OPERATING SYSTEM
MU (IT Dept)
SEM IV
UNIT – III
(PROCESS COORDINATION)

By,
Himani Deshpande

UNIT- III (PROCESS COORDINATION)

- Basic Concepts of Inter-process Communication and Synchronization
  Race Condition;
  Critical Region and Problem;
  Peterson's Solution;
  Synchronization Hardware and Semaphores;
  Classic Problems of Synchronization;
  Message Passing;
  Introduction to Deadlocks;
  System Model, Deadlock Characterization;
  Deadlock Detection and Recovery;
  Deadlock Prevention;
  Deadlock Avoidance.

Inter-process Communication

- Processes frequently need to communicate with other processes.

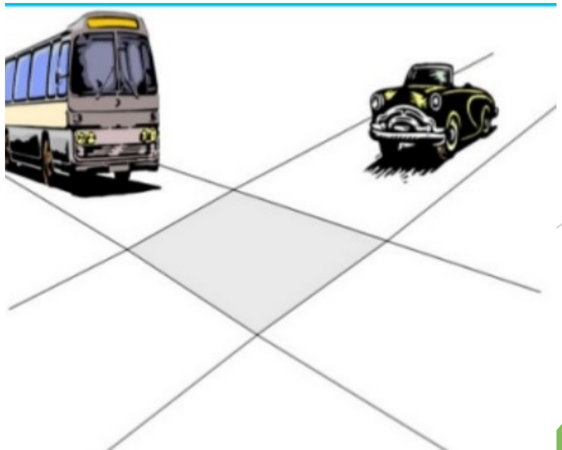
- Issues with process-communication :
  - how one process can pass information to another.
  - two or more processes do not get in each other's way
  - proper sequencing when dependencies are present:
    - if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print.

Synchronization

On the basis of synchronization, processes are categorized as

Process

- Independent
- Cooperative/ dependent



Process Synchronization

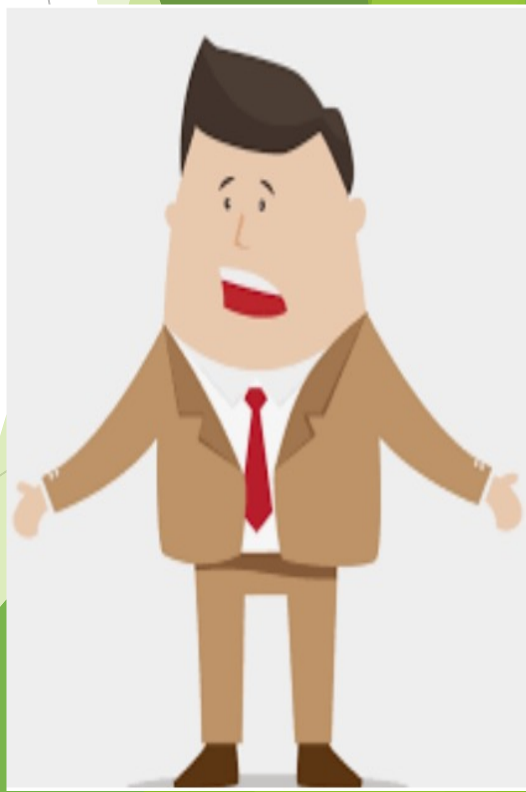
Independent Process :
Execution of one process does not affects the execution of other processes.

Cooperative Process :
Execution of one process affects the execution of other processes.

Race Condition

Race Condition

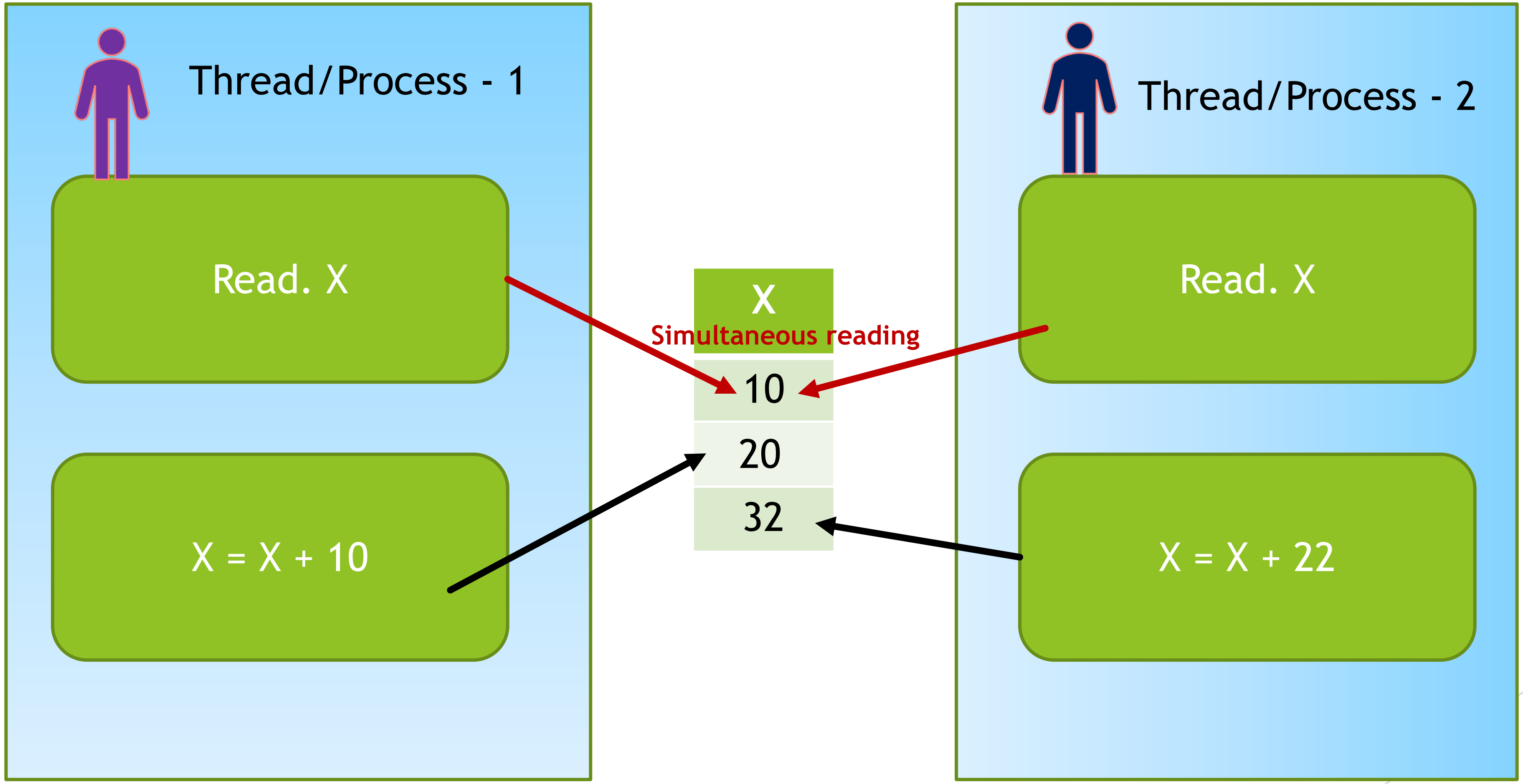
- It is possible to have a software system in which the output depends on the sequence of events.
- When events doesn't occur as the developer wanted, a fault happens. This is "Race Condition" .
- Race condition can take place when multiple processes operate on a shared Data.



