Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Synchronization Tools-Part2

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts

- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
    - Can some processes starve -- never enter their critical section.
  - What if there are two CPUs?

Software Solution 1

- **Two process solution.**
- **Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted.**
- The two processes share one variable:
  - int ***turn***;
  - ***turn*** indicates whose turn it is to enter the critical section.

Algorithm for Process P_i

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

Algorithm for P_0 and P_1

Initially turn = 0

```
while (TRUE) {
    while (turn != 0)          /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)          /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

(a) Process 0.   (b) Process 1.

Correctness of the Software Solution

- **Mutual exclusion is preserved**
  - P_i enters critical section only if:

    turn = i
  - **turn cannot be both 0 and 1 at the same time**
- It ***wastes*** CPU time
  - So we should avoid busy waiting as much as we can.
- **Can be used** only when the waiting period is expected to be **short**.

Correctness of the Software Solution (cont.)

- However there is a problem in the above approach!
  - What about the **Progress requirement**?
  - What about the **Bounded-waiting requirement**?

Correctness of the Software Solution (cont.)

- ***P_0 leaves its critical region*** and sets turn to 1, enters its non-critical region.
- ***P_1 enters its critical region***, sets turn to 0 and leaves its critical region.
- ***P_1 enters its non-critical region***, quickly finishes its job and goes back to the while loop.
- Since turn is 0, process 1 ***has to wait*** for process 0 to finish its non-critical region so that it can enter its critical region.
- This violates the **second condition (progress)** of providing mutual exclusion.

P_0

Initially turn = 0

```
while (TRUE) {
    while (turn != 0)
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

P_1

How About this solution?

```
//Algorithm for Pi
while (true){

    turn = i;
    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

Peterson’s Solution

- The previous solution solves the problem of one process blocking another process while its outside its critical section.

- Peterson’s Solution is a neat solution with busy waiting, that defines the procedures for entering and leaving the critical region.

Peterson's Solution (cont.)

- Two process solution
- Assume that the **load** and **store** machine-language instructions are **atomic**; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - boolean flag[2]
- The **variable turn indicates whose turn it is to enter the critical section.**
- The **flag array** is used to indicate **if a process is ready to enter the critical section**.
  - flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);

            /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either flag[j] = false or turn = I

  2. Progress requirement **is satisfied**

  3. Bounded-waiting requirement **is met**

Peterson’s Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution **is not guaranteed to work on modern architectures.**
  - To improve performance, **processors and/or compilers may reorder operations that have no dependencies.**
- Understanding why it **will not work is useful for better understanding race conditions.**

- **For single-threaded this is ok as the result will always be the same.**

- For multithreaded the **reordering may produce inconsistent or unexpected results**!