Peterson’s Solution

Peterson’s Solution

It is restricted to two processes that alternates the execution between their critical and remainder sections.
Consider that the LOAD and STORE 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 Pi is ready!
Note:- Peterson’s Solution is a software based solution.

Algorithm for Process Pi

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

do
{
    acquire lock
            Critical Section
    release lock
            Remainder Section
```

```
}
while (True);
```

Note:-Race Conditions are prevented by protecting the critical region by the locks.

Two Process Working Concurrently

Process 1

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

Process 2

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

Shared Variables }--- flag1, flag2, turn

Example

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

Phase-1

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

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**

Implementation

```
public class cSection {
int turn;
boolean flag[] = new boolean[2];  int i = 0, j = 1;
// CSC variables
int counter = 0;// counter for giving processes an upper bound
int cscVar = 13;

private class ProcessI extends Thread { // process thread for i
@Override
public void run() {
try {
do {
flag[i] = true;  turn = j;
while (flag[j] && turn == j)
; // wait for j to finish
// critical section
System.out.println("I is in critical section");  cscVar++;
System.out.println(cscVar);  counter++;
System.out.println("counter is " + counter + "n      ");
//
flag[i] = false;
// remainder section
```

```
} while (counter < 100); // 100 is upper bound, to remove
// infinite looping
}
catch (Exception ex) {  ex.printStackTrace();
}
}
}

private class ProcessJ extends Thread { // process thread for j
@Override
public void run() {
try {
do {
flag[j] = true;  turn = i;
while (flag[i] && turn == i)
;// wait for i to finish
// critical section
System.out.println("J is in critical section");  cscVar--;
System.out.println(cscVar);  counter++;
System.out.println("counter is " + counter + "n      ");
//
flag[j] = false;
// remainder section
} while (counter < 100); // 100 is upper bound, to remove
// infinite looping
}
catch (Exception ex) {  ex.printStackTrace();
}
}
}
public cSection() {
System.out.println("Starting Threads/Processes");  Thread I = new ProcessI();
Thread J = new ProcessJ();  I.start(); // start process i  J.start(); // start process j
}
public static void main(String[] args) {  cSection cSec = new cSection();
}
}
```