**PETERSON’S SOLUTION**

A classic software-based solution to the critical-section problem known as Peterson's solution. It may not work correctly on modern computer architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and this solution guarantees that it provides mutual exclusion, bounded waiting, and progress of the processes. Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Pi and Pj.

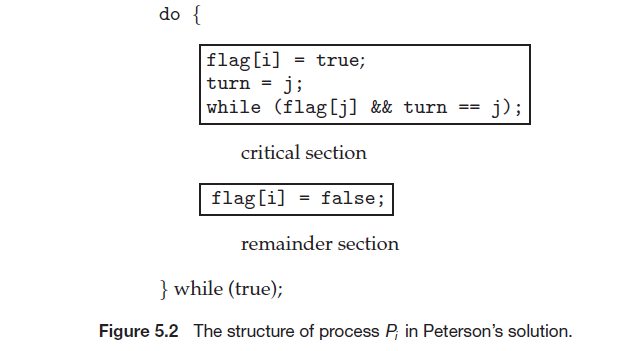
Peterson's solution requires two data items to be shared between the two processes:

int turn;

Boolean flag [2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi 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. For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section.

The general structure of process Pi in Peterson’s algorithm description is



The following code segments shows how Peterson's algorithm works:

**Process Pi:**

do

            {

               flag[i]=true;

               turn=j;

               while(flag[j]&&turn==j);

               critical section

               flag[i]=false;

               remainder section

            }while(true);

**Process Pj:**

do

            {

               flag[j]=true;

               turn=i;

               while(flag[i]&&turn==i);

               critical section

               flag[j]=false;

               remainder section

            }while(true);

We now prove that this solution is correct. We need to show that:

 1. Mutual exclusion is preserved.

2. The progress requirement is satisfied.

 3. The bounded-waiting requirement is met.

To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes—say, Pj—must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“turn == j”). However, at that 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.

To prove properties 2 and 3,we note 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; this loop is the only one possible. 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] to 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. However, 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 to 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).