Race Condition

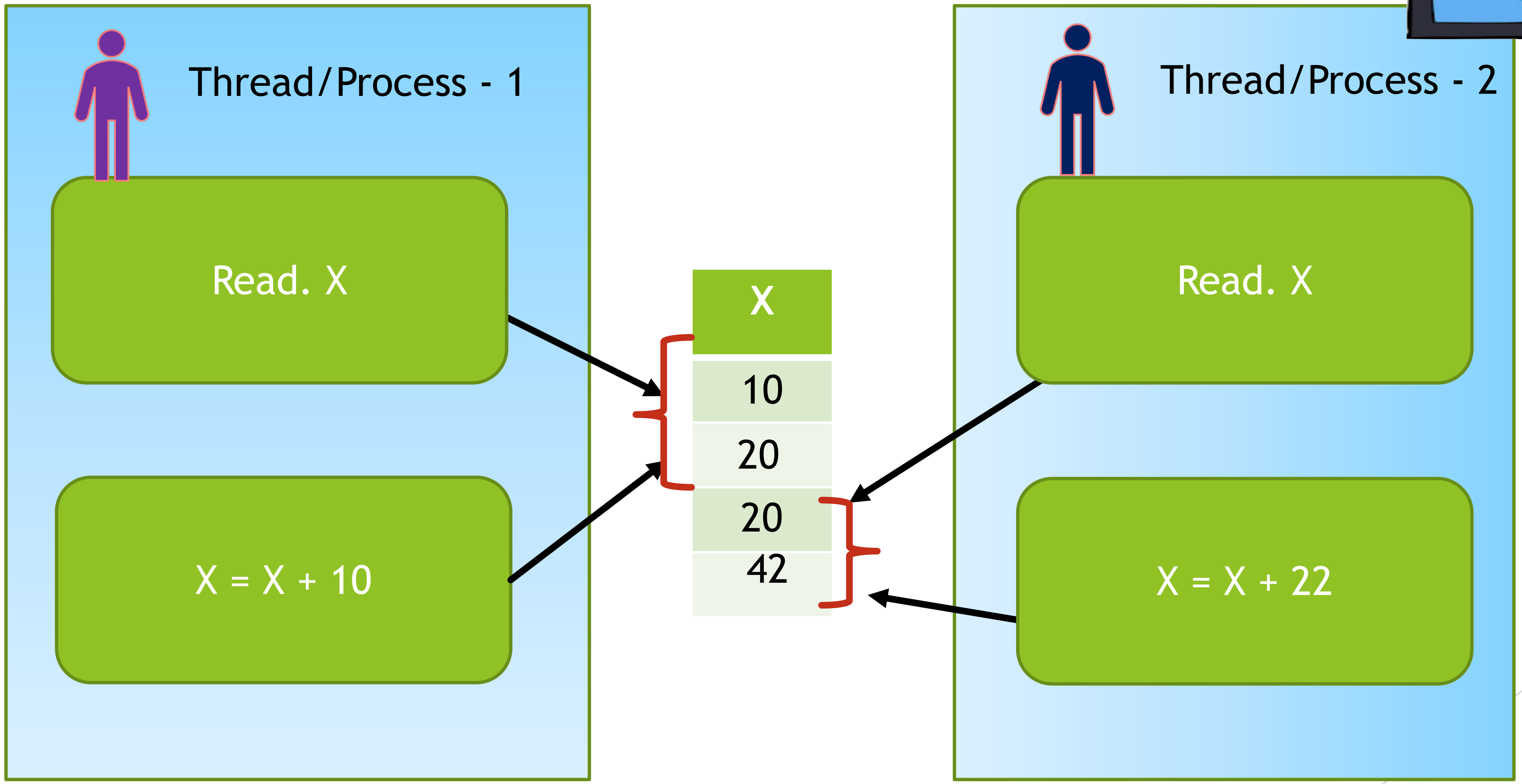
- A **race condition** or **race hazard** is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events.

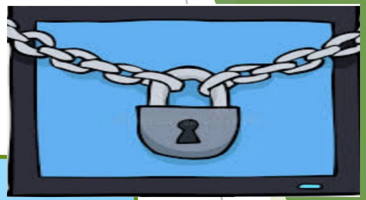
Example Race condition



Final Value of "X" was supposed to be 42, but its 32. Due to untimely read operation done by Process-2

Locking


Thread/Process - 1
Read. X
X = X + 10
X
10
20
20
42
Thread/Process - 2
Read. X
X = X + 22



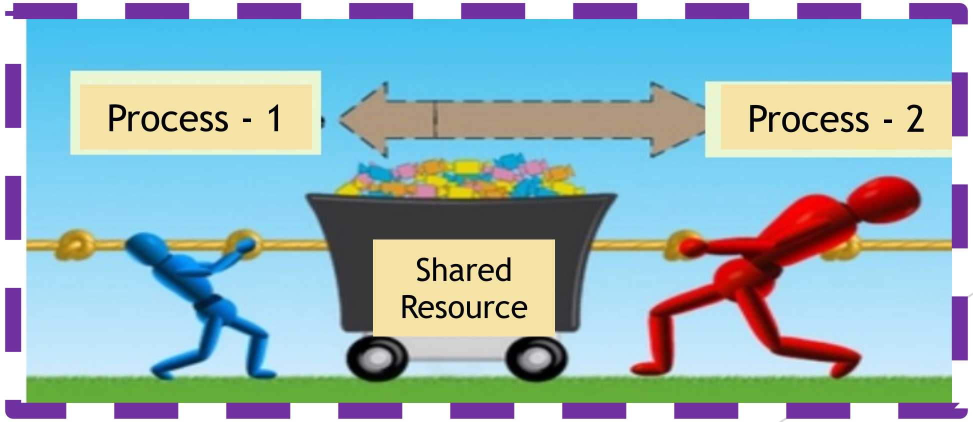
Critical Section

- One resource will be used exclusively by one person at a time.

Critical Section

Process synchronization is defined as a mechanism which ensures that :

"two or more concurrent **processes** do not simultaneously execute some particular program segment known as critical section."



Critical section

- Critical section is a code segment that accesses shared variables and has to be executed as an atomic action.

- Multiple processes wants to access the same code. But only one process must be executing its critical section at a given time.

```
do{
        entry-section

              Critical Section

        exit-Section

               remainder -section
   } while (true);
```

Critical Section

Critical section

```
do{
        entry-section
            Critical Section
        exit-Section
              remainder –section
   } while (true);
```

→ Gives lock to a process to enter critical section, when a condition is satisfied.

→ Resource will be locked

Only one process at a time

→ Removes the lock,
→ Other process knows current process Critical section is over

→ Remainder sector is independent of critical section.

→ If a process doesn't want to use critical section it can directly enter reminder section

→ Not every process enters Critical section
→ Only those processes that use shared variables enters the critical section

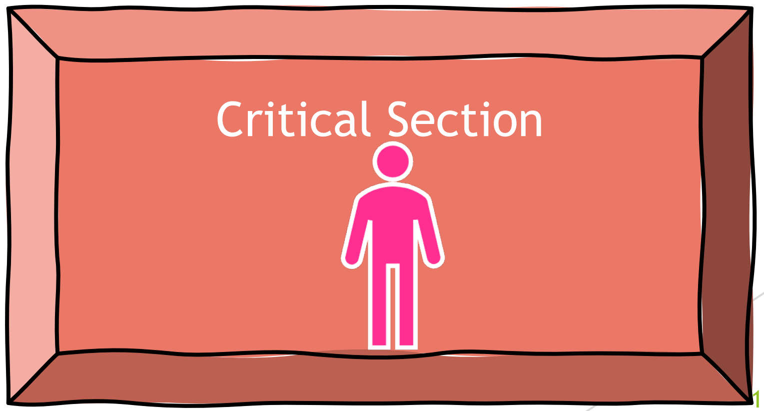
Critical Section Solution requirement

- **Mutual Exclusion:**

  Only one process should execute in its critical section at a time.

  Exclusive access of each process to the shared memory/resource.

