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.

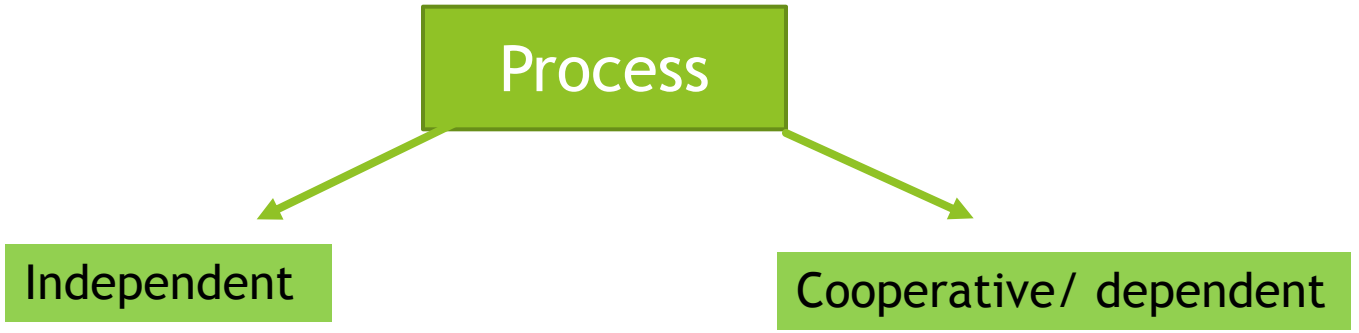
Inter-process Communication

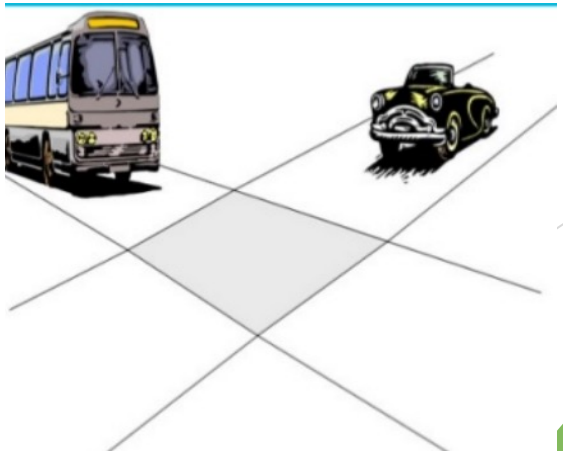
- Processes frequently need to communicate with other processes.

- 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 Synchronization

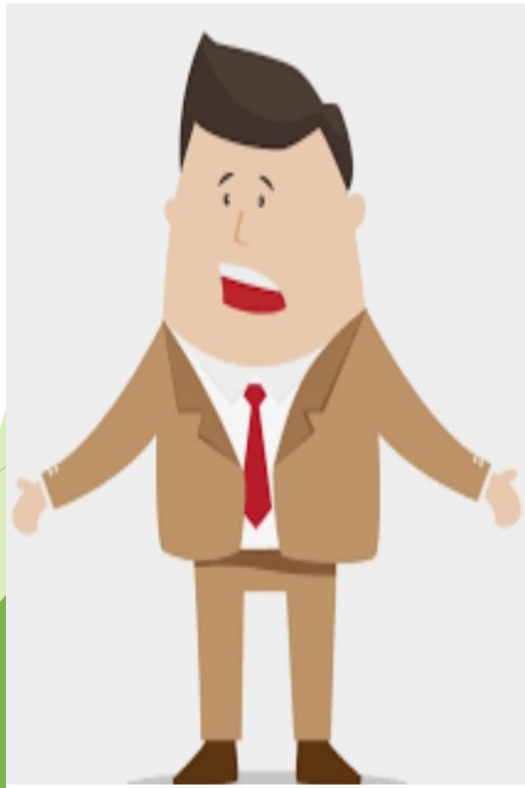
Independent Process :
Execution of one process does not affects the execution of other processes.

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

Race Condition

Race Condition

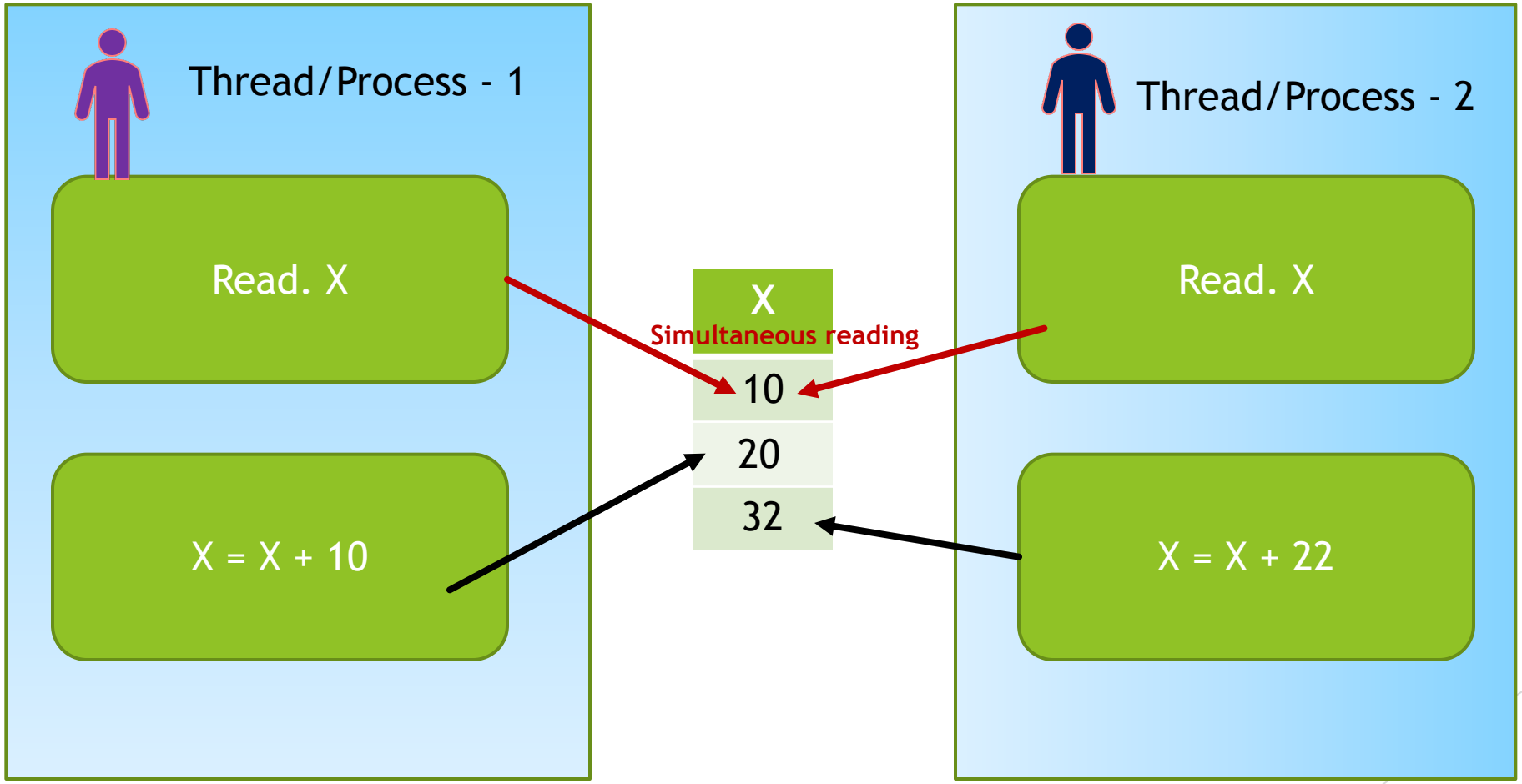
- It is possible to have a software system in which the output depends on the sequence of events.
- When events doesn't occur as the developer wanted, a fault happens. This is "Race Condition" .
- Race condition can take place when multiple processes operate on a shared Data.



Race Condition

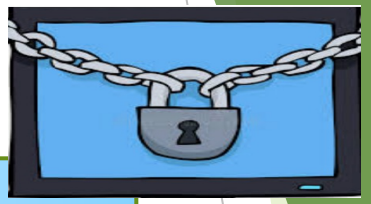
- A **race condition** or **race hazard** is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events.

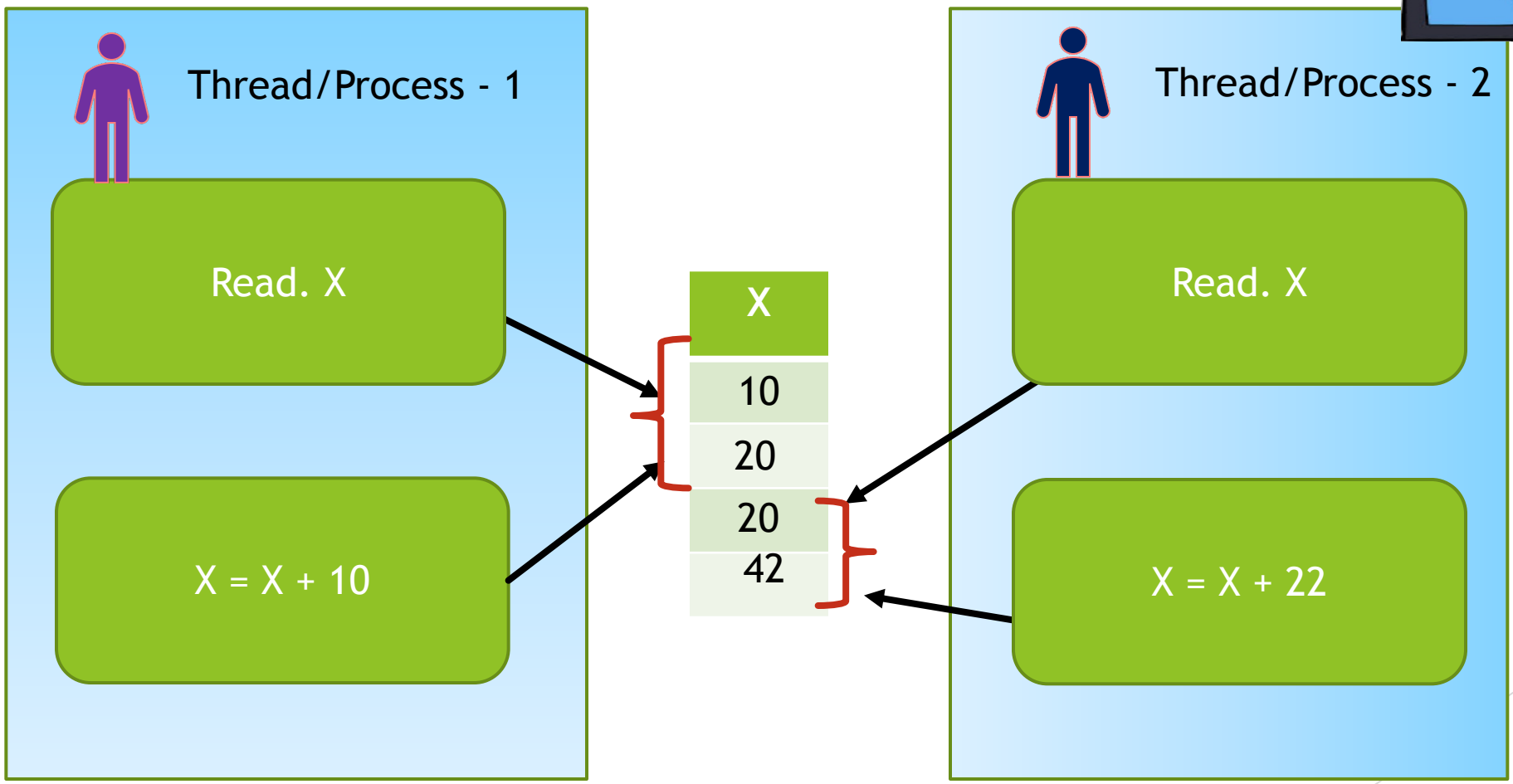
Example Race condition



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

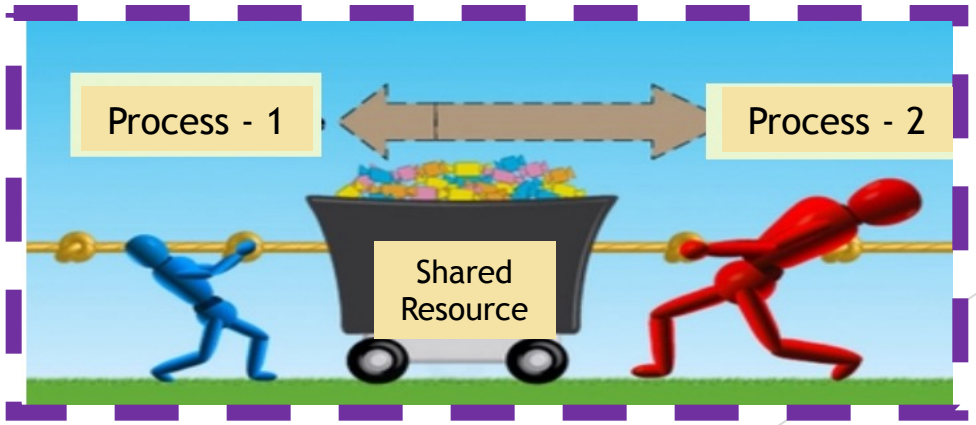
Critical Section

- 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."

