Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

- directly share a logical address space (that is, both code and data)
- or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency!

In this chapter, we discuss various mechanisms to ensure –

The orderly execution of cooperating processes that share a logical address space,

So that data consistency is maintained.

Producer Consumer Problem

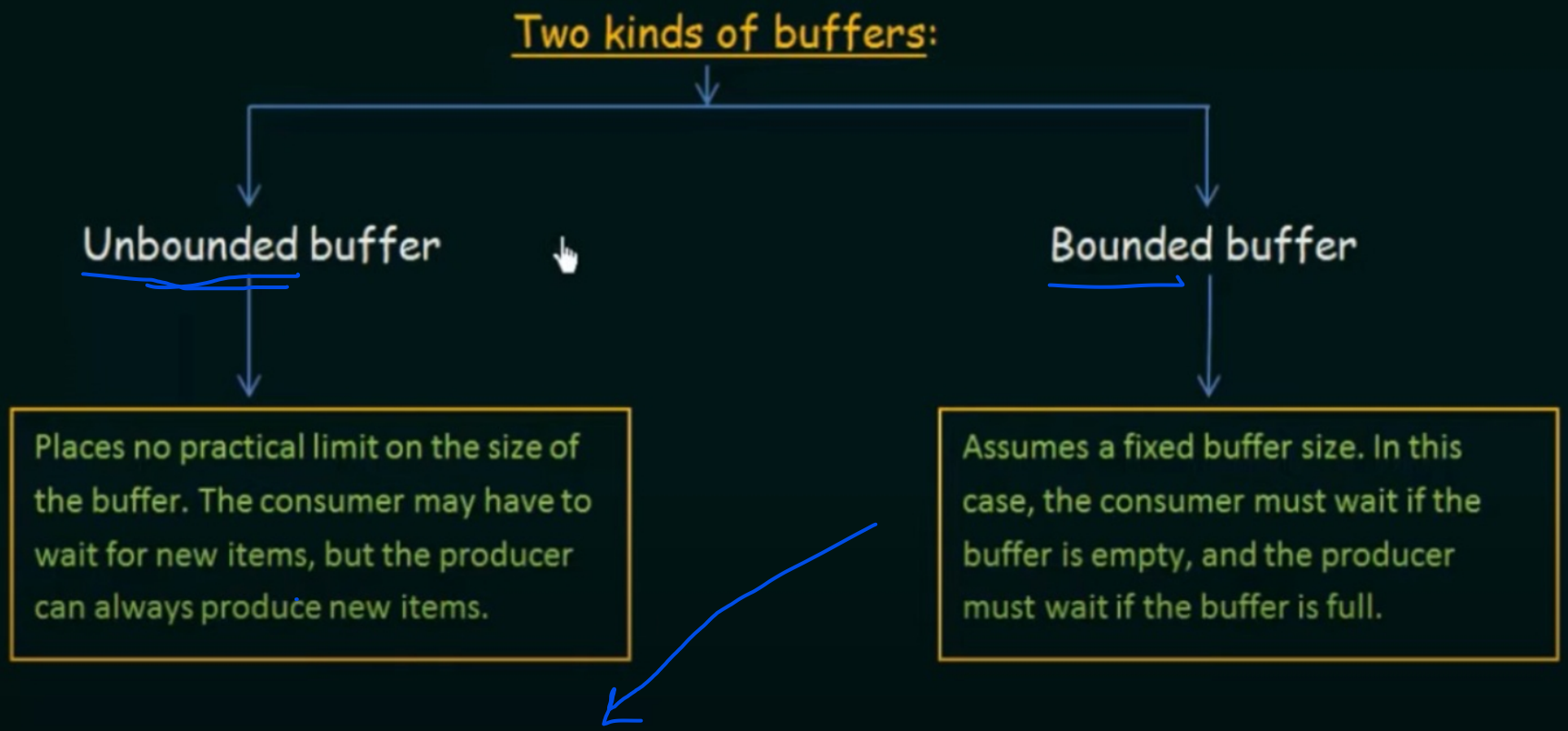
A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.