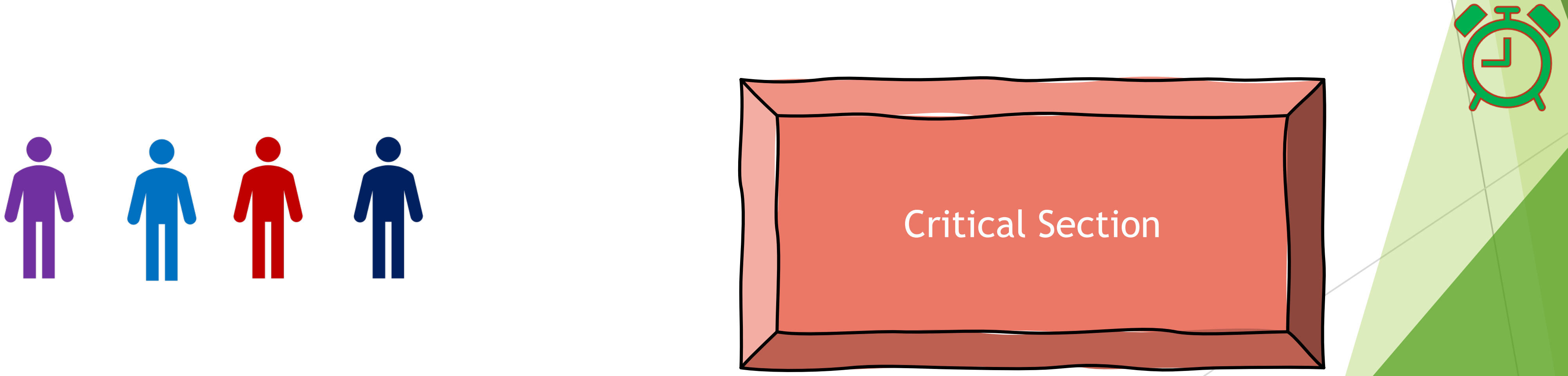
“no two processes can exist in the critical section at any given point of time”.



Critical Section Solution requirement

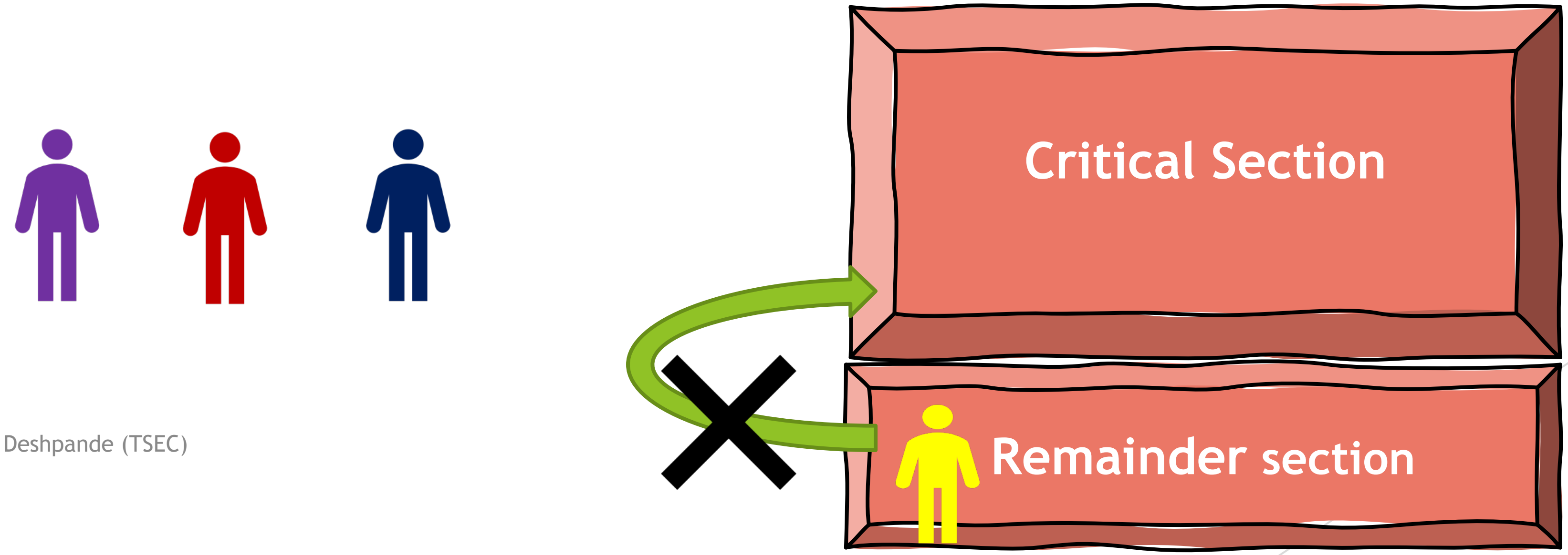
- **Bounded Waiting:**

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

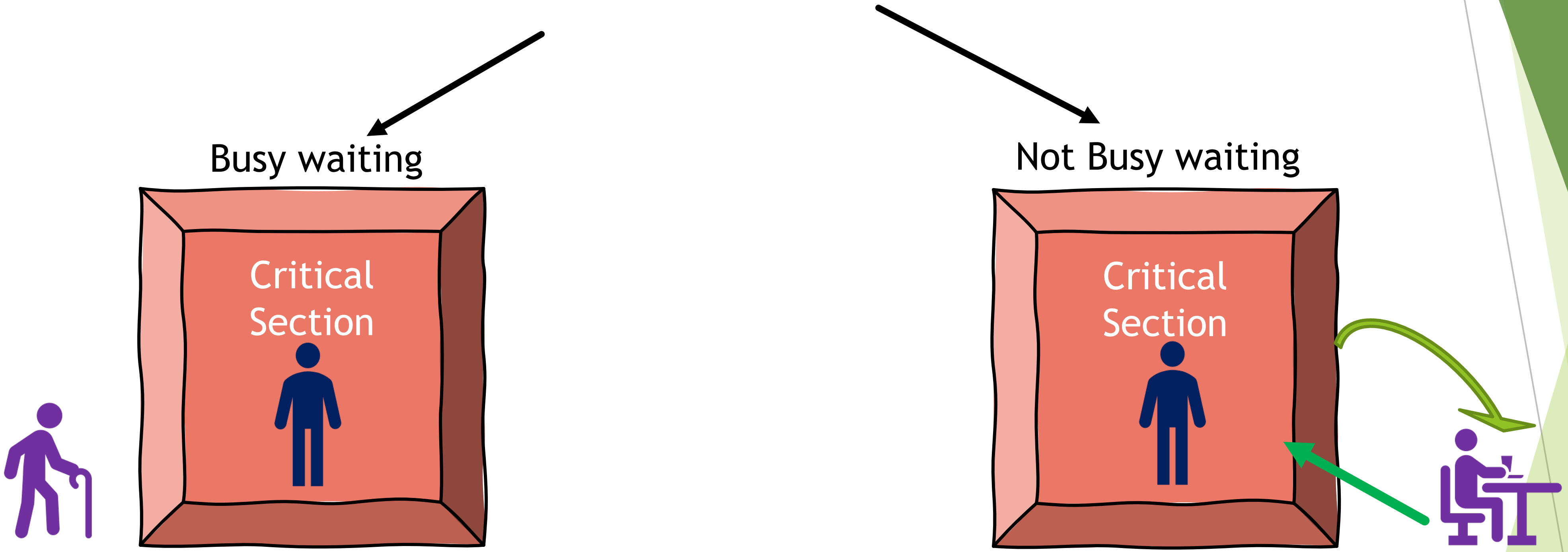


Critical Section Solution requirement

- **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 section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.



Synchronization



Critical Section Solution

- Peterson's Algorithm
- Semaphore
- Hardware Synchronization

Peterson’s Algorithm

- Peterson’s solution requires the two processes to share two data items.
- Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- Assume that the LOAD/read and STORE/write instruction are atomic; i.e. cant be interrupted.

Peterson's Solution

In Peterson's solution, we have two shared variables:

→**boolean flag[2]**

Initialized to FALSE, initially no one is interested in entering the critical section

→**int turn**

The process whose turn is to enter the critical section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Critical Section

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

Peterson's Solution

Peterson’s Solution

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

Process “i” is interested to enter Critical Section

Process “i” gives chance to process “j”

If process “j” is willing and its j’s turn.
Process “I” will keep waiting

Process “i” turns its flag to FALSE

```
bool flag[0] = {false};
bool flag[1] = {false};
int turn;
```

1. Mutual Exclusion 2. Progress. 3. Bounded Waiting

The turn value can not be 0 and 1 as the same time

```
do{
flag[0] = true;    // Process 0 is interested to enter CC
turn = 1;    // Process 0 giving turn to process 1
while (flag[1] == true && turn == 1);    // busy wait

    // critical section
    ...
    // end of critical section
    flag[0] = false;
}while(true);
```

Process 0

```
do{
flag[1] = true;    // Process 1 is interested to enter CC
turn = 0;    // Process 1 giving turn to process 0
while (flag[0] == true && turn == 0);    // busy wait

    // critical section
    ...
    // end of critical section
    flag[1] = false;
}while(true);
```

Process 1

Peterson’s Solution

- Peterson’s Solution preserves all three conditions :
  - Mutual Exclusion is assured as only one process can access the critical section at any time.
  - Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
  - Bounded Waiting is preserved as every process gets a fair chance.