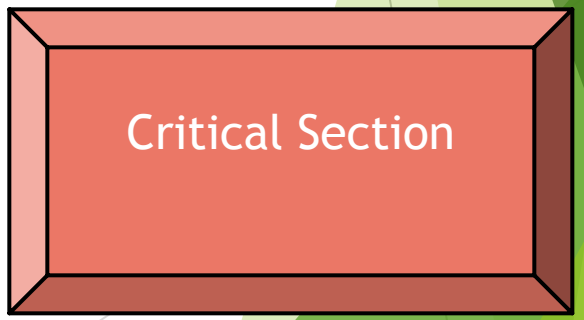


Critical section

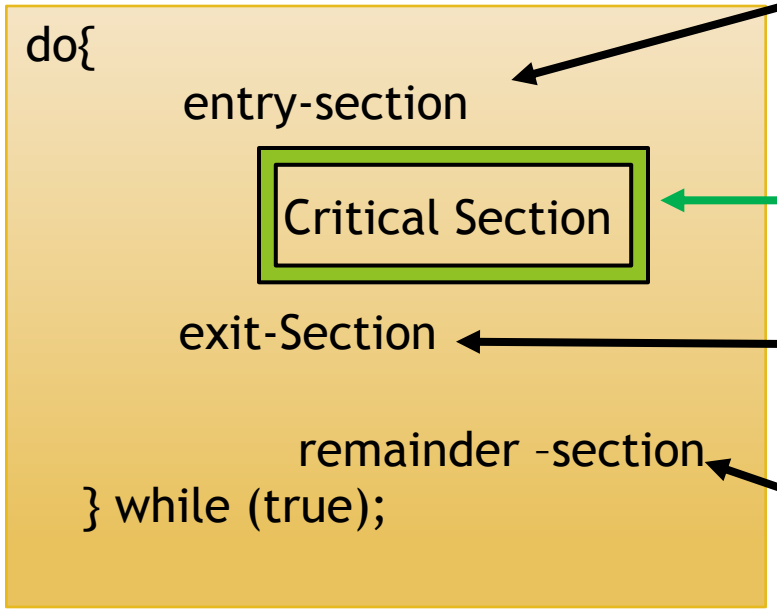
- Critical section is a code segment that accesses shared variables and has to be executed as an atomic action.

- 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



→ 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

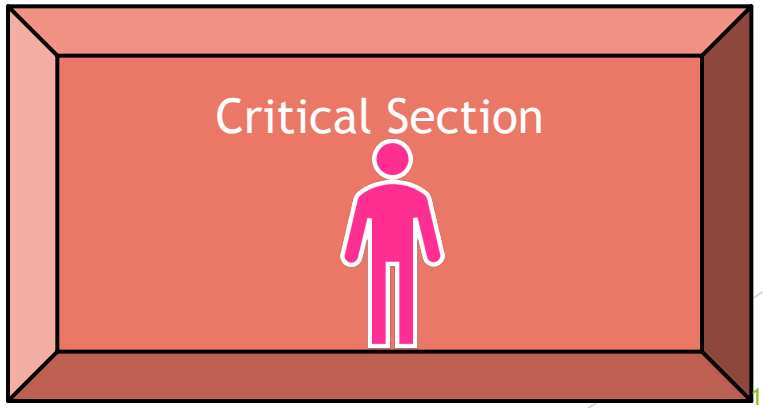
→ Not every process enters Critical section
→ Only those processes that use shared variables enters the critical section

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.

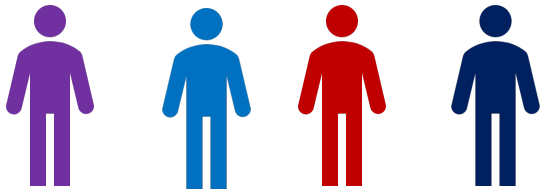
“no two processes can exist in the critical section at any given point of time”.

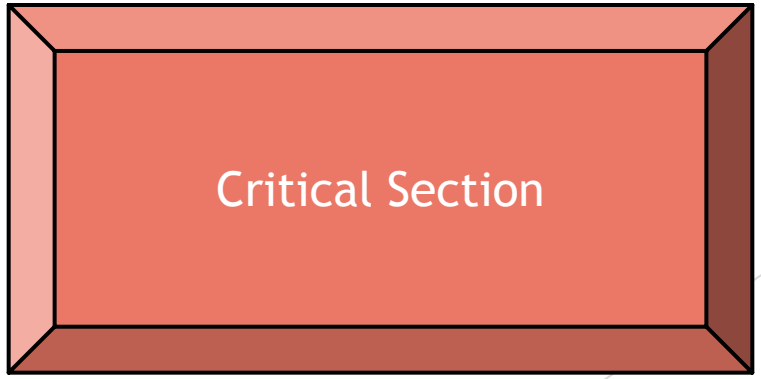


Critical Section Solution requirement

- **Bounded Waiting:**

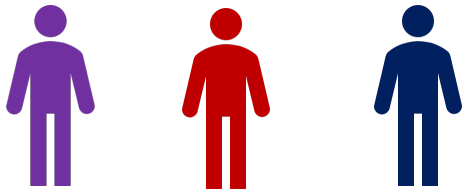
- There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted.

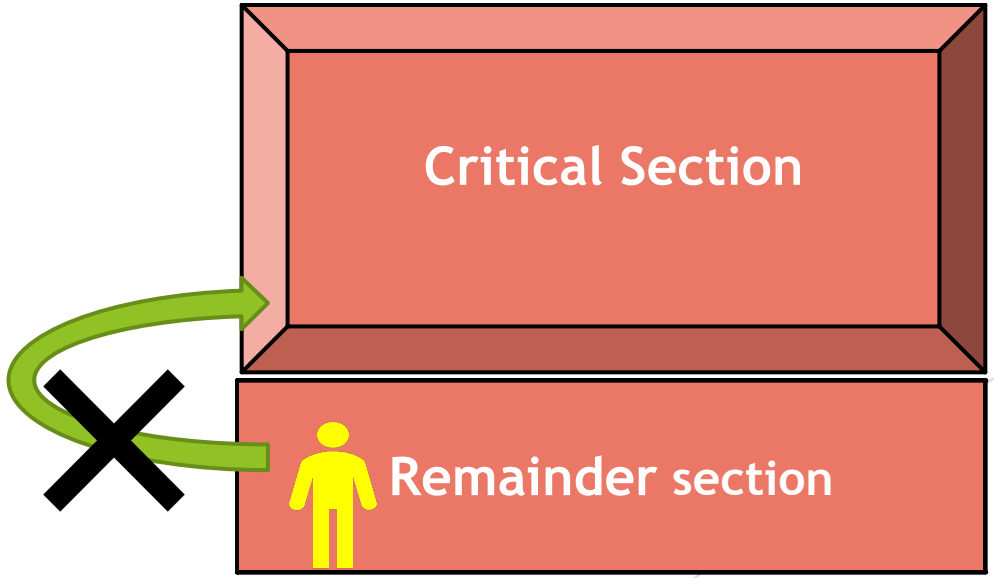




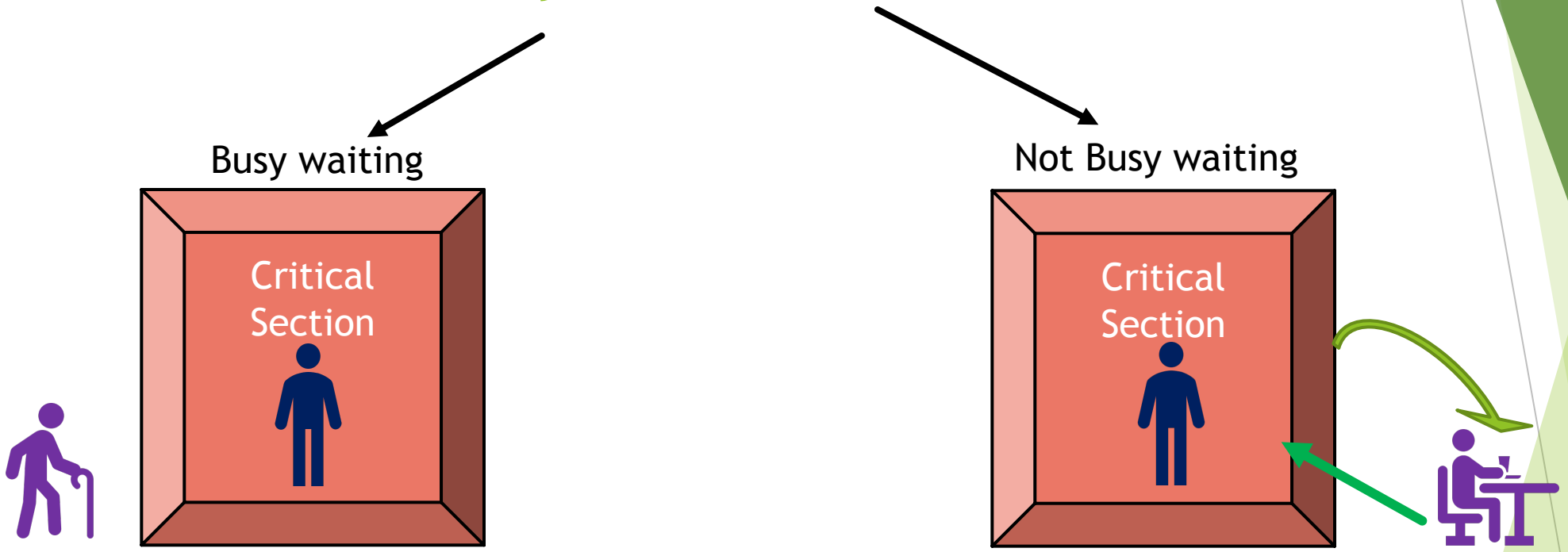
Critical Section Solution requirement

- **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.

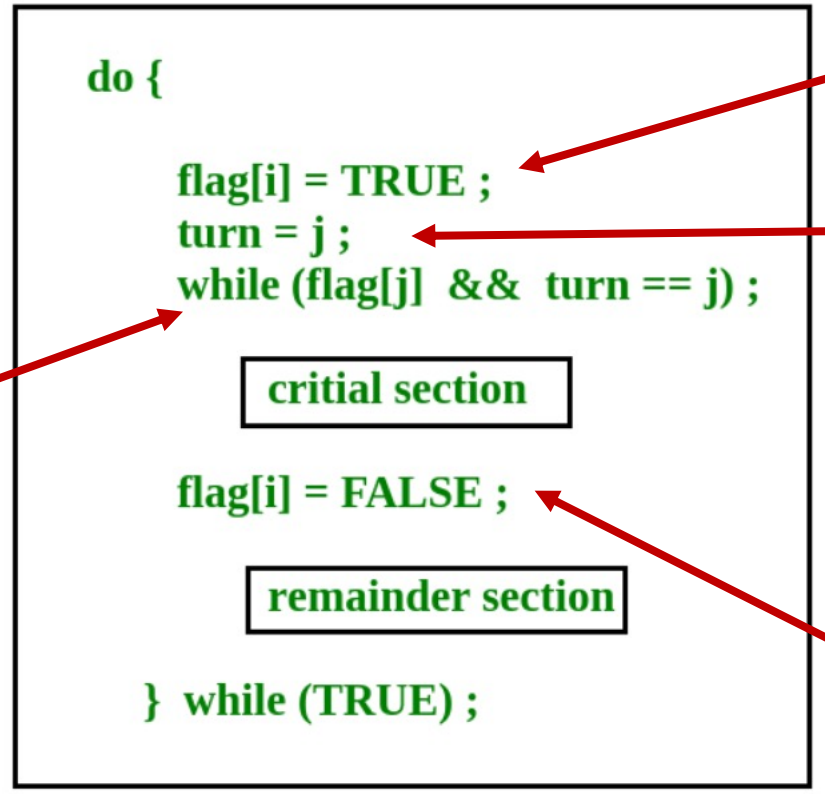
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Critical Section

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

Peterson's Solution

Peterson's Solution



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 at 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 or -ve is locked

>0 is available

Semaphore

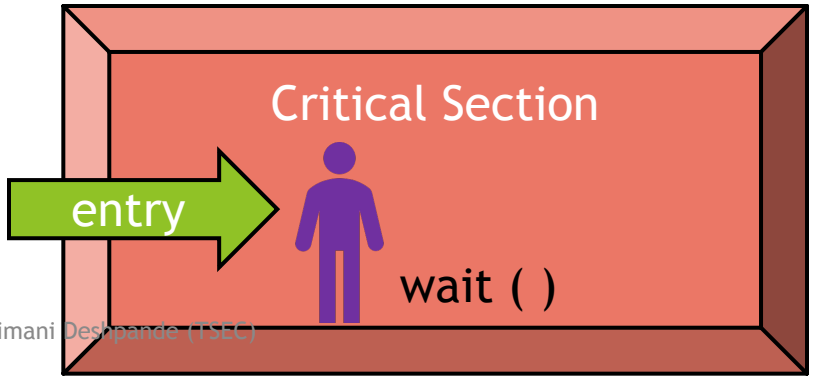

```
wait(s)
{
   while (s<=0) ;   //wait (no operation)
    s--  ;
}
```

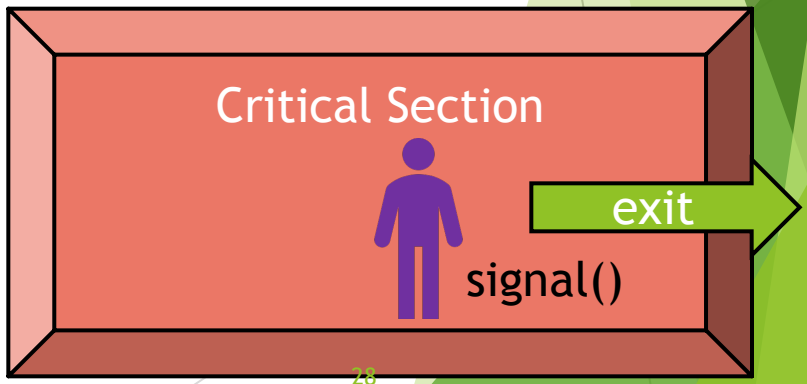
→ when a semaphore is zero it is "locked" or "in use".

→ Positive values indicate that the semaphore is available.