Two kinds of buffers:
Unbounded buffer
Bounded buffer
Places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

counter variable = 0

counter is incremented every time we add a new item to the buffer counter++

counter is decremented every time we remove one item from the buffer counter--

----------Example----------

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

Example

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

"counter++" may be implemented in machine language (on a typical machine) as:

$$\begin{aligned} register_1 &= counter \\ register_1 &= register_1 + 1 \\ counter &= register_1 \end{aligned}$$

"counter--" may be implemented in machine language (on a typical machine) as:

$$\begin{aligned} register_2 &= counter \\ register_2 &= register_2 - 1 \\ counter &= register_2 \end{aligned}$$

"counter++" may be implemented in machine language (on a typical machine) as:

```
register1 = counter
register1 = register1 + 1
counter   = register1
```

"counter--" may be implemented in machine language (on a typical machine) as:

```
register2 = counter
register2 = register2 - 1
counter   = register2
```

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	register	{ counter = 4 }

"counter++" may be implemented in machine language (on a typical machine) as:

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter = $register_1$

"counter--" may be implemented in machine language (on a typical machine) as:

$register_2$ = counter
$register_2$ = $register_2$ - 1
counter = $register_2$

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization.

The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_n\}$.

Each process has a segment of code, called a

critical section

in which the process may be changing common variables, updating a table, writing a file, and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections at the same time.

The critical-section problem is to design a protocol that the processes can use to cooperate.

The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_n\}$.

Each process has a segment of code, called a

critical section

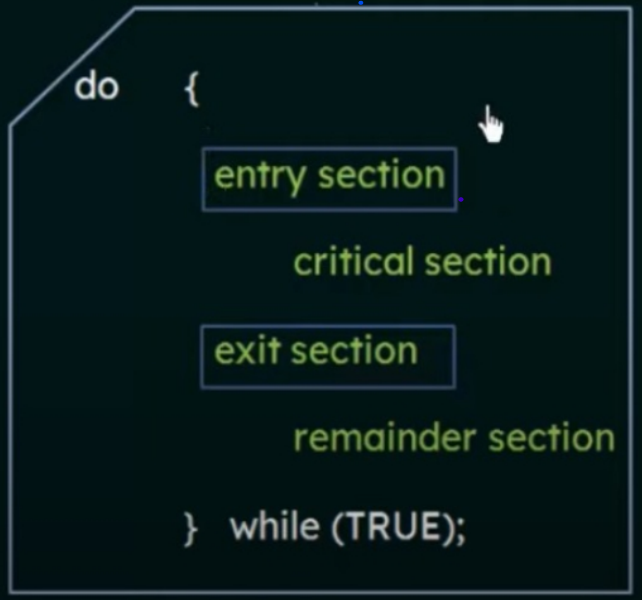
in which the process may be changing common variables, updating a table, writing a file, and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections at the same time.

The critical-section problem is to design a protocol that the processes can use to cooperate.

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.



A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion:

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress:

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

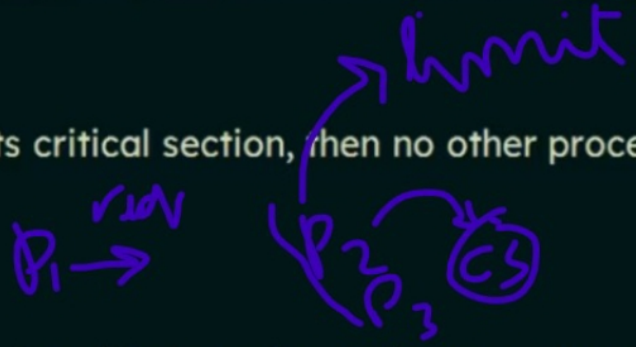
3. Bounded waiting:

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion:

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.



2. Progress:

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting:

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

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

int turn

- Indicates whose turn it is to enter its critical section.

boolean flag [2]

- Used to indicate if a process is ready to enter its critical section.