- Disadvantages of Peterson’s Solution
  - It involves Busy waiting
  - It is limited to 2 processes.

Semaphore

- A semaphore is a programming concept that is frequently used to solve CS synchronization problems.
- It is the oldest of the scheduler-based synchronization mechanisms.
- A semaphore is somewhat like an integer variable, but is special in its operations (increment and decrement) .
- Semaphore can facilitate and restrict access to shared resources in a multi-process environment.
- Semaphores are also specifically designed to support an efficient waiting mechanism.
- Semaphores helps avoid race condition.

Atomic behaviour

Friend fighting for last slice

Slice acquiring is atomic
only one can pick at a time

Semaphore

A semaphore is a an integer variable that is used to solve the CS problem by using two atomic operations, wait and signal that are used for process synchronization

P

V

Operation **P or Wait ( )**
atomically **decrements** the counter and then waits until it is non-negative.

Operation **"V" or Signal()**
atomically **increments** the counter and wakes up a waiting process, if any.

>0 is available

0 or –ve is locked

Semaphore

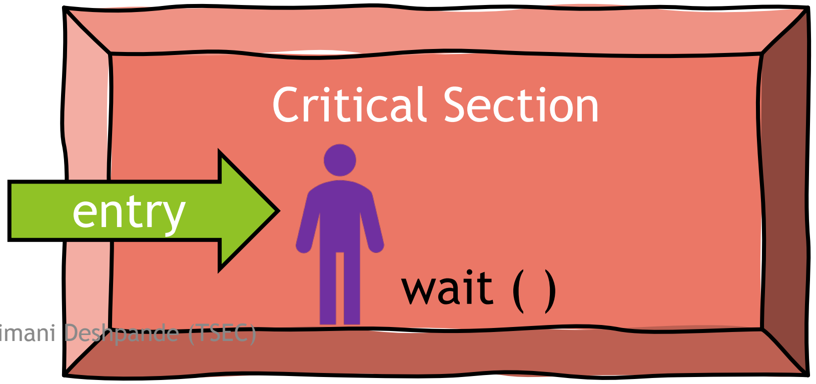

```
wait(s)
{
  while (s<=0) ;   //wait (no operation)
   s--  ;
}
```

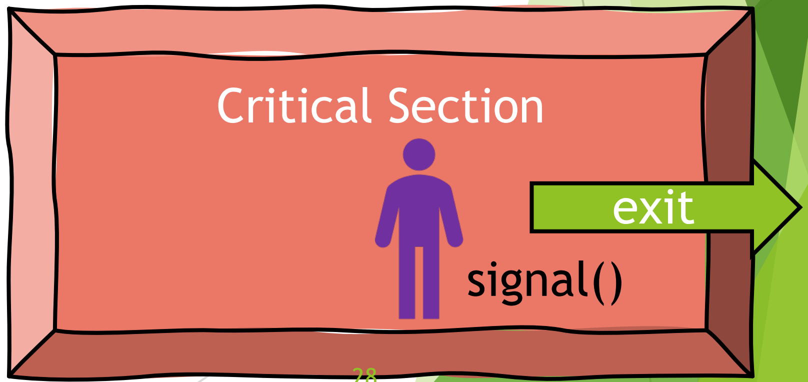
→ when a semaphore is zero it is "locked" or "in use".

→ Positive values indicate that the semaphore is available.

→ Only one process can modify same semaphore value at a time.

```
signal(s)
{
  s++;
}
```





Semaphore

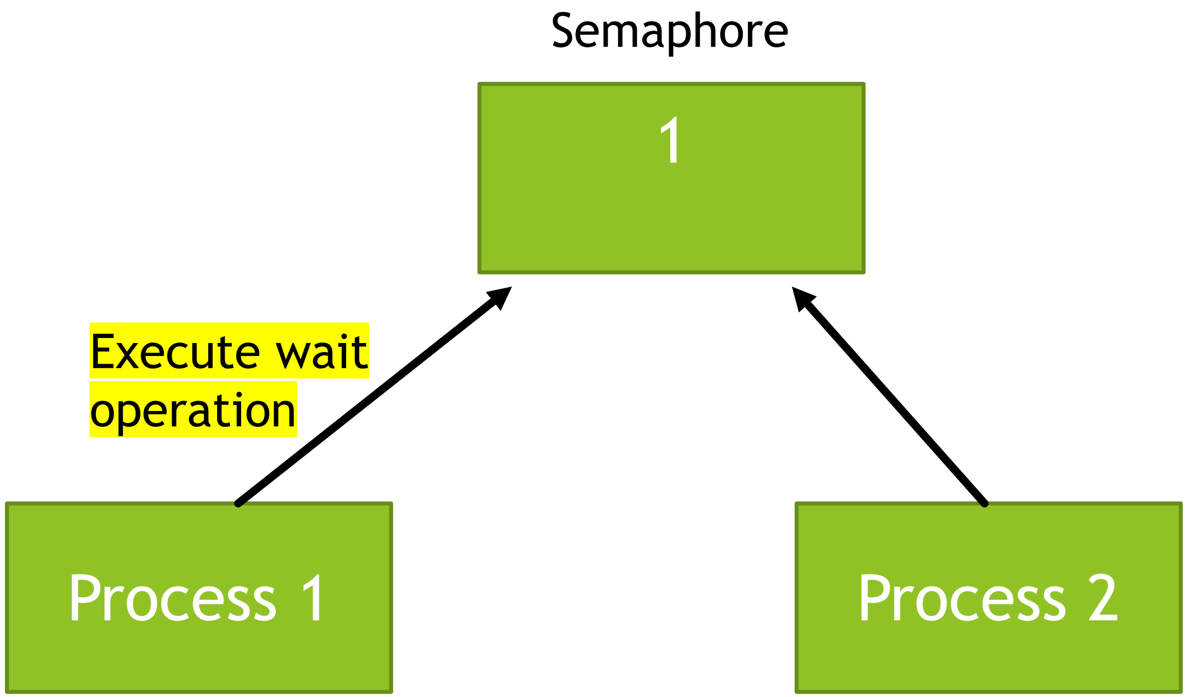
```
do
{
    wait (s);
    // Critical Section
    signal(s);
    //remainder section
} while(true);
```

```
wait(S)
{
  while (S<=0);   // no operation

  S--;
}
```