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

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





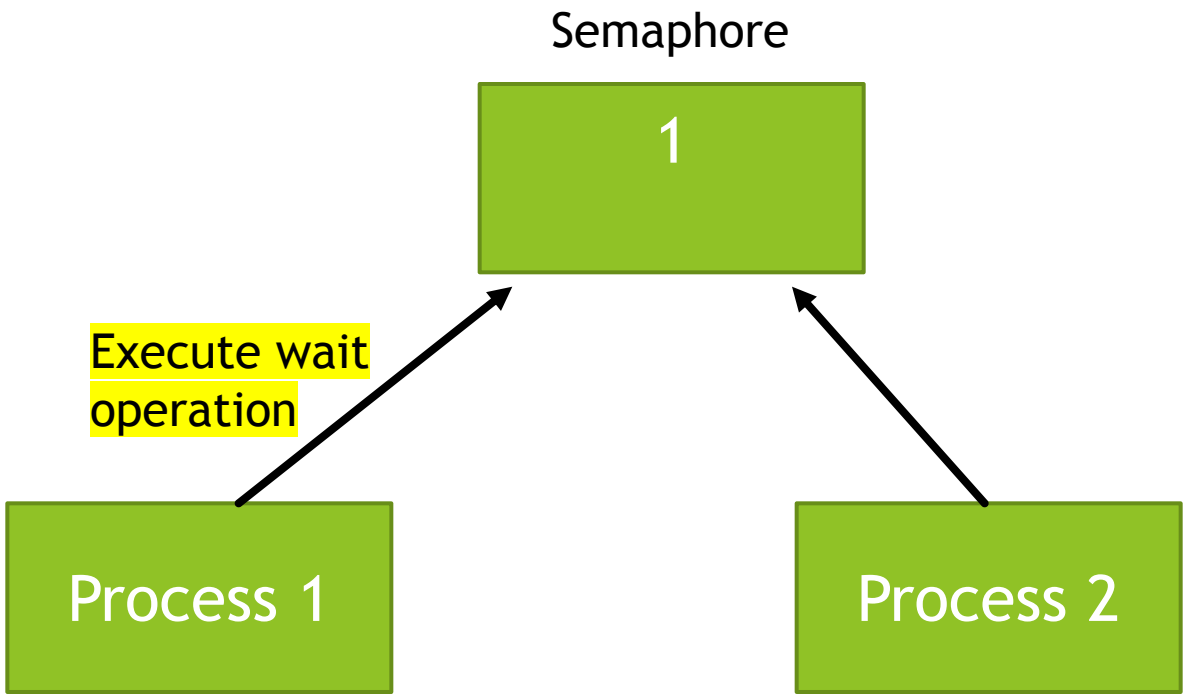
Semaphore

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

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

  S--;
}
```

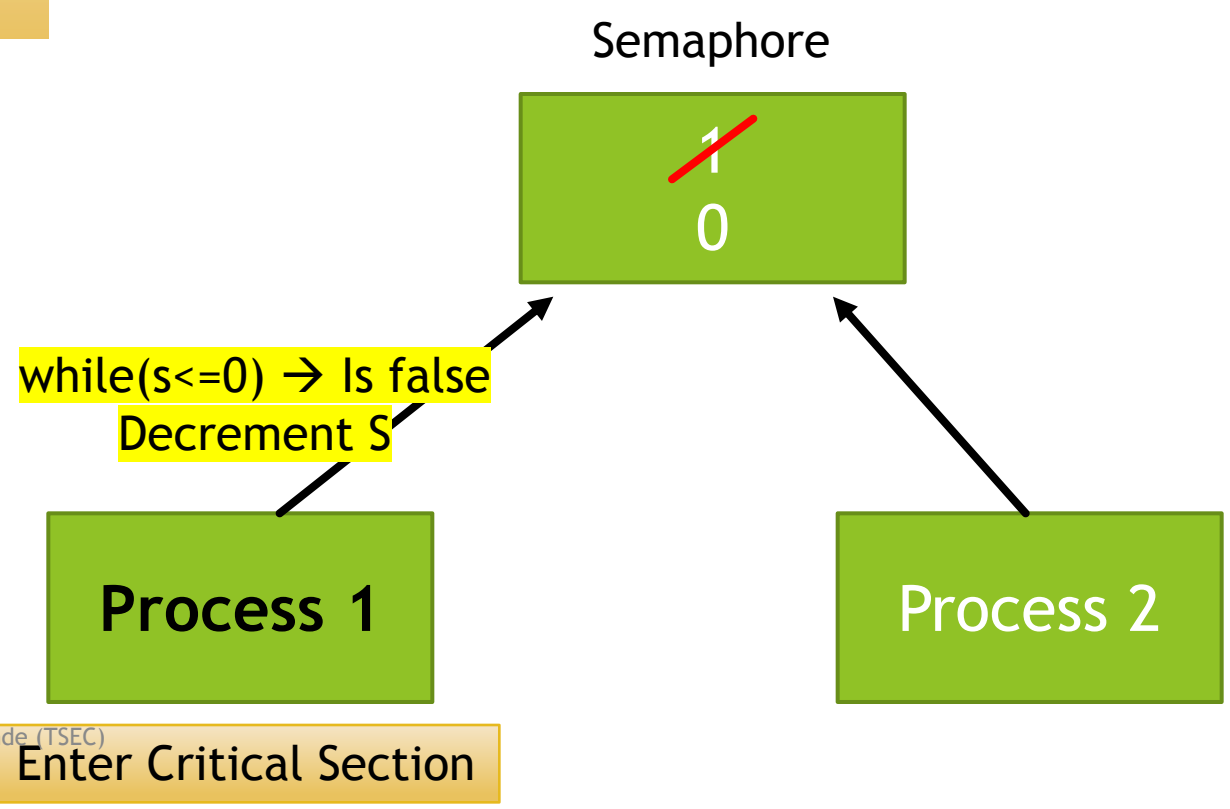
```
signal(S)
{
  S++;
}
```

Semaphore

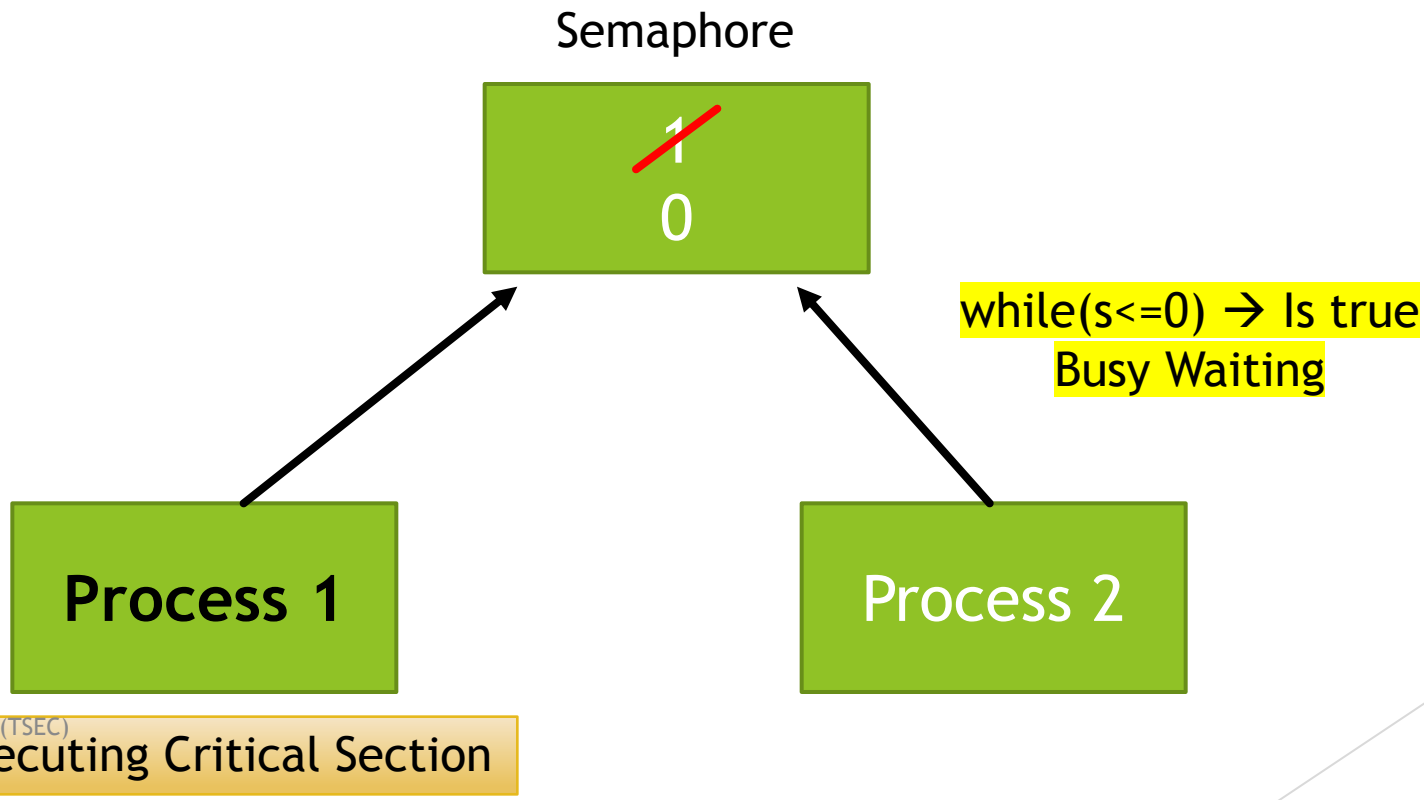


Semaphore

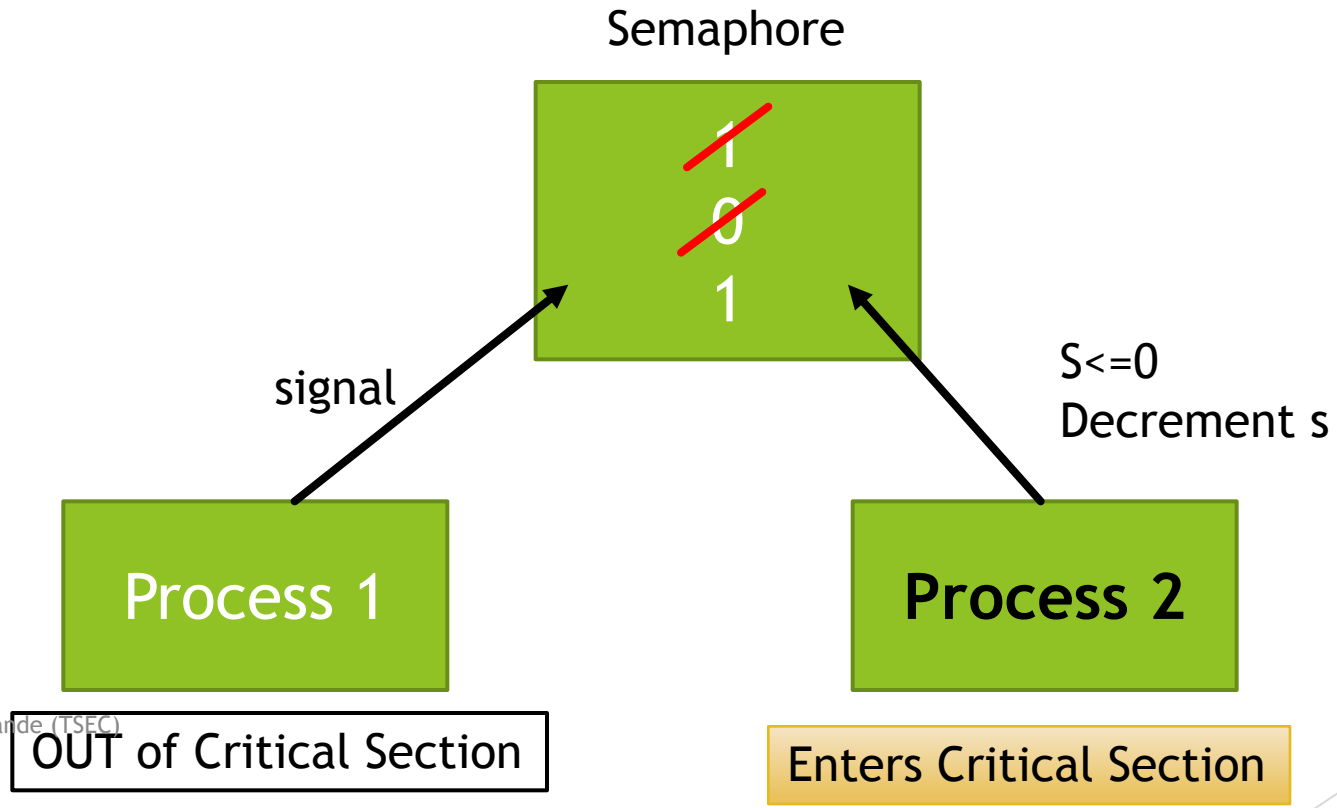
```
wait(s)
{
   while (s<=0) ;
    s--  ;
}
```



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

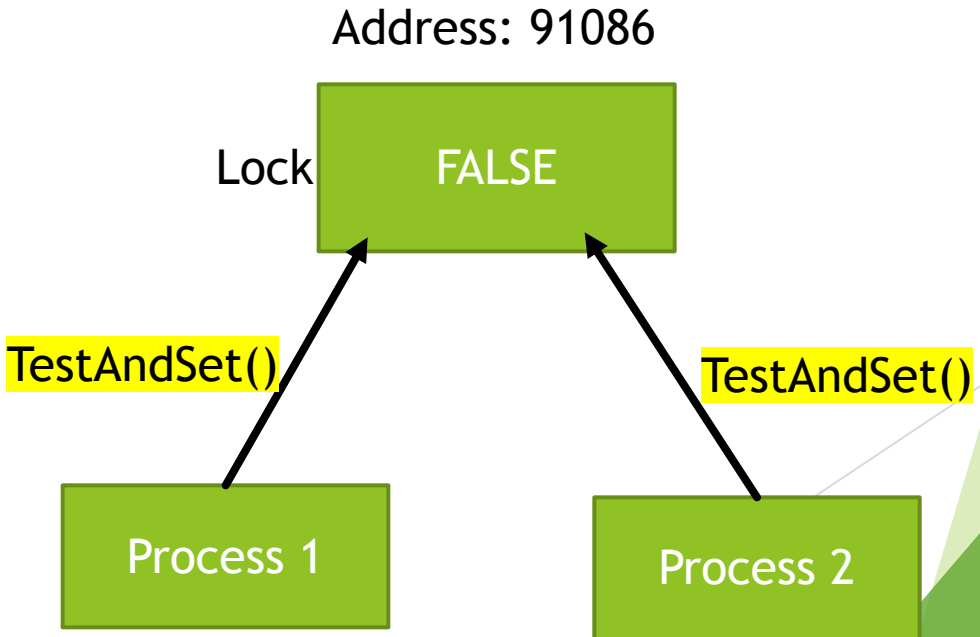
- Shared Boolean variable lock, initialized to False
- TestAndSet instruction is executed Automatically

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

Solution using TestAndSet

