Department of Computer Science and Engineering

FACULTY OF ENGINEERING AND TECHNOLOGY
UNIVERSITY OF LUCKNOW
LUCKNOW

Dr. Zeeshan Ali Siddiqui
Assistant Professor
Deptt. of C.S.E.

PROCESS SYNCHRONIZATION

(Peterson's Solution for The Critical-Section Problem)

Peterson's Solution 1/3

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

- For convenience, lets represent one *process* as Pi and other one as Pj, here j equals 1 − i.

Peterson's Solution 2/3

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

  - int turn;
    - 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.

  - boolean flag[2];
    - 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.

Peterson's Solution 3/3

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);
```

Peterson's Solution: Analysis 1/3

- To 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*

Peterson's Solution: Analysis 2/3

- 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.

Peterson's Solution: Analysis 3/3

- 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*).

References

1. Silberschatz, Galvin and Gagne, “Operating Systems Concepts”, Wiley.

2. William Stallings, “Operating Systems: Internals and Design Principles”, 6th Edition, Pearson Education.

3. D M Dhamdhere, “Operating Systems: A Concept based Approach”, 2nd Edition, TMH.

Thank You.