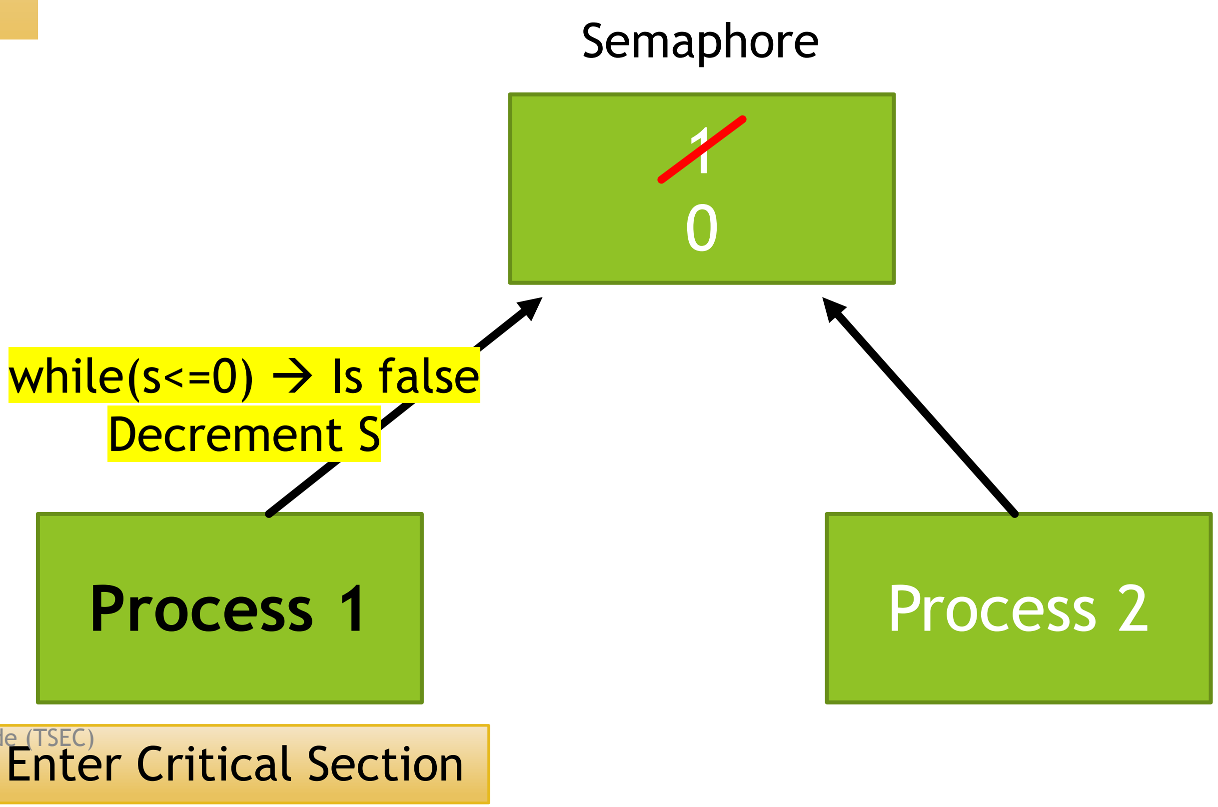
```
signal(S)
{
  S++;
}
```

Semaphore

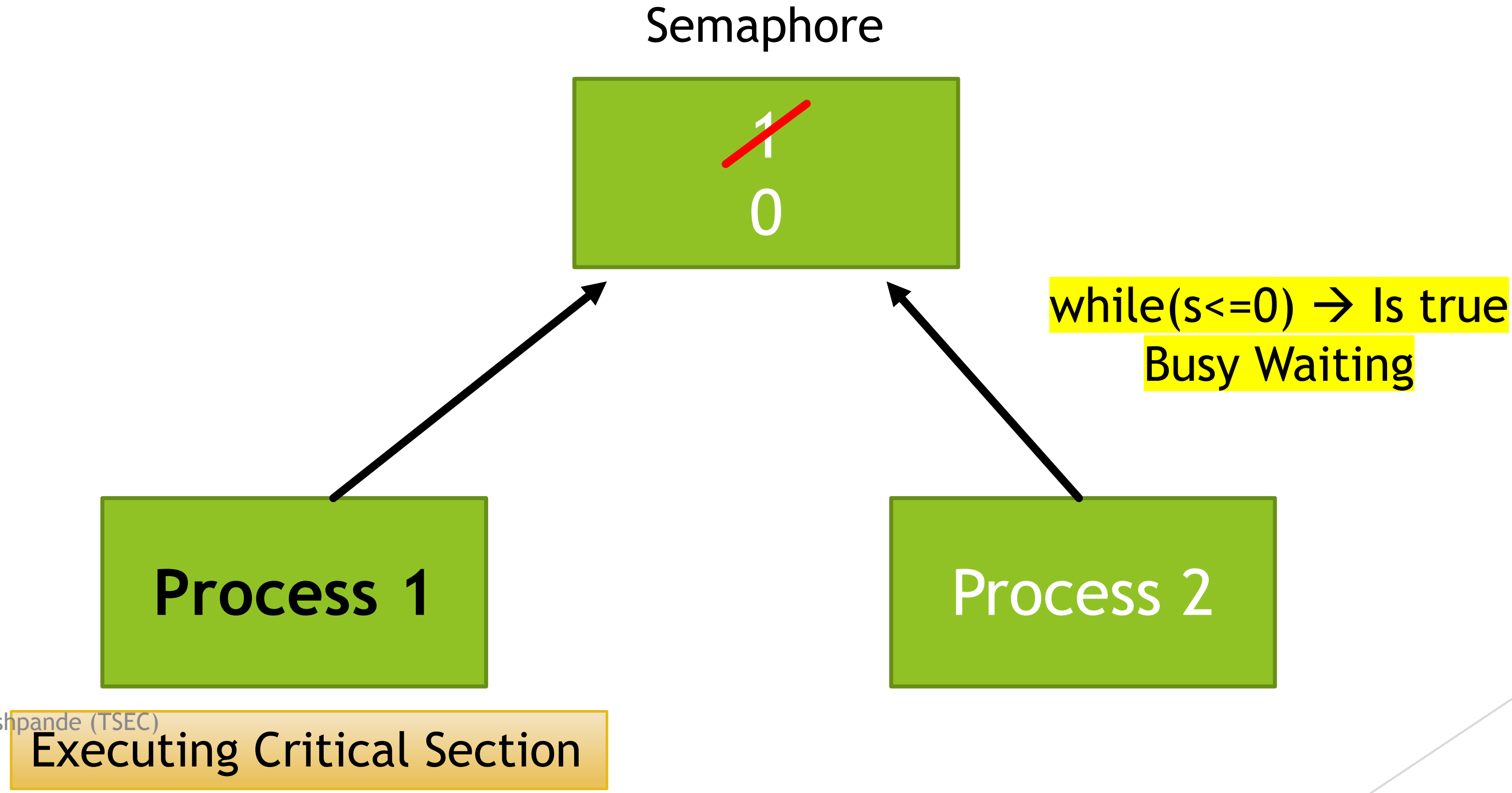


Semaphore

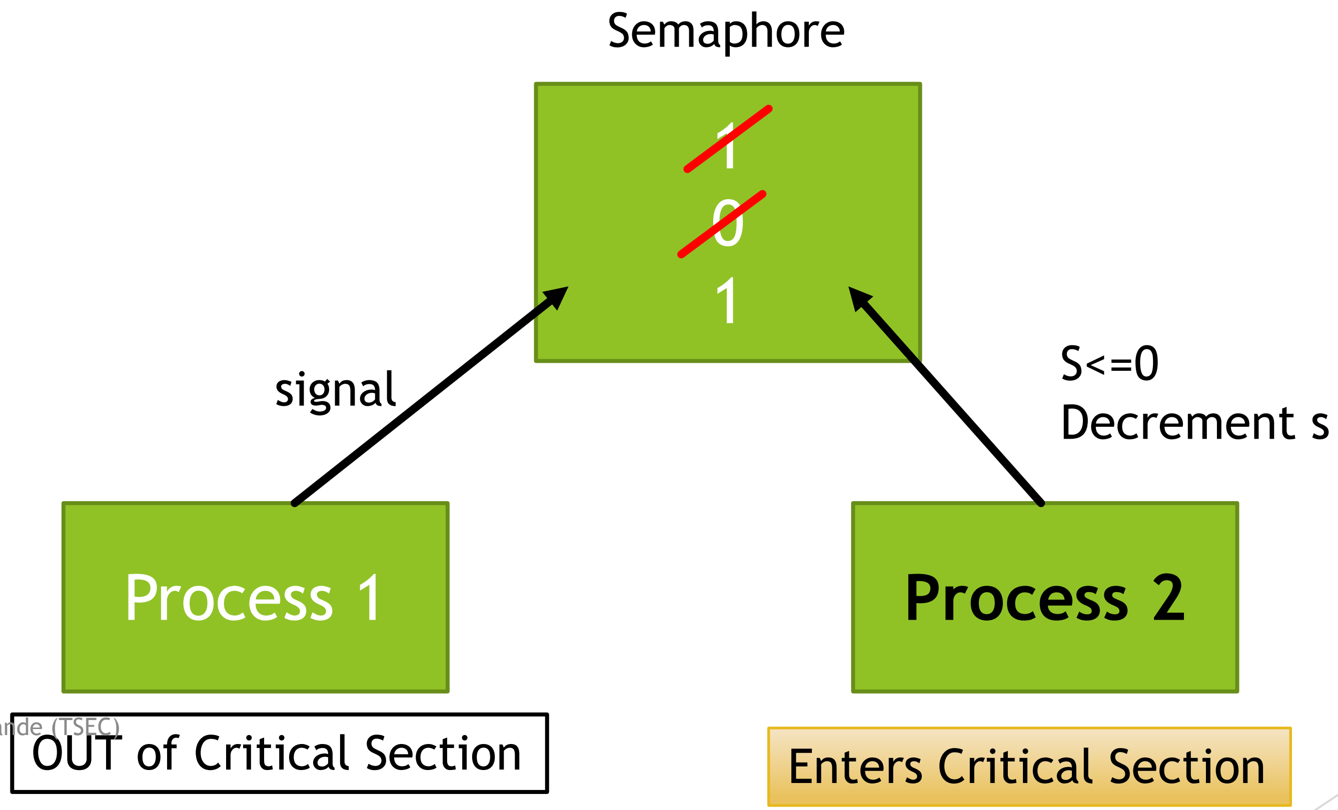
```
wait(s)
{
   while (s<=0) ;
    s--  ;
}
```