```
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

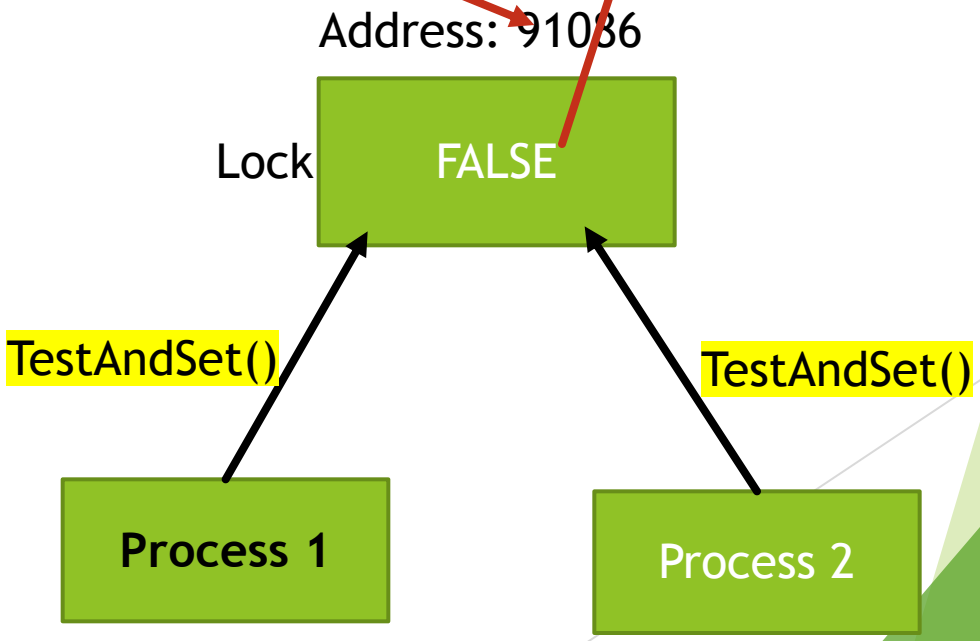
```
boolean  TestAndSet (boolean *target)
      {
          boolean  rv = *target;
          *target = TRUE;
          return  rv;
      }
```

Process 1

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

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

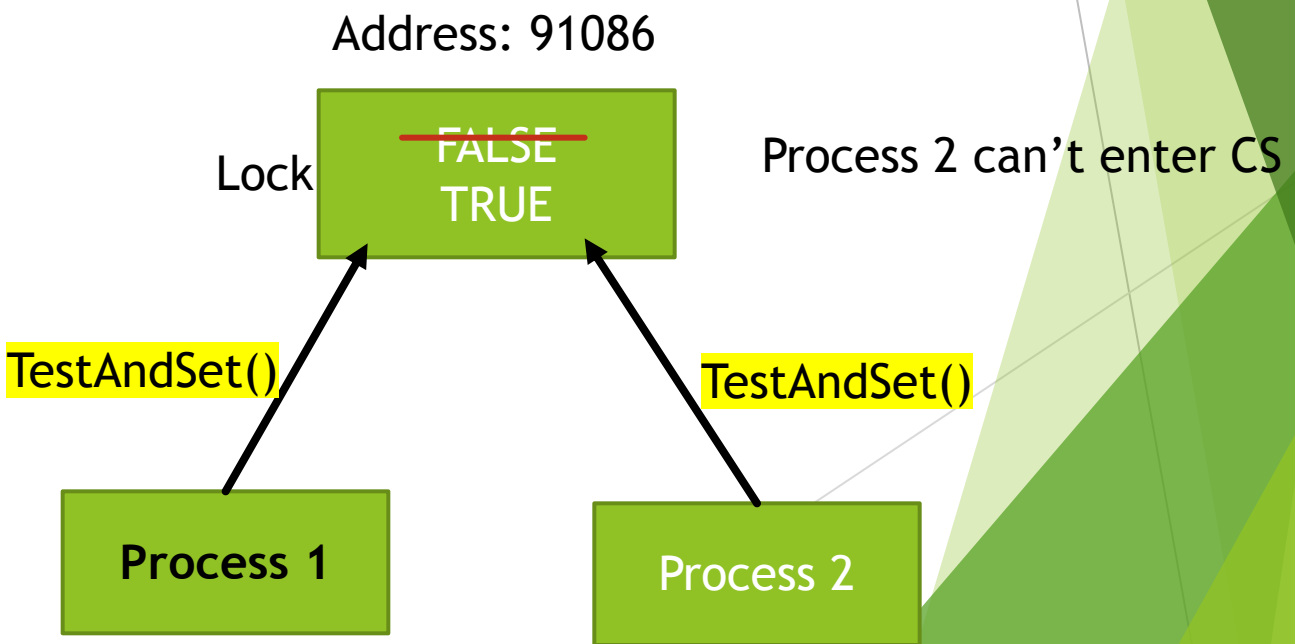
Let's , assume Process 1 execute



Solution using TestAndSet

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

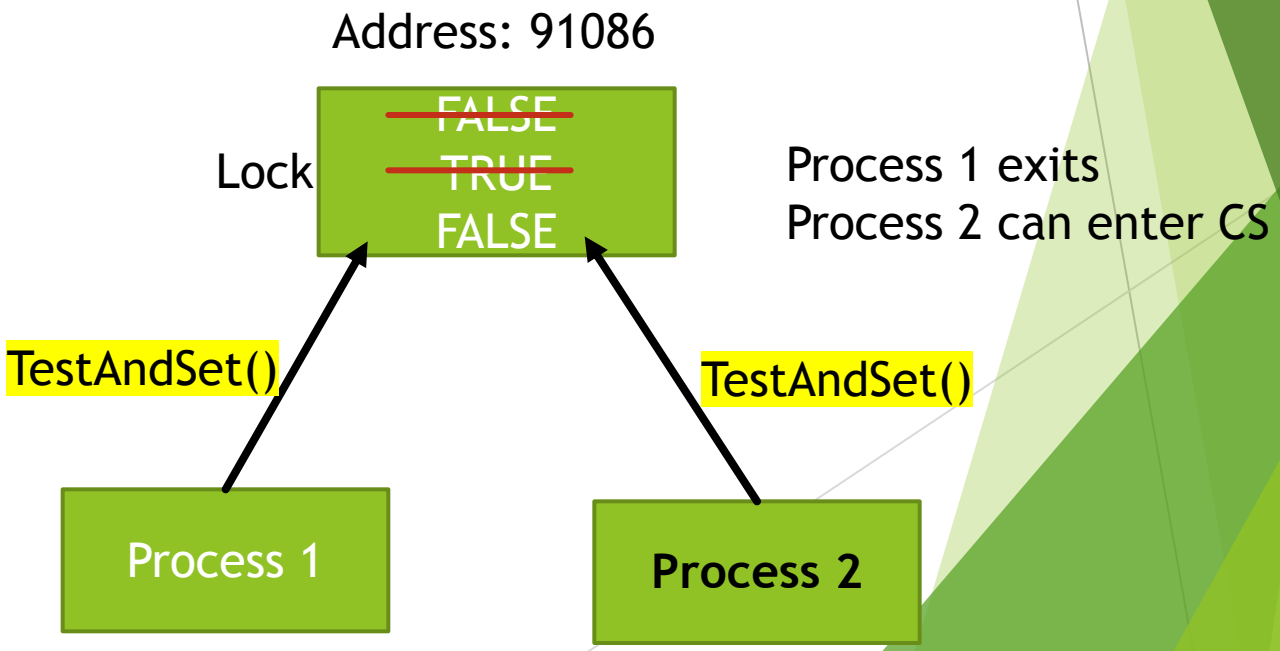
```
do {
          while ( TestAndSet (&lock ))     ;   // do nothing
                //    critical section
          lock = FALSE;
                //     remainder section
     } while (TRUE);
```



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);
```



Swap Instruction

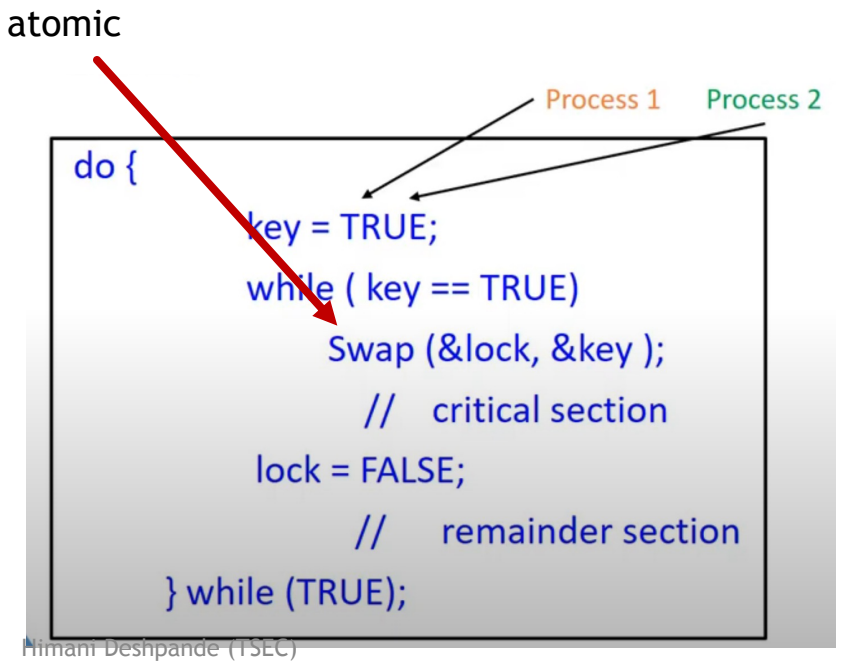
Swap Instruction

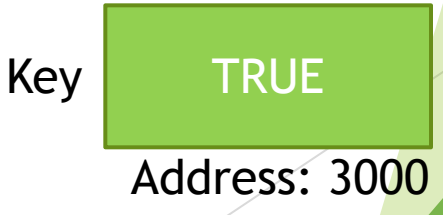
Definition:

