach
  - Reserves the access to the CS before testing its availability (i.e., performs setting before testing)
  - But deadlock (livelock) is possible
  - Again, busy form of waiting with spin lock

Mutual exclusion: Solution 3

- Shared variables
  - int turn = i;

Or
int turn = j;

P_i / T_i

```
while (TRUE) {
  while (turn!=i);
  CS
  turn = j;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  while (turn!=j);
  CS
  turn = i;
  non critical section
}
```

Mutual exclusion
Deadlock
Starvation
Symmetry
?

Mutual exclusion: Solution 3

- Shared variables
  - int turn = i;

Or
int turn = j;

P_i / T_i

```
while (TRUE) {
  while (turn!=i);
  CS
  turn = j;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  while (turn!=j);
  CS
  turn = i;
  non critical section
}
```

- **Undefined** wait
  - T_i and T_j access their CS only alternatively
  - If T_i (T_j) has not interest in using its CS, P_j (P_i) cannot enter its CS (**starvation**)

Mutual exclusion: Solution 3

- Solution 3 uses
  - A binary variable "turn", which indicates that the thread is enabled to enter its CS
  - Mutual Exclusion is ensured by the assignment of the access turn
  - The solution involves alternation and possible starvation
  - Busy form of waiting with spin lock (as solutions 1 and 2)

Mutual exclusion: Solution 4

- Shared variables
  - int turn = i;
  - int flag[2] = {FALSE, FALSE};

Or
int turn = j;

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  flag[j] = TRUE;
  turn = i;
  while (flag[i] &&
    turn==i);
  CS
  flag[j] = FALSE;
  non critical section
}
```

Mutual exclusion
Deadlock
Starvation
Symmetry
?

Mutual exclusion: Solution 4

- Shared variables
  - int turn = i;
  - int flag[2] = {FALSE, FALSE};

Or
int turn = j;

Mutual exclusion?

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  CS
  flag[i] = FALSE;
  non critical section
}
```

In CS iff
flag[j]==FALSE OR turn==i

T_i and T_j both in their CSs?
No, because turn==i or turn==j,
not both

If T_j is in its CS, T_i can enter its CS?
If T_j is inside its CS, flag[j]==TRUE (set by T_j)
AND turn==j (set by T_i) ,
thus T_i will wait

Mutual exclusion: Solution 4

- Shared variables
  - int turn = i;
  - int flag[2] = {FALSE, FALSE};

Or
int turn = j;

Deadlock?

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  CS
  flag[i] = FALSE;
  non critical section
}
```

T_i/T_j wait only on this while loop

If T_i is waiting and T_j is not interested in its CS, flag[j]==FALSE, thus T_i can access its CS

T_i and T_j cannot be both waiting, because variable **turn** stores a **single value at a time**

If T_i is waiting and T_j releases its CS, T_j sets flag[j]=FALSE, thus T_i can access its CS

Mutual exclusion: Solution 4

- Shared variables
  - int turn = i;
  - int flag[2] = {FALSE, FALSE};

Or
int turn = j;

Starvation?

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  CS
  flag[i] = FALSE;
  non critical section
}
```

T_j is in its CS, and is very fast at reserving again access to its CS. Can T_i wait forever (starve)?

T_j sets flag[j] to FALSE but immediately after to TRUE. However, it sets turn=i, enabling access for T_i thus T_j will waits

Mutual exclusion: Solution 4

- Shared variables
  - int turn = i;
  - int flag[2] = {FALSE, FALSE};

Or
int turn = j;

Symmetric?

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  flag[j] = TRUE;
  turn = i;
  while (flag[i] &&
    turn==i);
  CS
  flag[j] = FALSE;
  non critical section
}
```

Symmetrically identical codes

Mutual exclusion: Solution 4

- Shared variables
  - int turn = i;
  - int flag[2] = {FALSE, FALSE};

Or
int turn = j;

Symmetric?

P_i / T_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] &&
    turn==j);
  CS
  flag[i] = FALSE;
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  flag[j] = TRUE;
  turn = i;
  while (flag[i] &&
    turn==i);
  CS
  flag[j] = FALSE;
  non critical section
}
```

- Correct solution:
  - All the conditions related to the CS are met

Mutual exclusion: Solution 4

- The first software solution that allows two or more processes to share a single-use resource without conflict, using only shared memory and normal instructions, has been proposed by G. L. Peterson [1981]
  - It guarantees
    - Mutual exclusion
    - Progress (no deadlock)
    - Defined wait (no starvation)
    - Symmetry
  - The wait of P (or T) is a **busy waiting** on a **spin lock**
    - The problem of the consumption of "CPU time" remains

Conclusions

- In general, the software solutions to the problem of CS are complex and inefficient
  - Setting and testing a variable by a P/T is an operation that is "invisible" to the other P/T
  - **Test and set operations are not atomic,** thus a P/T can react to the presumed value of a variable rather than to its current value
  - The solutions for a number n of P/T are even more complex
    - McGuire [1972]
    - Lamport [1974]