Peterson's Solution

Peterson's Solution

Peterson's solution is a classic software based solution to solve the race condition. It is said to be classic because this solution may not work correctly on modern day architectures.
However, it provides a good algorithmic description of race condition problem and demonstrates the complexities involved in designing software that addresses the requirement of mutual exclusion for processes.

Description of the Solution

It is restricted to two processes that alternates the execution between their critical and remainder sections. Let us call these processes Pi and Pj.

> **Note**: Consider that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

The solution requires the two processes to share following 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 Pi is ready!

Principle of the algorithm:
In this algorithm, if process Pi wants to enter the critical then it marks flag[i] as true and sets the turn variable to **j.** It indicates that it is j's turn to enter the critical section. Hence, the principle on which this algorithm is based can be understood through an analogy. For example, you and your friends want to enter the bus. You both are interested to get in, but you show humbleness and allow your friend to get in first and once they get in, then you get in the bus as well.

Algorithm for Process Pi

```
while (true)
{
  flag[i] = TRUE;
  turn = j;
  while ( flag[j] && turn == j);
  CRITICAL SECTION
  flag[i] = FALSE;
  REMAINDER SECTION
 }
```

Two Process Working Concurrently

Let us see how the code works when the processes are running simultaneously:

Process 1

```
while (1)
{
  flag1 = TRUE;
  turn = 2;
  while (flag2 && turn == 2);
  critical section.....
  flag1 = FALSE;
  remainder section.....
 }
```

Process 2

```
while (1)
{
  flag2 = TRUE;
  turn = 1;
  while (flag1 && turn == 1);
  critical section.....
  flag2 = FALSE;
  remainder section.....
}
```

Execution shown in phases:

Process 0:

flag[0] := TRUE
turn := 1
check (flag[1] = TRUE and turn = 1)
*Condition is false because flag[1] = FALSE
* Since condition is false, no waiting in while loop
* Enters the critical section

Process 1:

flag[1] := TRUE
turn := 0
check (flag[0] = TRUE and turn = 0)
*Since condition is true, it keeps busy waiting until it loses the processor
*Process 0 resumes and continues until it finishes in the critical section

Phase-2

Phase-1	
Process 0: *Leaves critical section Sets flag[0] := FALSE * Start executing the remainder (anything else a process does besides using the critical section) * Process 0 happens to lose the processor **Phase-3**	**Process 1:** check (flag[0] = TRUE and turn = 0) * This condition fails because flag[0] = FALSE * No more busy waiting *Enter the critical section **Phase-4**

Conclusion

We hope that you have understood how Peterson's solution solves the race condition and makes sure that only one process remains in the critical section. However, this is valid only for two processes, hence, for more than two processes, we have to use hardware support. In the following lecture videos, we will discuss other methods that solve the race condition problem.