```
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





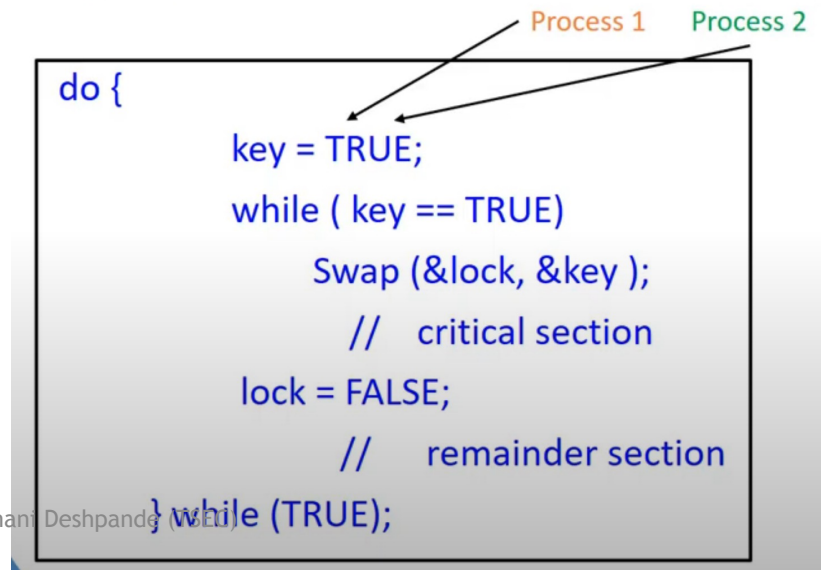
Process 1

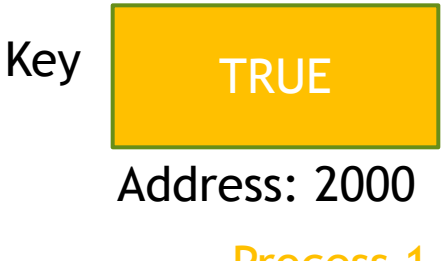
Process 2

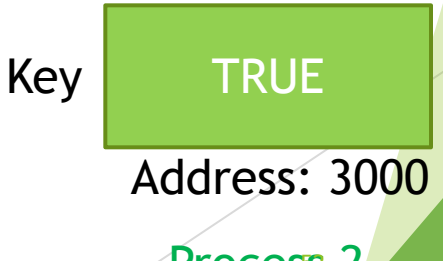
Swap

Assume Process 1 is executing

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



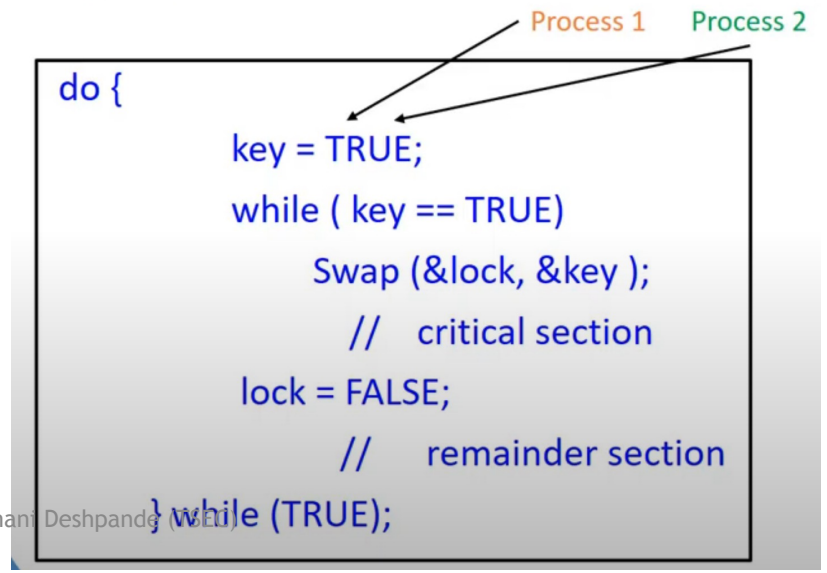


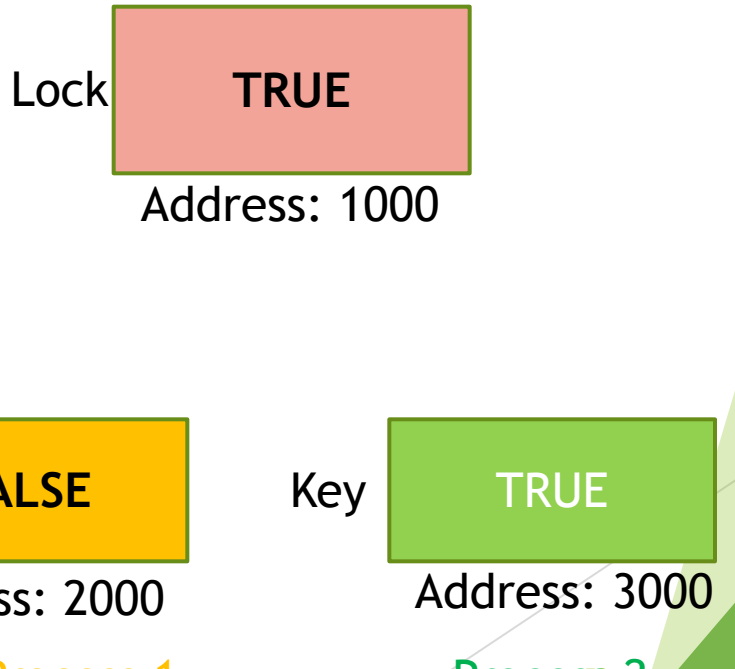


Swap

Process 1 is in Critical Section

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

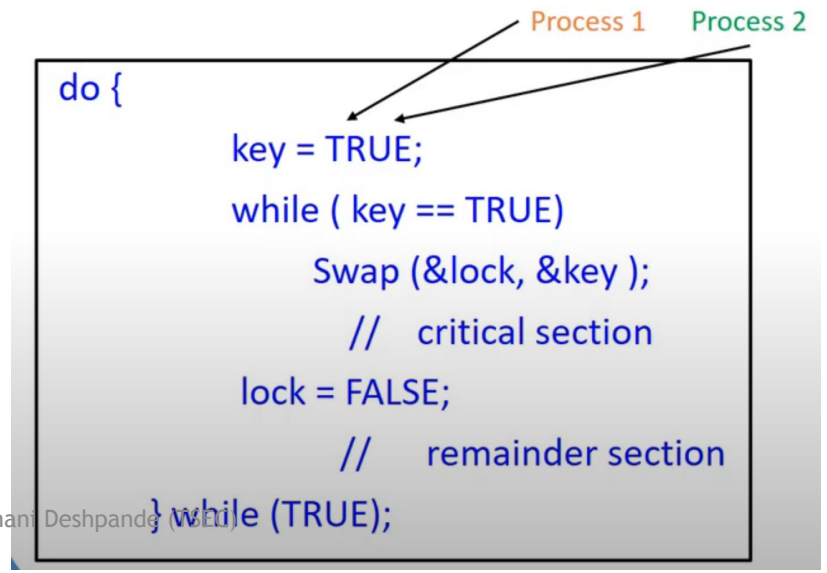


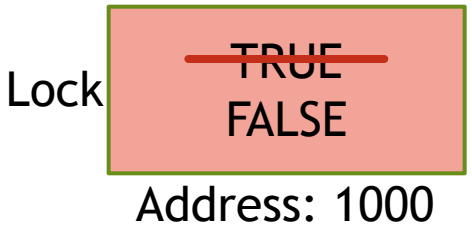


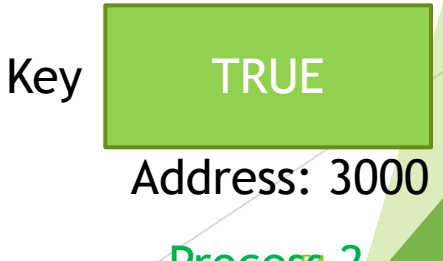
Swap

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



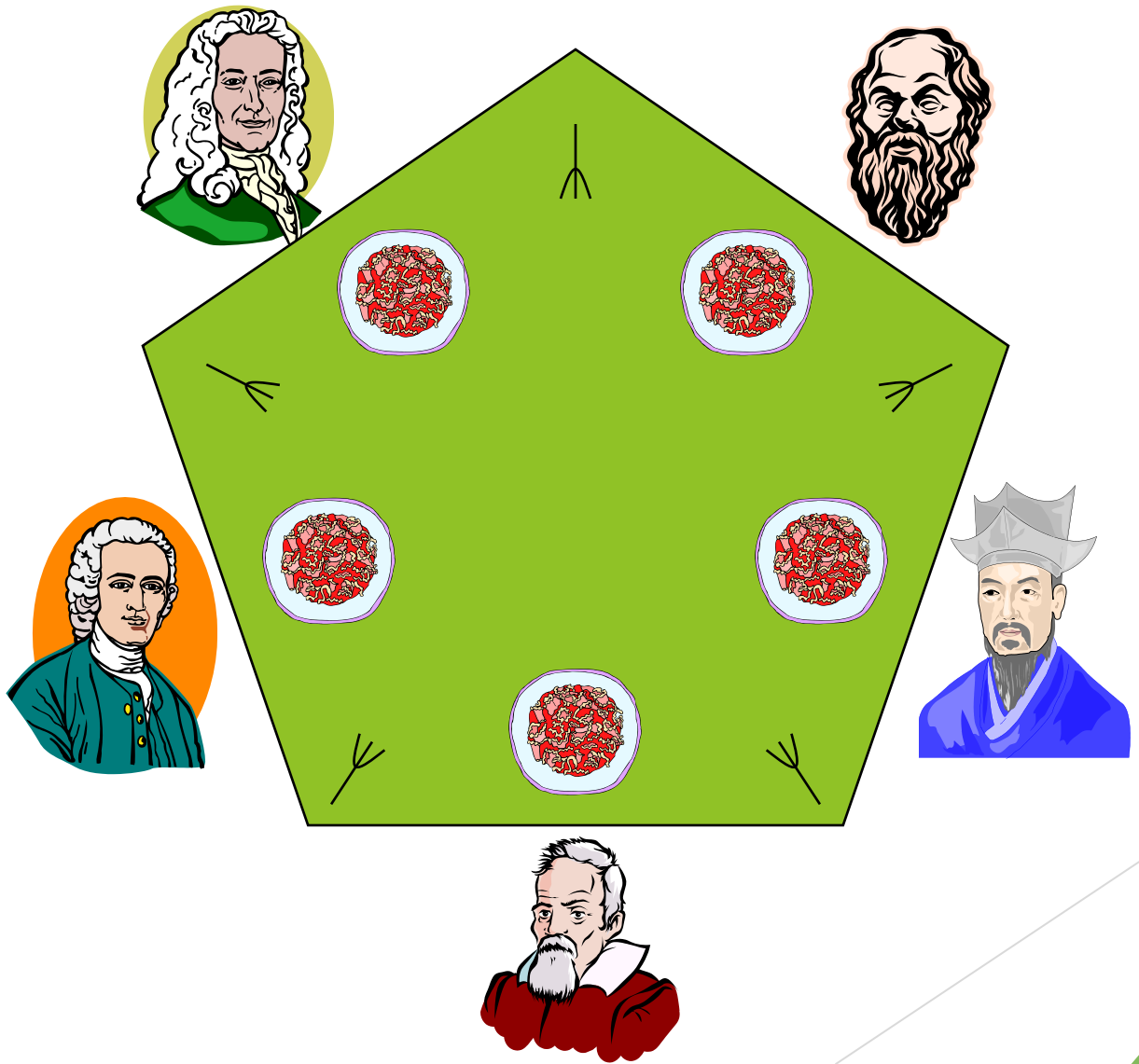




CLASSIC PROBLEM OF SYNCRONIZATION

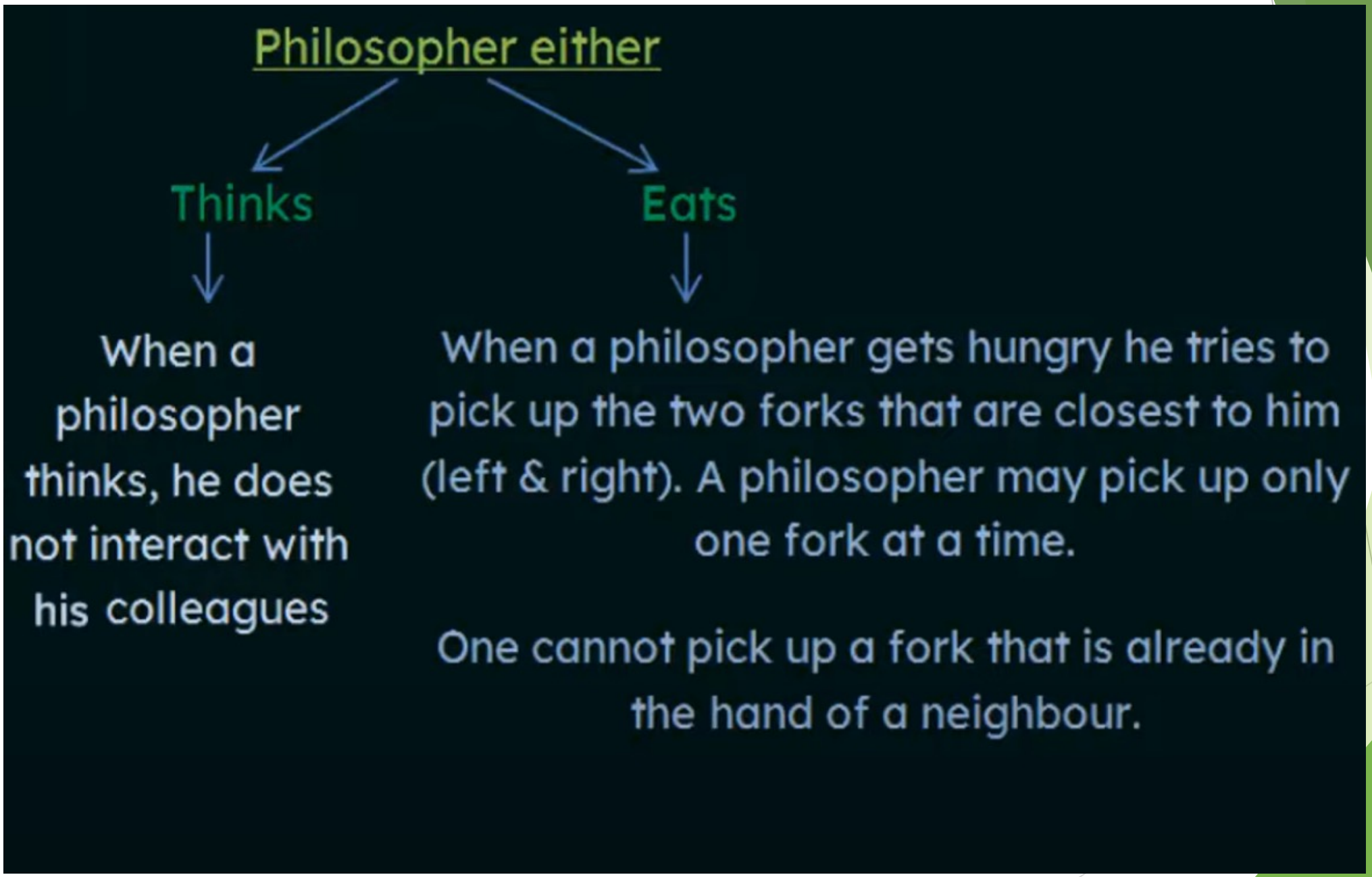
- Bounded-buffer
  (or Producer-Consumer) Problem,
- Dining-Philosphers Problem,
- Readers and Writers Problem,
- Sleeping Barber Problem

Dining Philosopher's Problem



There are three states of the philosopher: **THINKING, HUNGRY, and EATING**.

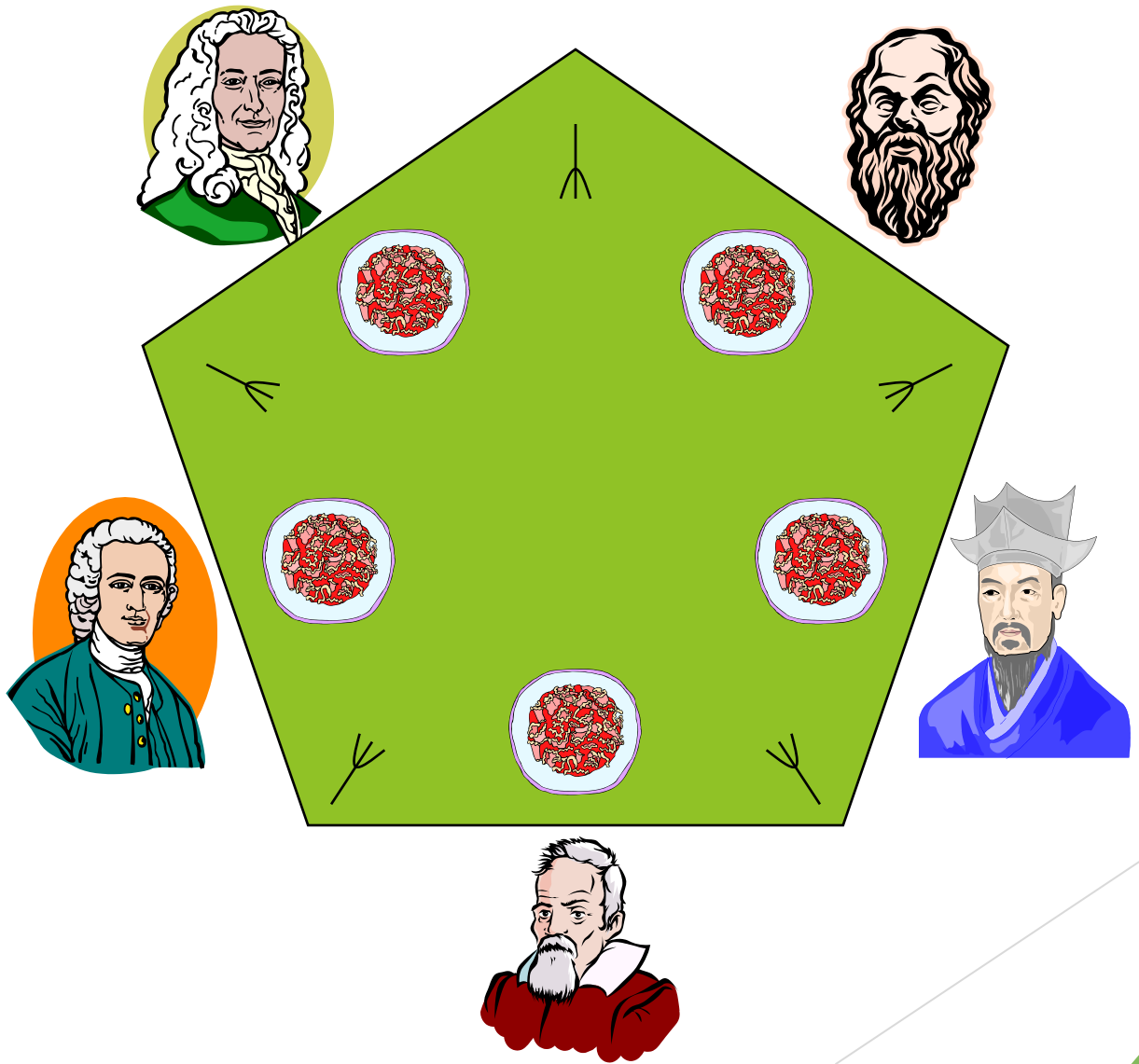
5 Philosopher
5 chop sticks
Each needs two chop sticks to eat


Philosopher either
Thinks
Eats
When a philosopher thinks, he does not interact with his colleagues
When a philosopher gets hungry he tries to pick up the two forks that are closest to him (left & right). A philosopher may pick up only one fork at a time.
One cannot pick up a fork that is already in the hand of a neighbour.