Semaphore


Semaphore
~~1~~
0
while(s<=0) → Is true
Busy Waiting
Process 1
Process 2
Executing Critical Section

Semaphore


Semaphore
~~1~~
~~0~~
1
signal
S<=0
Decrement s
Process 1
Process 2
OUT of Critical Section
Enters Critical Section

Semaphore

- Binary (mutex lock)

A semaphore whose counter is initialized to 1 and for which P and V operations always occur in matched pairs is known as a *binary semaphore*.

- Counting

Integer value can range over an unrestricted domain.

Mutex

Mutex is the short form for 'Mutual Exclusion Object'.

A Mutex and the binary semaphore are essentially the same.

Both Mutex and the binary semaphore can take values: 0 or 1.

Hardware Synchronization

Solution to Critical-section Problem Using Locks

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

Eg. SE Class teacher blocks Saturday 11:00-12:00 slot

Other teachers will compete for 12:00-1:00 slot

Hardware Synchronization

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**

  → Protecting critical regions via locks
- **Uniprocessors** – could disable interrupts

  Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems

Modern machines provide special atomic hardware instructions

Atomic = non-interruptible

Two types of Instructions :

1. Either test memory word and set value
2. swap contents of two memory words

TestAndSet Instruction

TestAndSet Instruction

Definition:

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

Must be executed atomically

Solution using TestAndSet

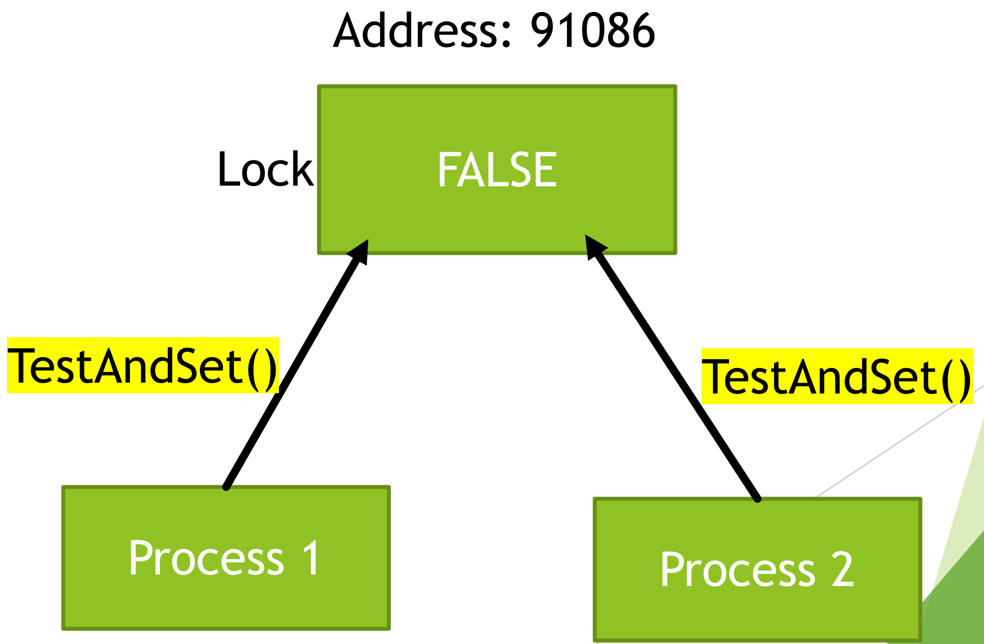
- Shared Boolean variable lock, initialized to False
- TestAndSet instruction is executed Automatically

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

Solution using TestAndSet

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

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

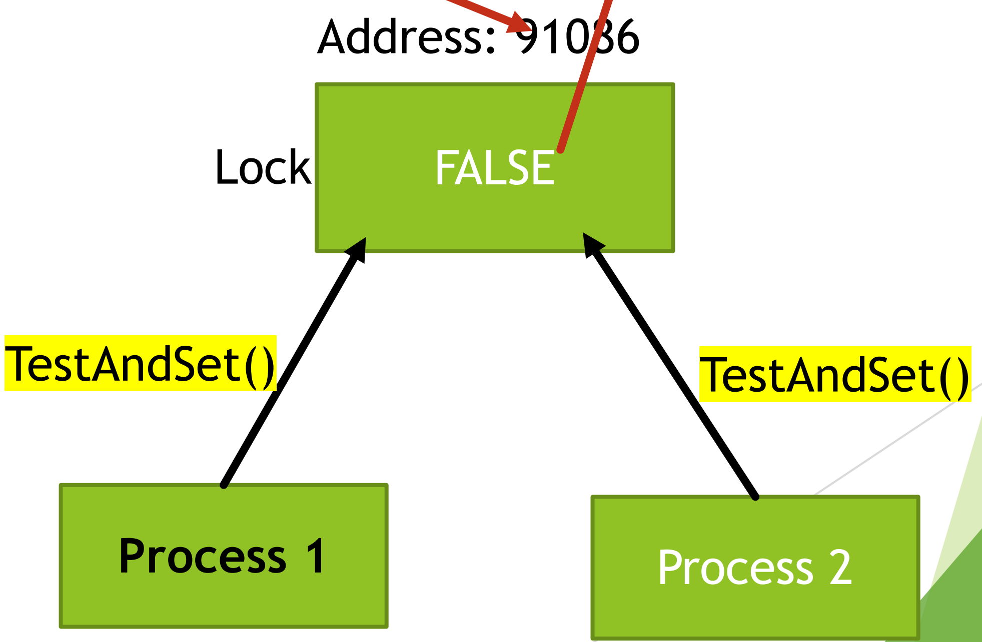


Solution using TestAndSet

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

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

Process 1



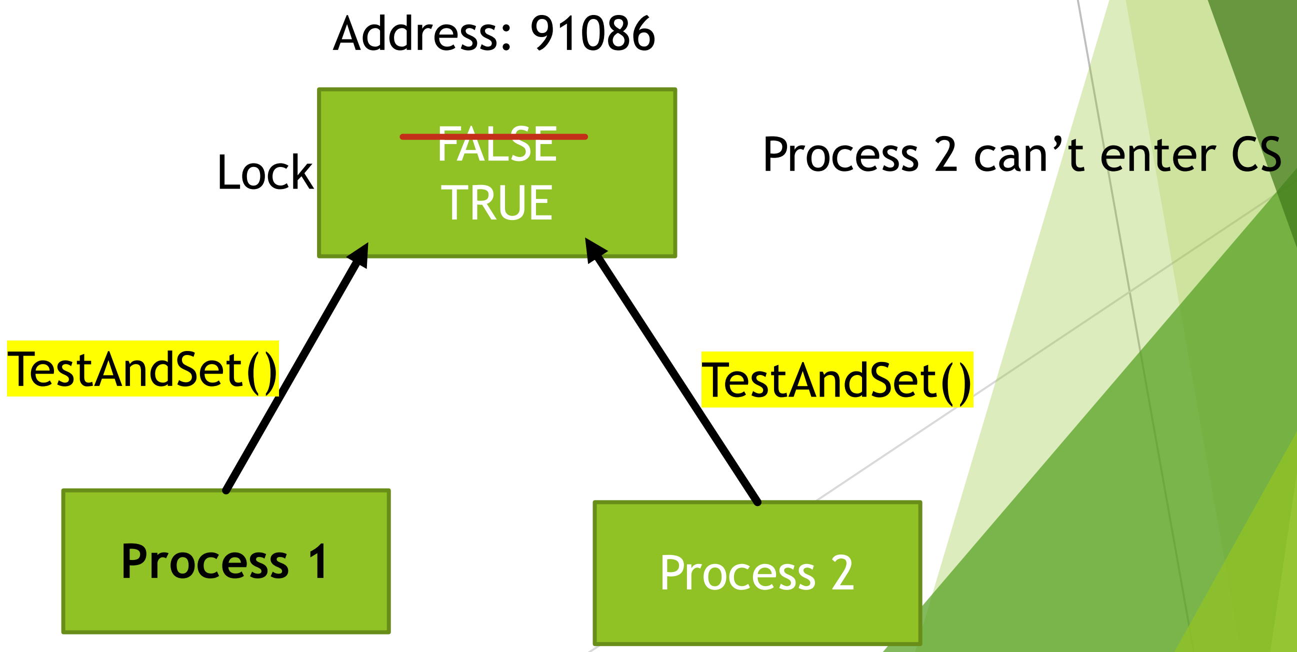
As TestAndSet() is atomic only one of the processes will be able to execute at a time.

