## PETERSON'S SOLUTION

Peterson's solution is a classic **Software-Based Solution** to the critical-section problem.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered ***P0*** and ***P1***. Let ***Pi*** represents one process and ***Pj*** represents other processes (i.e. j = i-1)

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

    flag[i] = false;

    Remainder Section
} while (true);
```

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. At any point of time the turn value will be either 0 or 1 but not both.

- if turn == i, then process *Pi* is allowed to execute in its critical section.
- if turn == j, then process *Pj* is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section.

Example: if flag[i] is true, this value indicates that *Pi* is ready to enter its critical section.

- To enter the critical section, process *Pi* first sets **flag[i]=true** and then sets **turn=j**, thereby **Pi** checks if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at the same time. Only one of these assignments will be taken. The other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

The above code must satisfy the following requirements:

1. Mutual exclusion
2. The progress
3. The bounded-waiting

**Check for Mutual Exclusion**

- Each ***Pi*** enters its critical section only if either **flag[j] == false** or **turn == i**.
- If both processes can be executing in their critical sections at the same time, then **flag[0] == flag[1] == true**. But the value of turn can be either **0** or **1** but cannot be both.

- Hence ***P0*** and ***P1*** could not have successfully executed their **while** statements at about the same time.
- If ***Pi*** executed "**turn == j**" and the process ***Pj*** executed **flag[j]=true** then Pj will have successfully executed the while statement. Now Pj will enter into its **Critical section**.
- At this time, **flag[j] == true** and **turn == j** and this condition will persist as long as ***Pj*** is in its critical section. As a result, mutual exclusion is preserved.

**Check for Progress and Bounded-waiting**

- The while loop is the only possible way that a process ***Pi*** can be prevented from entering the critical section only if it is stuck in the while loop with the condition **flag[j] == true** and **turn == j**.
- If ***Pj*** is not ready to enter the critical section, then **flag[j] == false** and ***Pi*** can enter its critical section.
- If ***Pj*** has set **flag[j] == true** and is also executing in its while statement, then either **turn == i** or **turn == j**.
- If **turn == i**, then ***Pi*** will enter the critical section. If **turn == j**, then ***Pj*** will enter the critical section.
- Once ***Pj*** exits its critical section, it will reset **flag[j]** to false, allowing ***Pi*** to enter its critical section.
- If ***Pj*** resets **flag[j]** to true, it must also set **turn == i**. Thus, since ***Pi*** does not change the value of the variable **turn** while executing the while statement, ***Pi*** will enter the critical section (**Progress**) after at most one entry by ***Pj*** (**Bounded Waiting**).

**Problem with Peterson Solution**

There are no guarantees that Peterson's solution will work correctly on modern computer architectures perform basic machine-language instructions such as load and store.

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. This is an Non-preemptive kernel approach.
- We could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable.

This solution is not as feasible in a multiprocessor environment.

- Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors.
- This message passing delays entry into each critical section and system efficiency decreases and if the clock is kept updated by interrupts there will be an effect on a system's clock.