Dining Philosopher's Problem

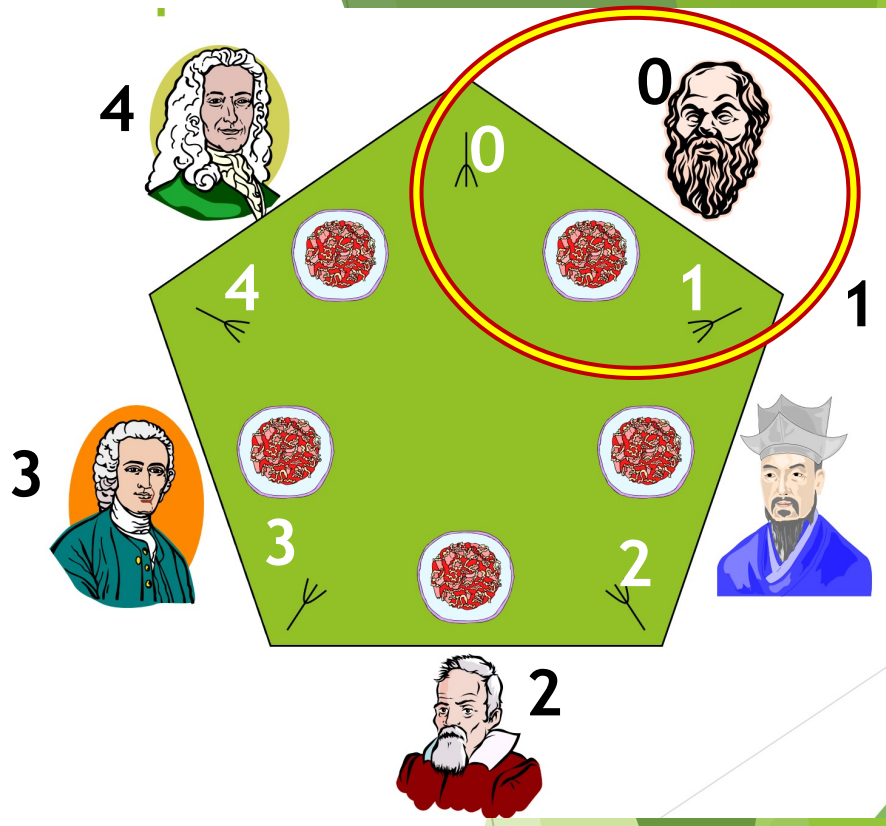
PROBLEMS

Starvation & Deadlock



There are three states of the philosopher: **THINKING, HUNGRY, and EATING**.

For Philosopher's 0




```
process P[i]
while (true)
{
      THINK;
      wait(chopstick[0]);
      wait (chopstick[1]);
      EAT;
      signal(chopstick[0]);
      signal(chopstick[1]) ;
}
```

Left chop stick

Right chop stick

Mutex and a semaphore array for the philosophers.

Mutex is used such that no two philosophers may access the pickup or putdown at the same time.

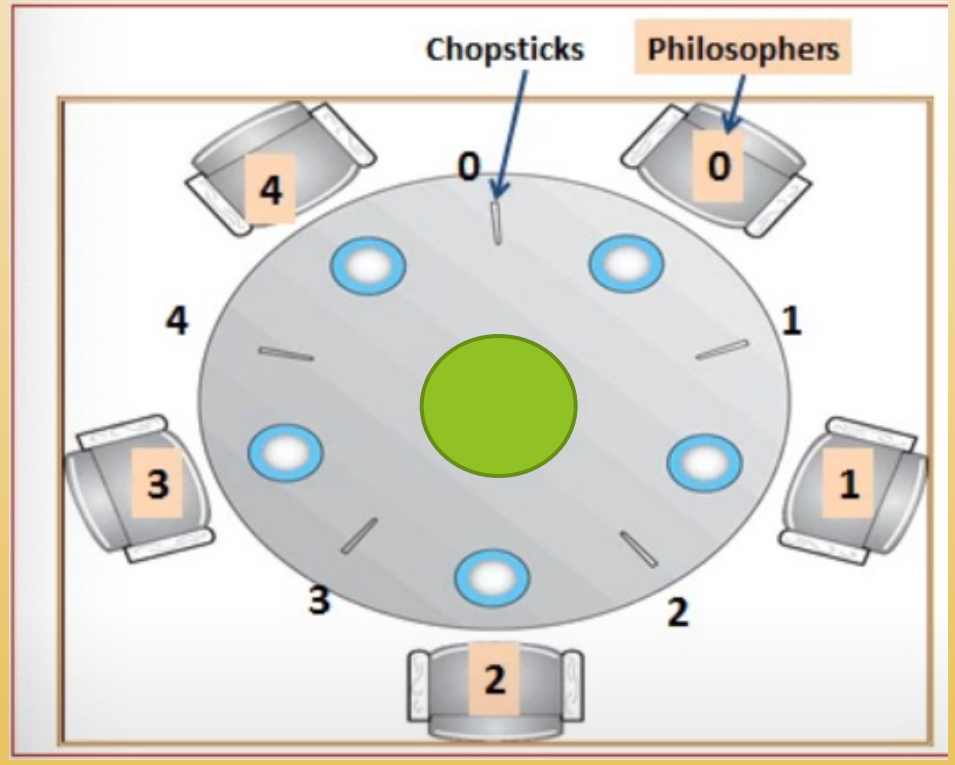
The array is used to control the behaviour of each philosopher.

For Philosopher P[i]

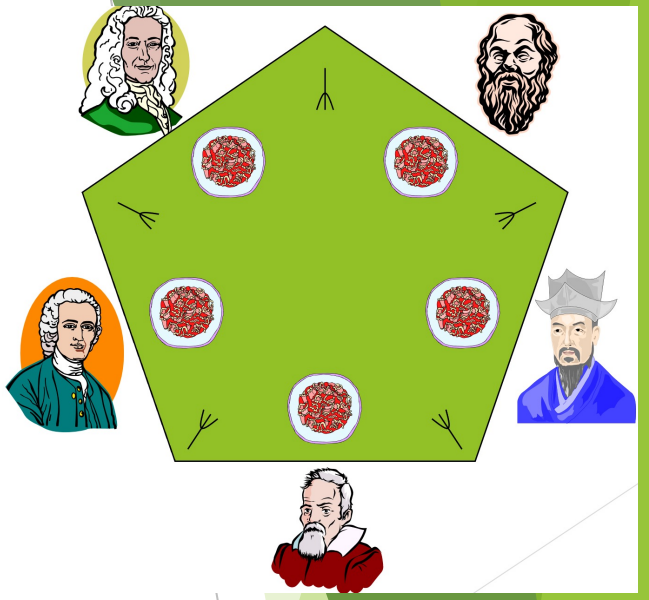
n = no of philosphers

```
semaphore chopstick[5] ;
```

```
process P[i]
while (true)
{
      think;
      wait(chopstick[i]);
      wait (chopstick[(i+1)%n]);
      eat;
      signal(chopstick[i]);
      signal(chopstick[(i+1)%5]) ;
}
```



Dining Philosophers: Solutions



- Simple: "waiting" state
  - Enter waiting state when neighbors eating
  - When neighbors done, get forks

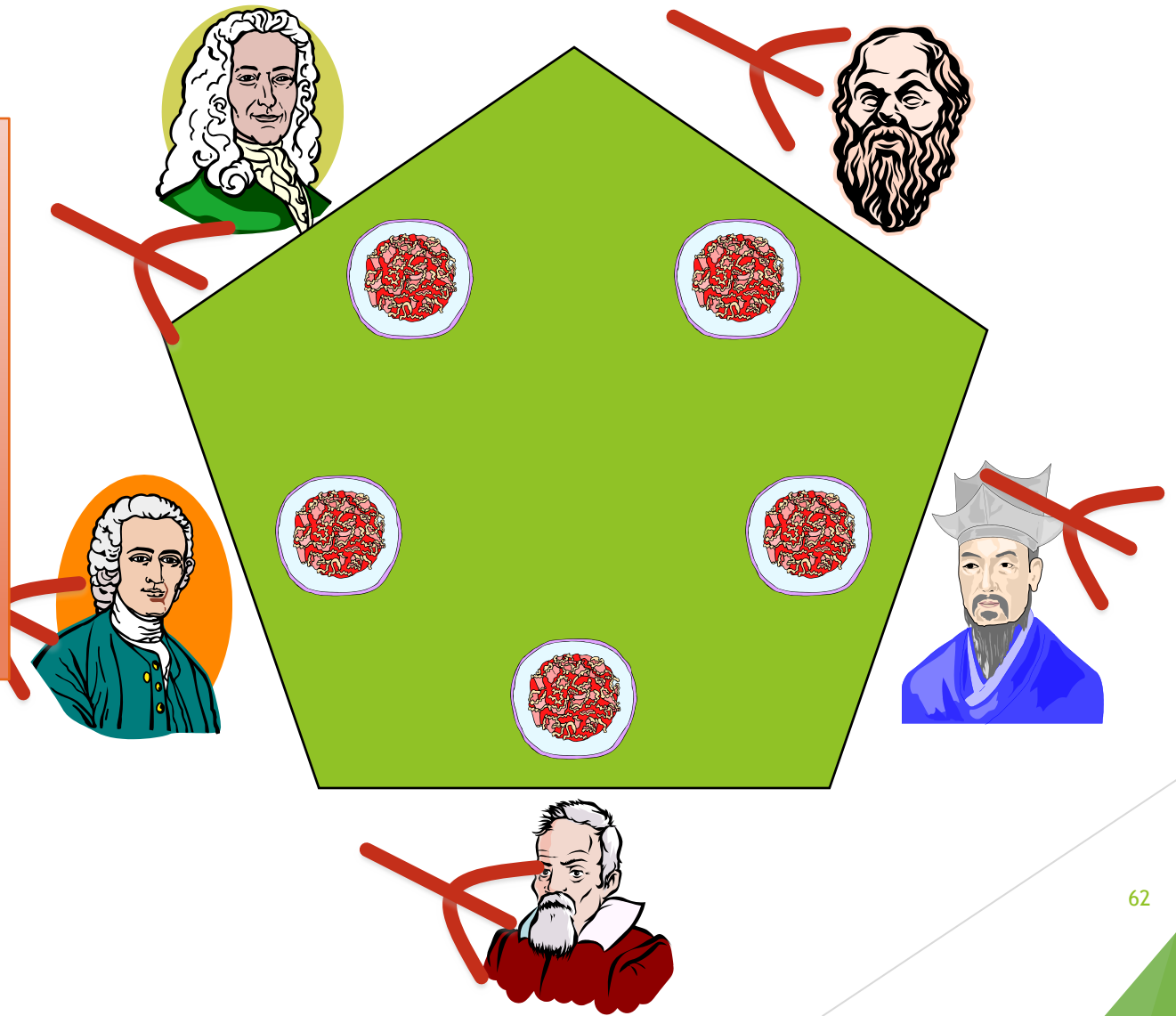
Dining Philosopher's Problem

PROBLEMS

Deadlock

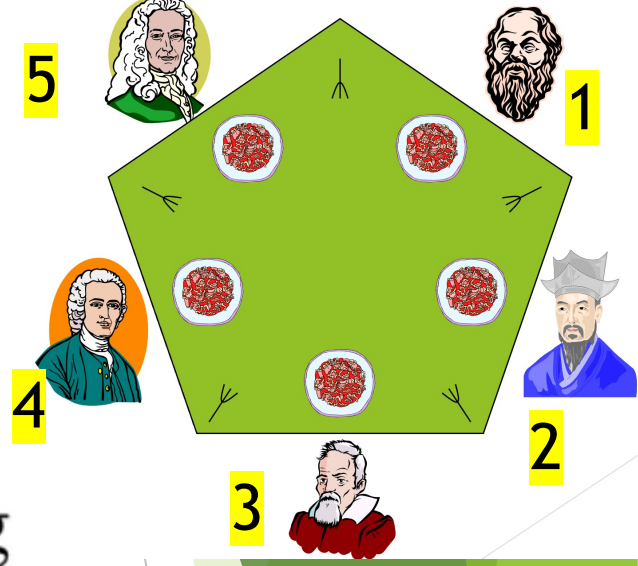
&

Starvation



Every Philosopher picks left chopstick

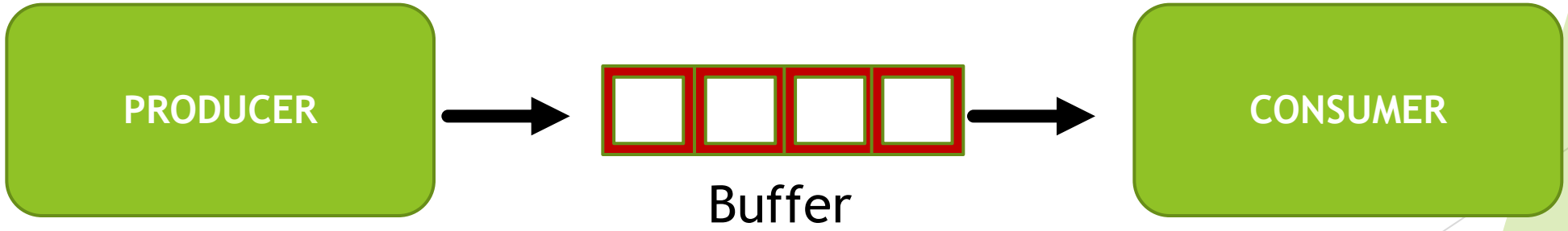
More Solution to Philospher's Deadlock



- Allow at most four philosophers to be sitting simultaneously. (One will get two chopsticks)
- An odd number philosopher picks up first her left chopstick and then her right chopstick.
- Even number philosopher picks up her right .. Then left

Bounded Buffer Problem

- Bounded Buffer problem is also called producer consumer problem.
  - Producers produce a product and
  - Consumers consume the product,
  - Both shares the buffer.