Let's , assume Process 1 execute

Solution using TestAndSet

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

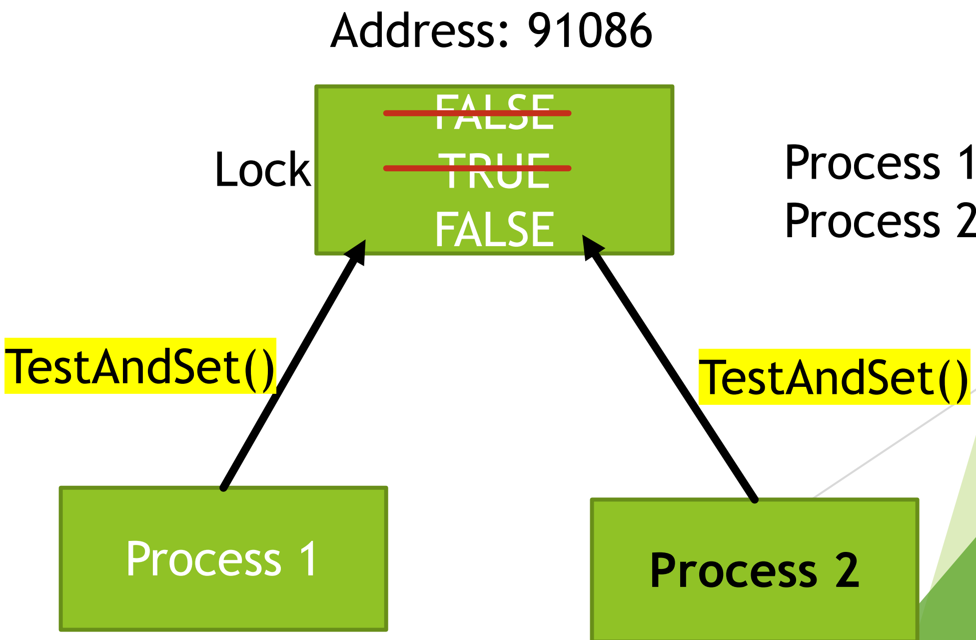
```
do {
            while ( TestAndSet (&lock ))      ;  // do nothing
                  //    critical section
            lock = FALSE;
                  //     remainder section
      } while (TRUE);
```



Solution using TestAndSet

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

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



Process 1 exits
Process 2 can enter CS

Swap Instruction

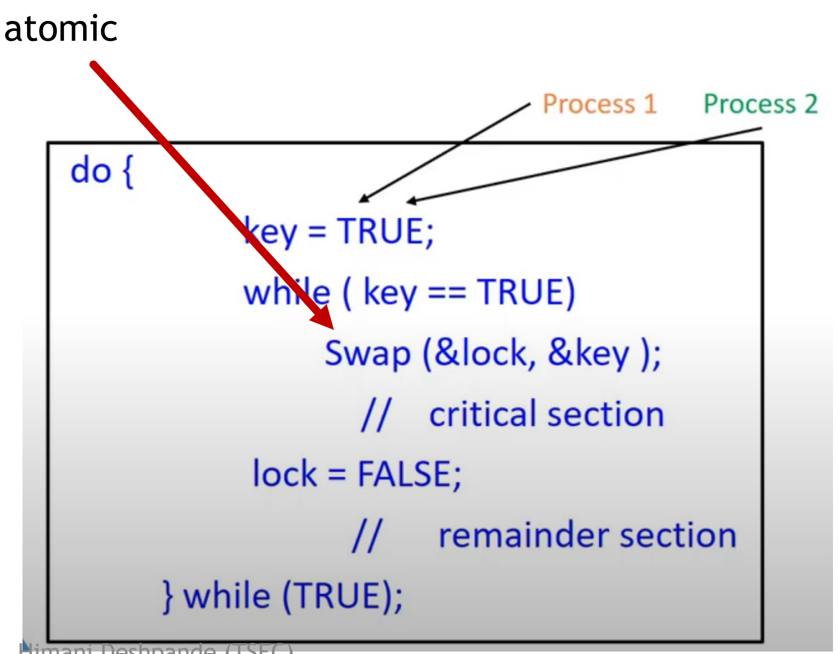
Swap Instruction

Definition:

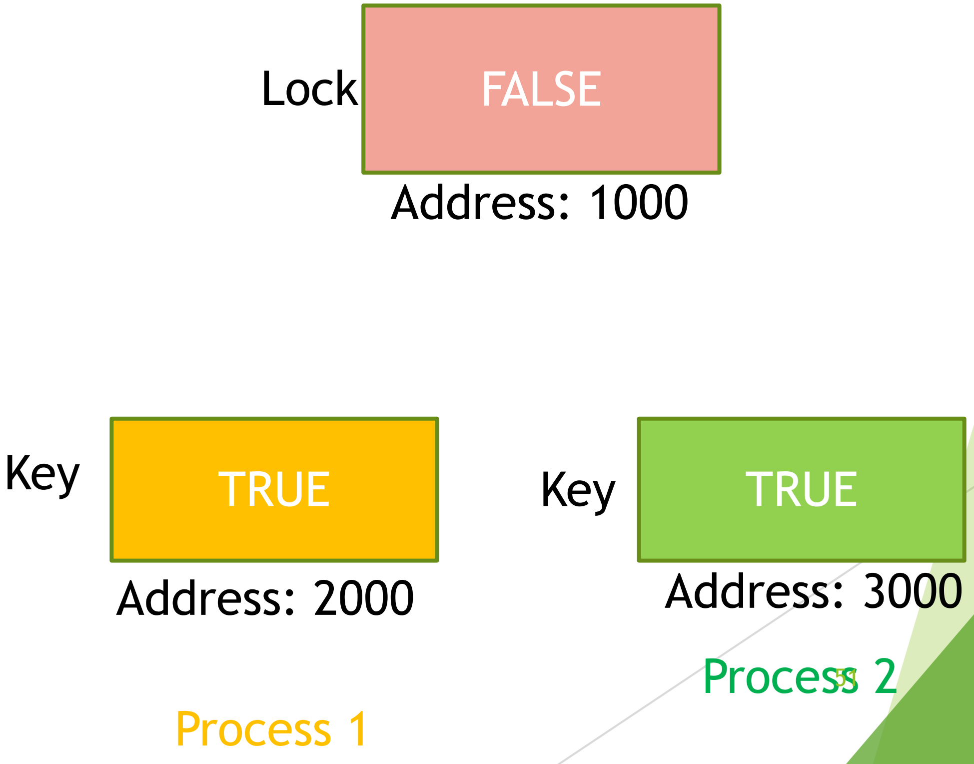
```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Swap

- Shared Boolean variable lock initialized to FALSE.
- Each process has a local Boolean variable key



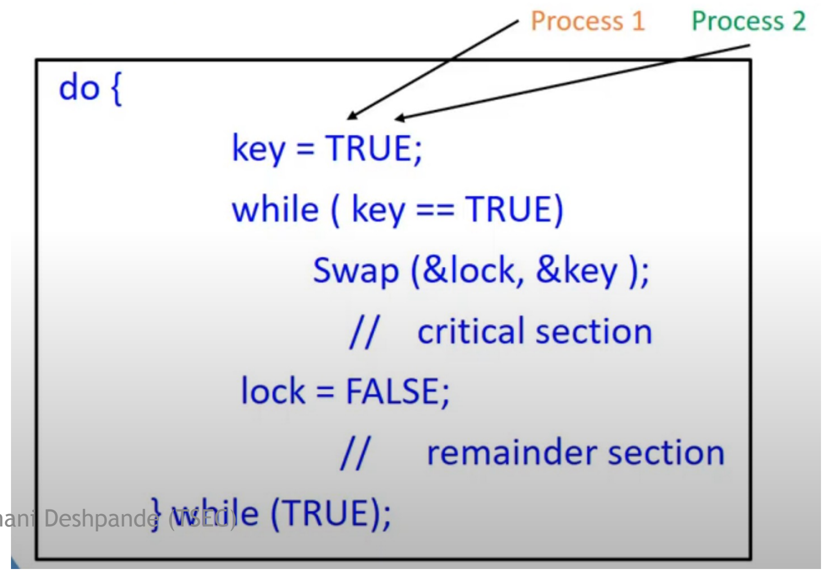