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

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

int turn

- Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

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

int turn

: Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

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

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

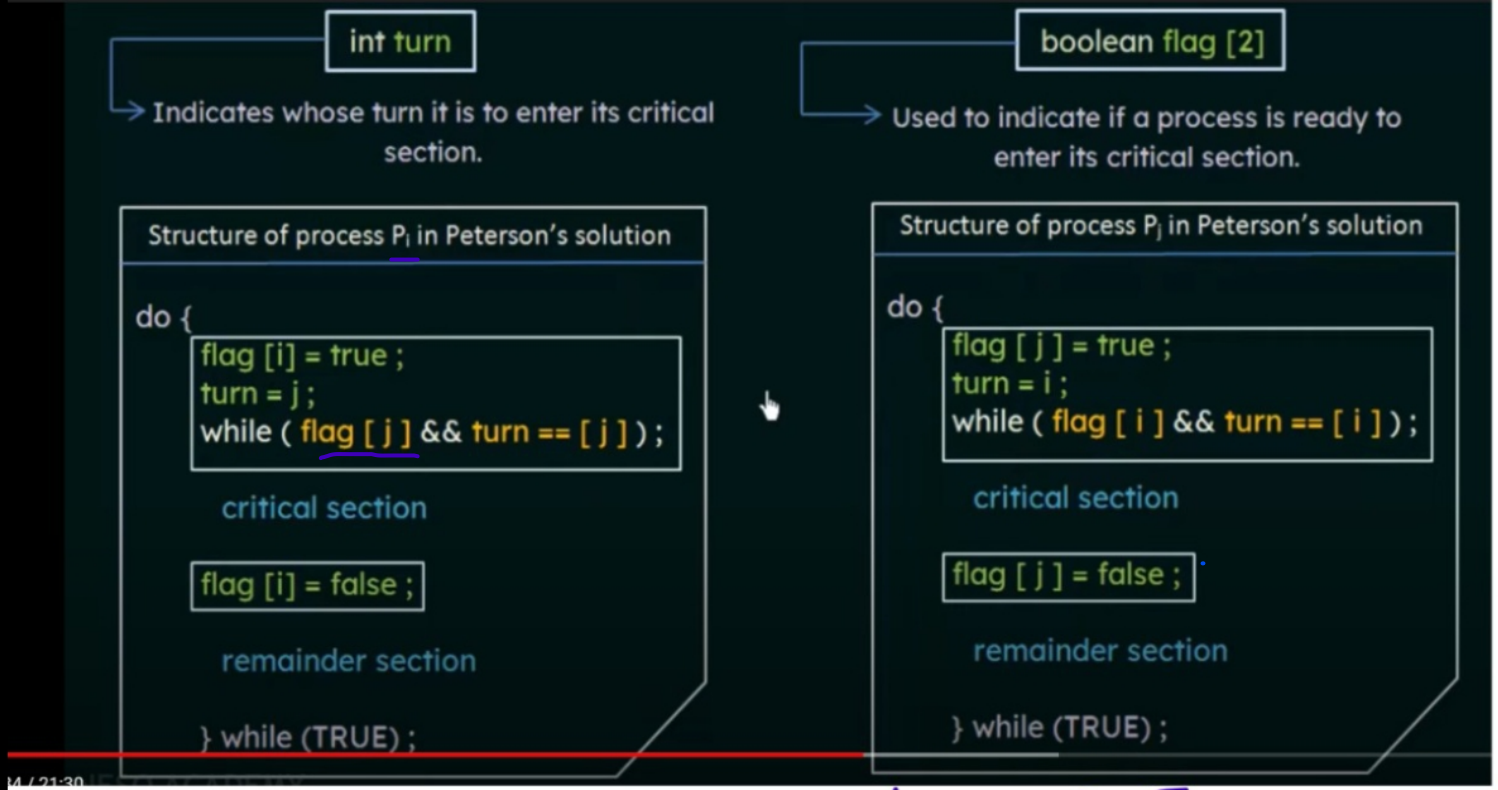
Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