Bounded Buffer Problem

The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full.
  - If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty.
  - If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

Bounded Buffer Problem

- Solution to this problem is,
  - creating two counting semaphores “full” and “empty”
  - to keep track of the current number of full and empty buffers respectively.

Reader and Writers PROBLEM

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.

VS

Reader enter into critical section

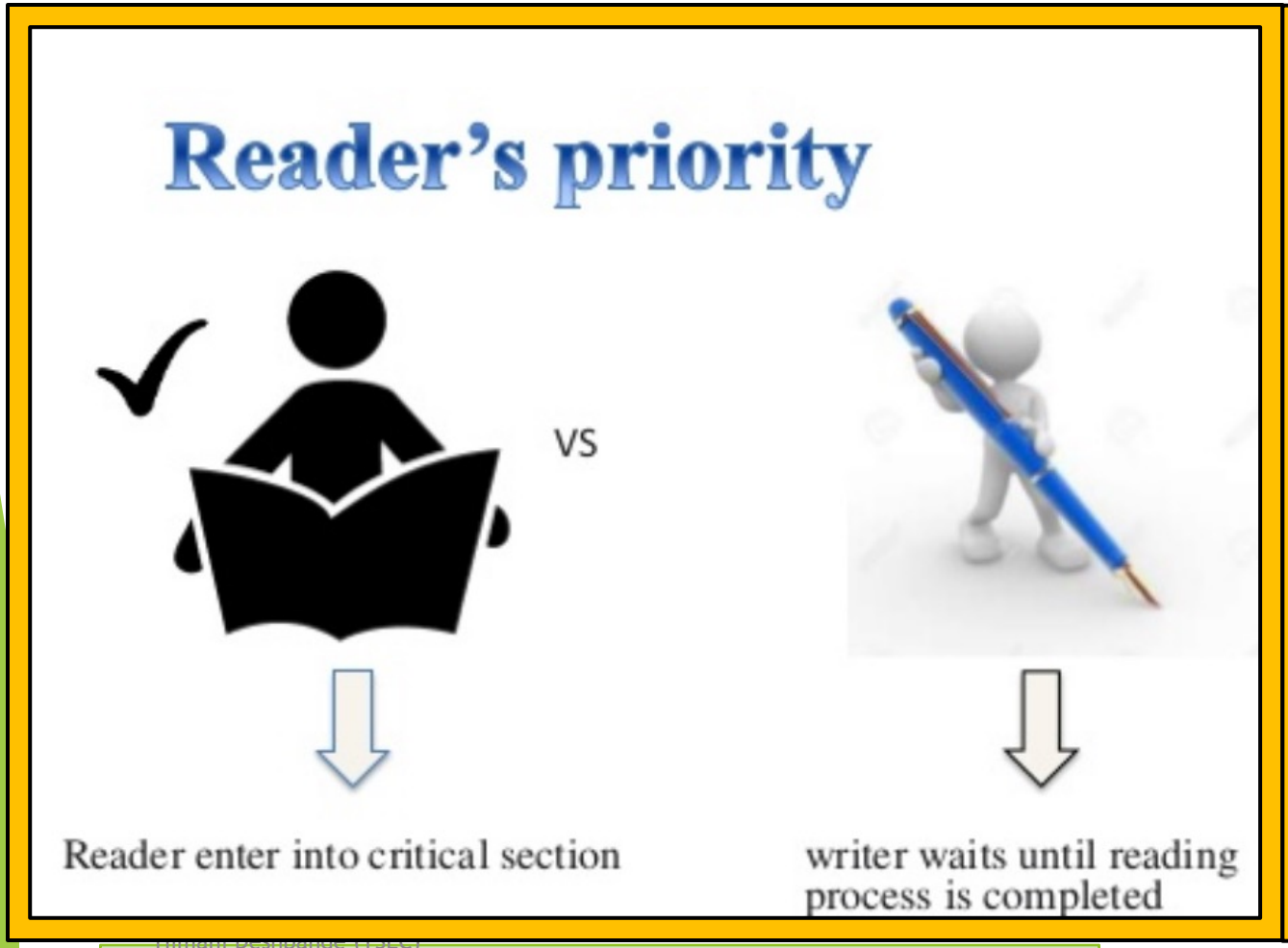
writer waits until reading process is completed

Problem parameters:

- One set of data is shared among a number of processes.
- Once a writer is ready, it performs its write.
- Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

Process 1	Process 2	Issue
Read	Write	Problem
Write	Read	Problem
Write	Write	Problem
Read	Read	No Problem

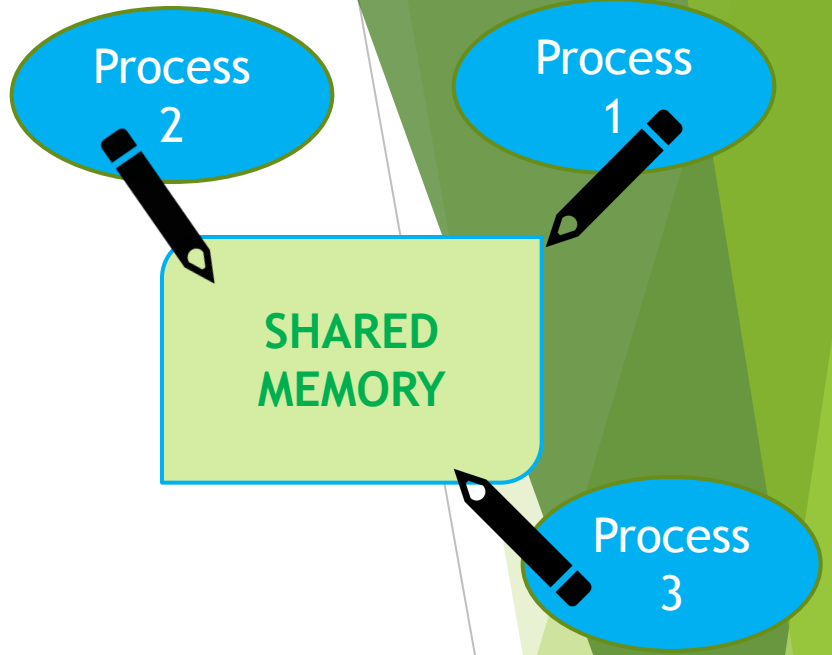
Reader writer Problem- Use semaphores



N readers can read simultaneously

N writers can write sequentially.

Inter-process Communication



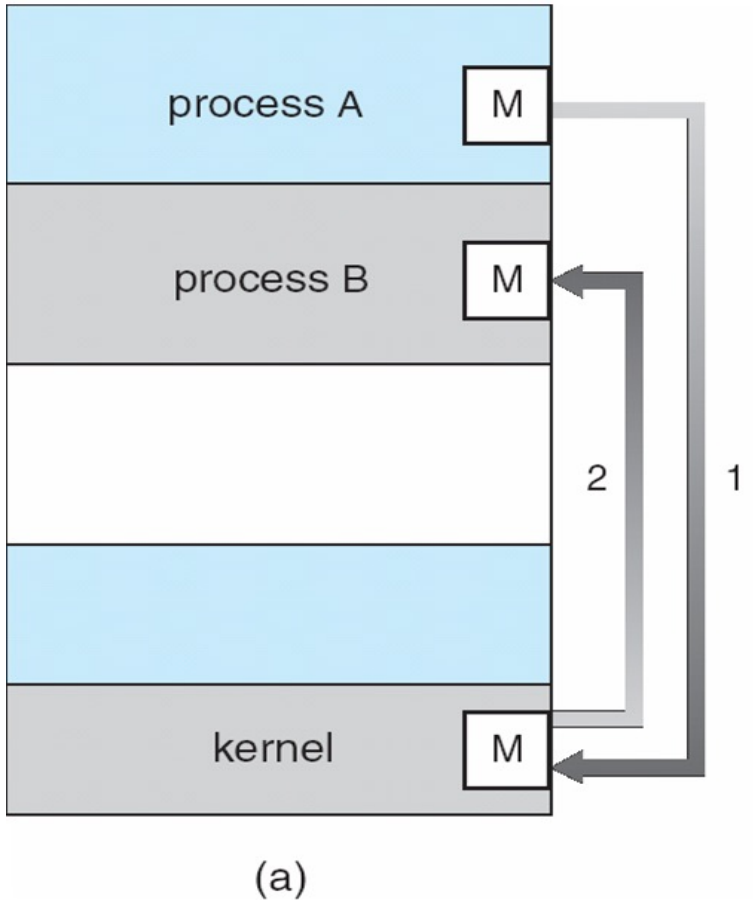
Shared Memory

→ A process can communicate with other by writing to a common variable

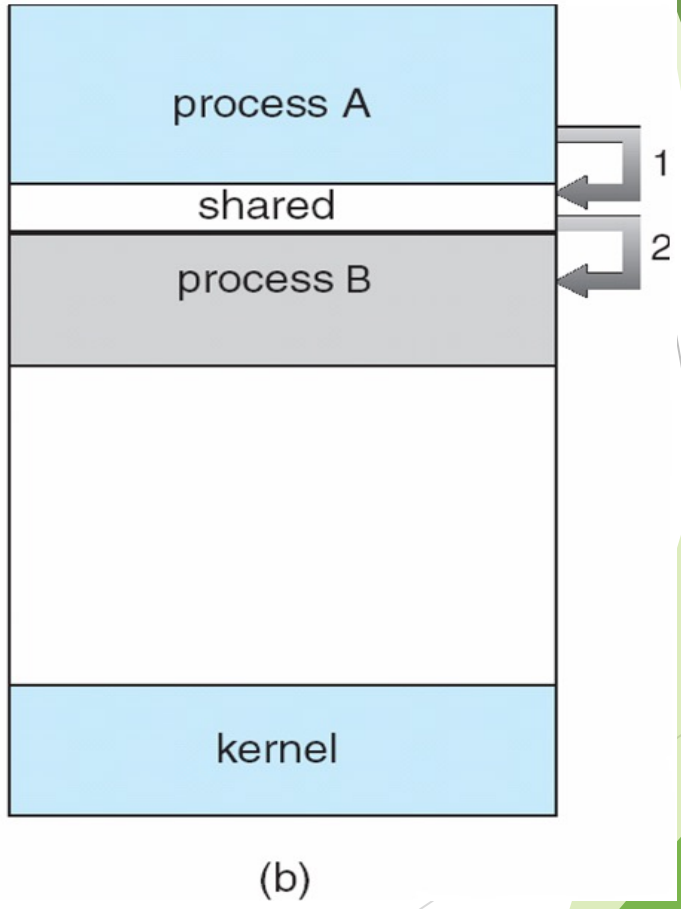
Message Passing

→ A process can communicate with another by an explicit send operation.

Communications Models



Message Passing



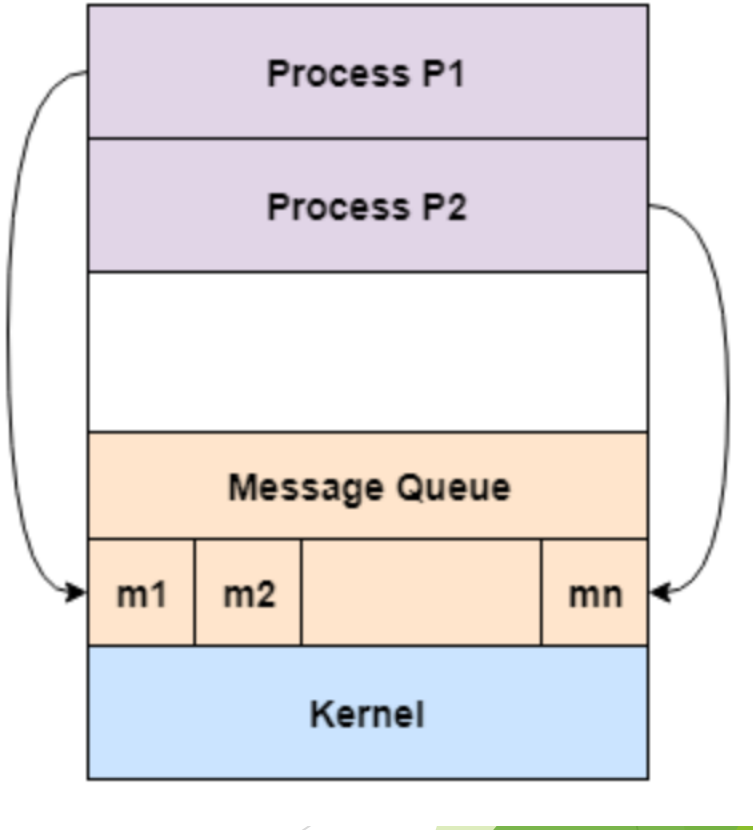
Shared Memory

Message Passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

Message Passing

- Message passing model allows multiple processes to read and write data to the message queue without being connected to each other.
- Messages are stored on the queue until their recipient retrieves them.
- Message queues are quite useful for interprocess communication and are used by most operating systems.