```
do {
        key = TRUE;
        while ( key == TRUE)
            Swap (&lock, &key );
                //   critical section
        lock = FALSE;
                //   remainder section
} while (TRUE);
```



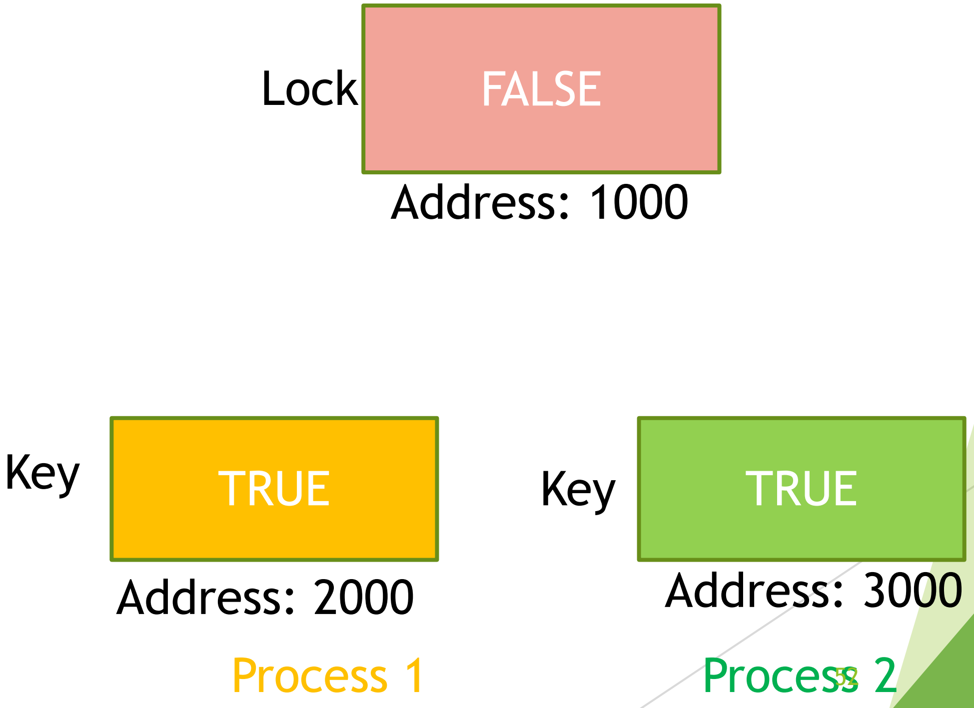
Swap

Assume Process 1 is executing

- Shared Boolean variable lock initialized to FALSE.
- Each process has a local Boolean variable key



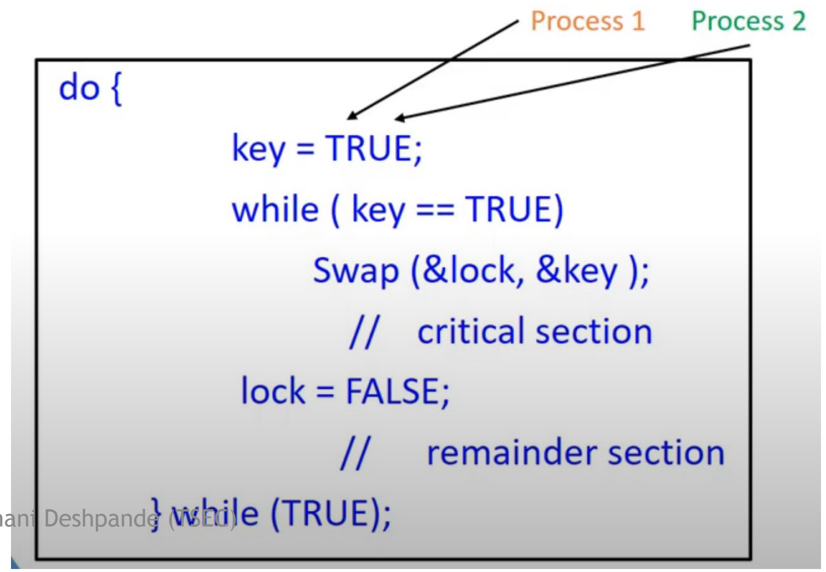

```
do {
        key = TRUE;
        while ( key == TRUE)
            Swap (&lock, &key );
              //   critical section
        lock = FALSE;
              //    remainder section
} while (TRUE);
```

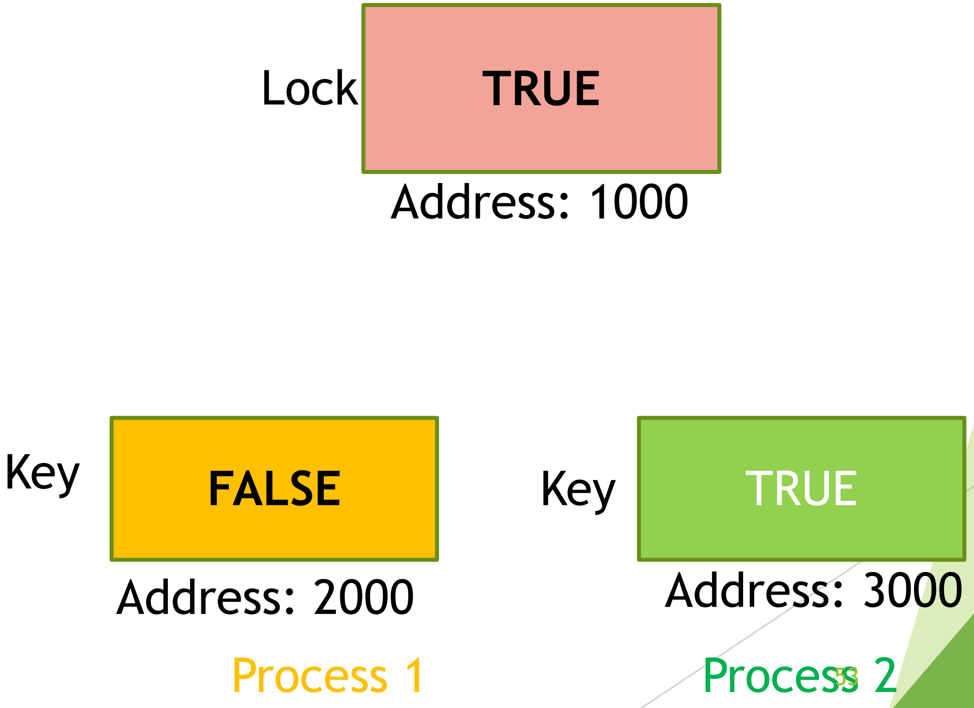


Swap

Process 1 is in Critical Section

- Shared Boolean variable lock initialized to FALSE.
- Each process has a local Boolean variable key





Swap

Process 1 comes out of CS ,changes lock to **FALSE**

- Shared Boolean variable lock initialized to FALSE.
- Each process has a local Boolean variable key

Process 1 Process 2

```
do {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );
                  //    critical section
        lock = FALSE;
                  //     remainder section
} while (TRUE);
```