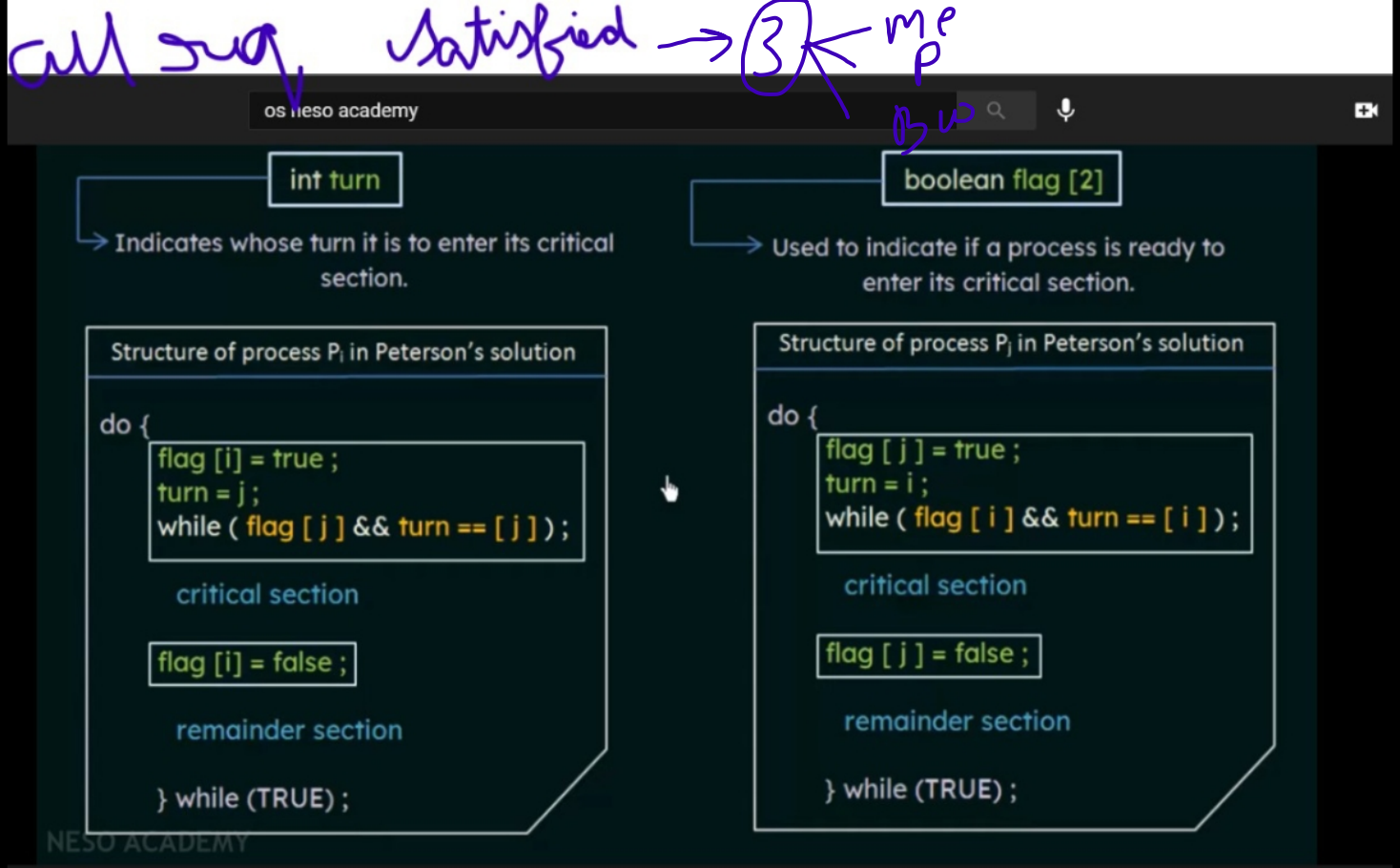
```
do {
    flag [i] = true ;
    turn = j ;
    while ( flag [ j ] && turn == [ j ] ) ;

        critical section

    flag [i] = false ;

        remainder section

    } while (TRUE) ;
```

Structure of process P_j in Peterson's solution

```
do {
    flag [ j ] = true ;
    turn = i ;
    while ( flag [ i ] && turn == [ i ] ) ;

        critical section

    flag [ j ] = false ;

        remainder section

    } while (TRUE) ;
```



no 2 Pro... should enter CS at same time


os neso academy
int turn
Indicates whose turn it is to enter its critical section.
boolean flag [2]
Used to indicate if a process is ready to enter its critical section.
Structure of process Pi in Peterson's solution
do {
flag [i] = true ;
turn = j ;
while (flag [j] && turn == [j]) ;
critical section
flag [i] = false ;
remainder section
} while (TRUE) ;
Structure of process Pj in Peterson's solution
do {
flag [j] = true ;
turn = i ;
while (flag [i] && turn == [i]) ;
critical section
flag [j] = false ;
remainder section
} while (TRUE) ;
NESO ACADEMY

Test and Set Lock

- A hardware solution to the synchronization problem.
- There is a shared lock variable which can take either of the two values, 0 or 1.
- Before entering into the critical section, a process inquires about the lock.
- If it is locked, it keeps on waiting till it becomes free.
- If it is not locked, it takes the lock and executes the critical section.

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

The definition of the TestAndSet () instruction

lock

Atomic Operation

Single operation (can't be interrupted)

lock = 0

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

Process P1

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

The definition of the TestAndSet () instruction

Initially lock → 0

Atomic Operation

now lock is 1

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

P1

Process P1 Process P2

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

Process P1 Process P2

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```