Peterson's Solution

- Two process solution
- 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

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

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

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

Algorithm for Process P_j

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

- Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Disadv:

Holds good only for 2 processes

Synchronization Hardware

- Many systems provide hardware support for critical section code

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while ( TestAndSet (&lock ))
            ;   // do nothing

            //    critical section

    lock = FALSE;

            //      remainder section

} while (TRUE);
```

- Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

The Test and Set solution does not satisfy bounded waiting. There is no inherent mechanism in the solution to limit the waiting time for a process trying to acquire the lock.

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section

        lock = FALSE;

                //     remainder section

} while (TRUE);
```

```
void Swap (boolean *a, boolean *b)
        {
                boolean temp = *a;
                *a = *b;
                *b = temp:
        }
```

Bounded-waiting Mutual Exclusion with TestandSet()

every process should get its turn eventually, and no process should be starved indefinitely.

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
            key = TestAndSet(&lock);
    waiting[i] = FALSE;
            // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
            j = (j + 1) % n;
    if (j == i)
            lock = FALSE;
    else
            waiting[j] = FALSE;
            // remainder section
} while (TRUE);
```


```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
            key = TestAndSet(&lock);
    waiting[i] = FALSE;
            // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
            j = (j + 1) % n;
    if (j == i)
            lock = FALSE;
    else
            waiting[j] = FALSE;
            // remainder section
} while (TRUE);
```

- Semaphores are versatile and can be used to control access to multiple resources, not just critical sections. This makes them suitable for scenarios where coordination and synchronization involve more than just protecting shared data.

  Semaphores can be used as counting mechanisms, allowing multiple threads or processes to acquire/release multiple permits. This is useful in scenarios where more than one resource is available, and multiple processes can proceed concurrently.