

Peterson's Solution

Peterson's Solution

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P_0 and P_1 . For convenience P_i and P_j , j equals $1 - i$.
- Assume that the **load** and **store** machine-language 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 P_i is ready.

Peterson's Solution-Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
  
        remainder section  
} while (true);
```

Peterson's Solution

- To enter the critical section, process P_i first sets $flag[i]$ to be true and then sets turn to the value j , so that 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 roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The final value of turn determines which of the two processes is allowed to enter its critical section first.

Peterson's Solution

1. Mutual exclusion is preserved.

- *P_i enters its critical section only if either $\text{flag}[j] == \text{false}$ or $\text{turn} == i$.*
- If both processes can be executing in their critical sections at the same time, then $\text{flag}[0] == \text{flag}[1] == \text{true}$.
- *P_0 and P_1 could not have successfully executed their while statements at about the same time, since value of turn can be either 0 or 1 but cannot be both.*
- One of the processes —say, P_j —*must have successfully executed the while statement, whereas P_i had to execute at least one additional statement (“ $\text{turn} == j$ ”).*
- However, at that time, $\text{flag}[j] == \text{true}$ and $\text{turn} == j$, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

Peterson's Solution

2. The progress requirement is satisfied.

3. The bounded-waiting requirement is met.

- A process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition $\text{flag}[j] == \text{true}$ and $\text{turn} == j$; this loop is the only one possible.
- If P_j is not ready to enter the critical section, then $\text{flag}[j] == \text{false}$, and P_i can enter its critical section.
- If P_j has set $\text{flag}[j]$ to true and is also executing in its while statement, then either $\text{turn} == i$ or $\text{turn} == j$. If $\text{turn} == i$, then P_i will enter the critical section.

Peterson's Solution

- If $\text{turn} == j$, then P_j will enter the critical section.
- Once P_j exits its critical section, it will reset $\text{flag}[j]$ to false, allowing P_i to enter its critical section. If P_j resets $\text{flag}[j]$ to true, it must also set turn to i .
- Since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Mutex Locks

Mutex Locks

- Used to protect critical regions and thus prevent race conditions.
- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock.

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Solution to the critical-section problem using mutex locks.

Mutex Locks

- The definition of acquire() is as follows:

```
acquire()
{
    while (!available); /* busy wait */
    available = false;
}
```

- The definition of release() is as follows:

```
release()
{
    available = true;
}
```

Mutex Locks

- Calls to either acquire() or release() must be performed atomically.

Disadvantage

- Busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- This type of mutex lock is also called a **spinlock** because the process “spins” while waiting for the lock to become available.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.