Message Passing

- IPC (Interprocess Communictaion) provides two operations:
  - send (message)
  - receive (message)
- The size of message could be either fixed or variable.
- If process P and Q wish to communicate , they need to
  - Establish a communication link between them
  - Exchange message via send/receive

Message passing- implementation

- Implementation of communication link
  - Physical
    - Shared memory
    - Hardware bus
    - Network
  - Logical
    - Direct or indirect communication

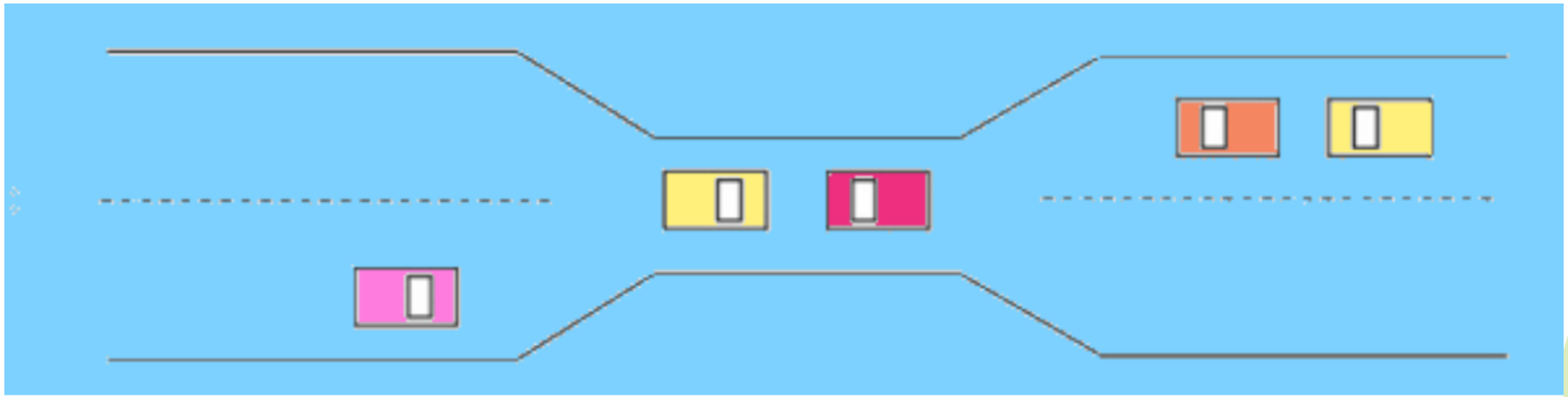
Direct communication

- Process must name each other explicitly :
  - Send(P,message)
  - Receive(Q,message)
- Properties of direct link:
  - Links are established automatically.
  - Between each pair there exists exactly one link.
  - Link can be unidirectional or bi directional

Indirect Communication

- Messages are directed and received from mailbox (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of link
  - Link establishes only if processes share a common mailbox
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link can be unidirectional or bi directional

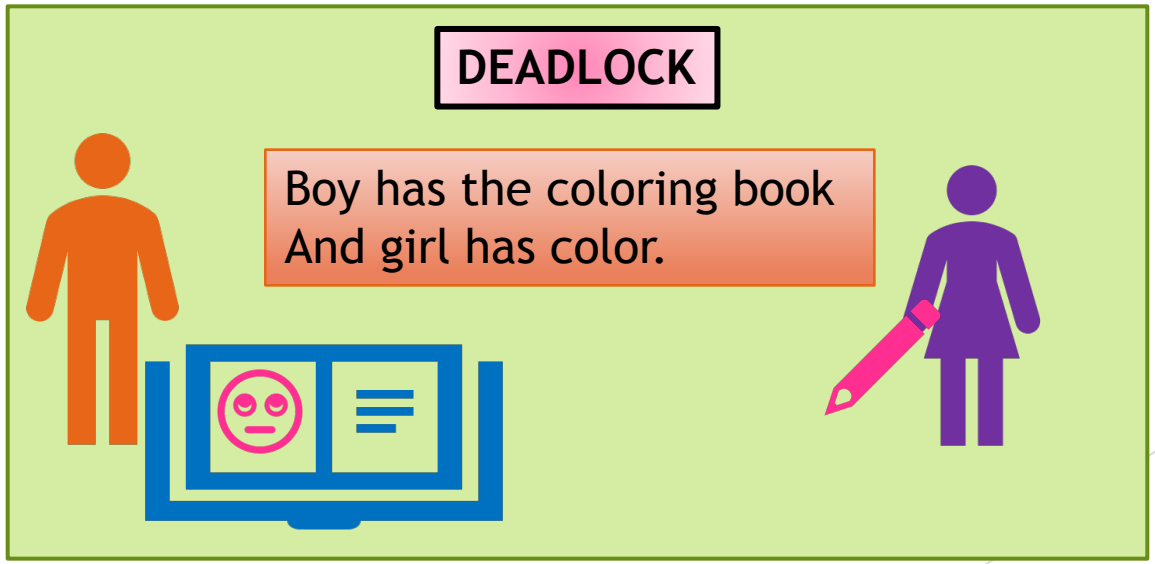
DEADLOCK



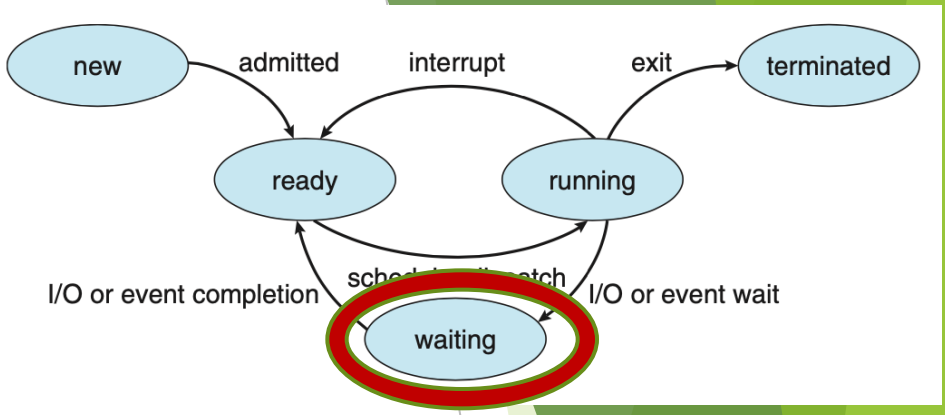
Stuck up

The Deadlock problem

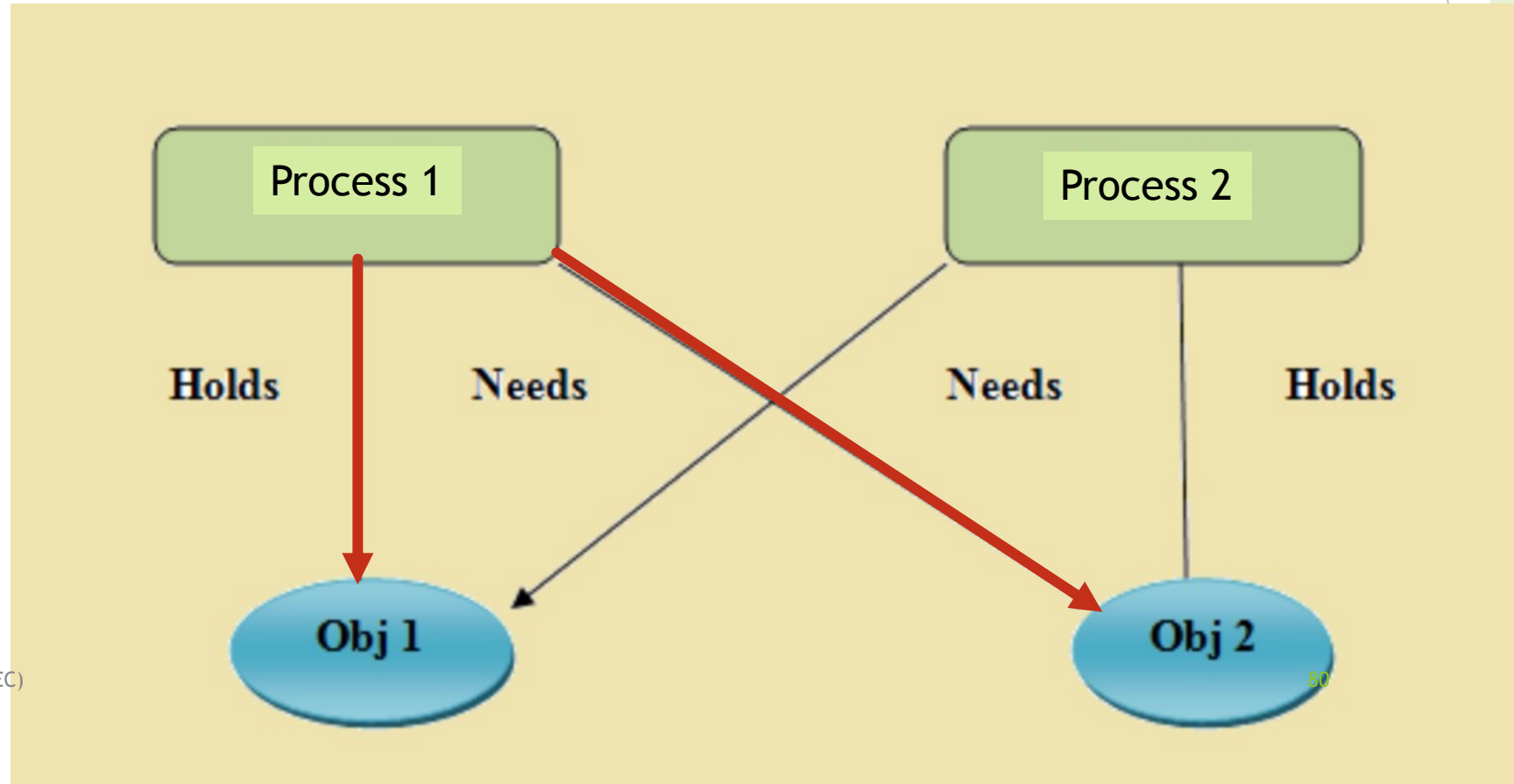
- In a computer system deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group.
- Waiting for something for infinite time with “no progress” for waiting process.



Deadlock

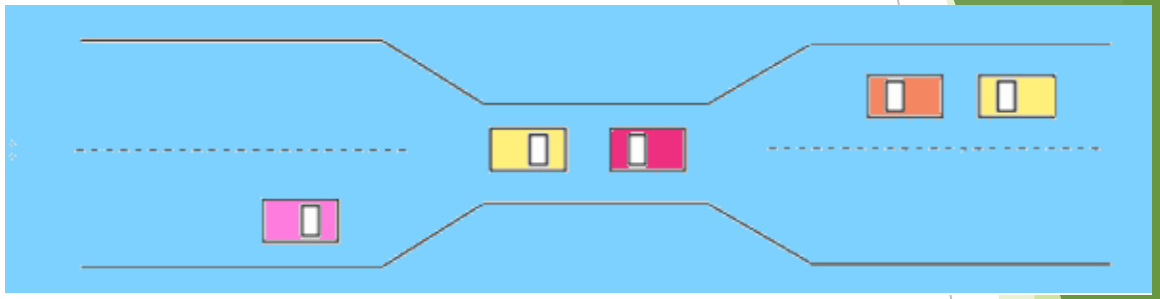


Deadlock occurs when two or more processes are blocked forever, waiting for each other's resources.



Starvation Vs Deadlock

- Deadlock jams the whole system. CPU is not running any process.



- Starvation make few process starve , while CPU is given to high priority processes.

Conditions for deadlocks

1) Mutual exclusion.

No resource can be shared by more than one process at a time.

2) Hold and wait.

There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

Conditions for deadlocks (cont.)

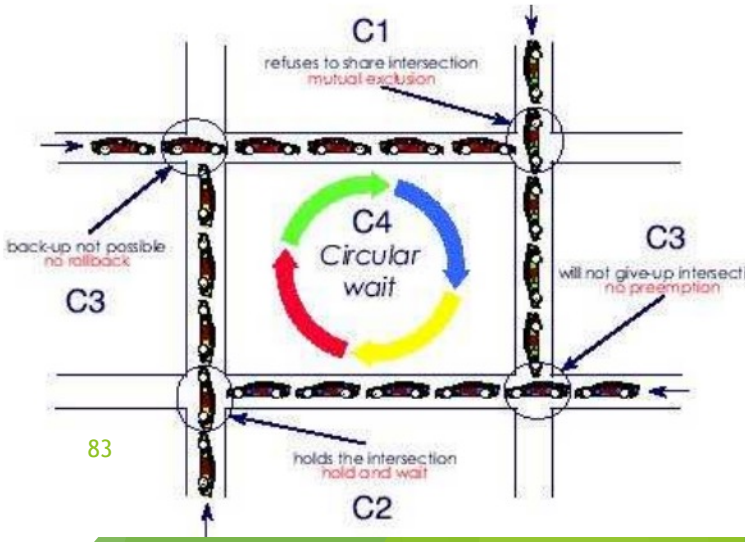
3)No preemption.

A resource cannot be preempted.
previously granted resources cannot be forcibly taken away

4) Circular wait.

There is a cycle in the wait-for graph.

i.e.
There must be a circular chain of 2 or more processes and Each is waiting for resource held by next member of the chain



Deadlock Detection Algorithms

Single Instance

Single instance of resource

(Wait for graph)

Multiple Instance

Multiple instance of resource

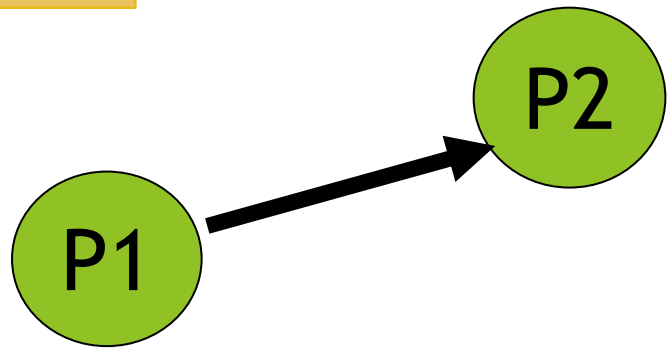
(Banker's. algorithm)

Graph-theoretic models

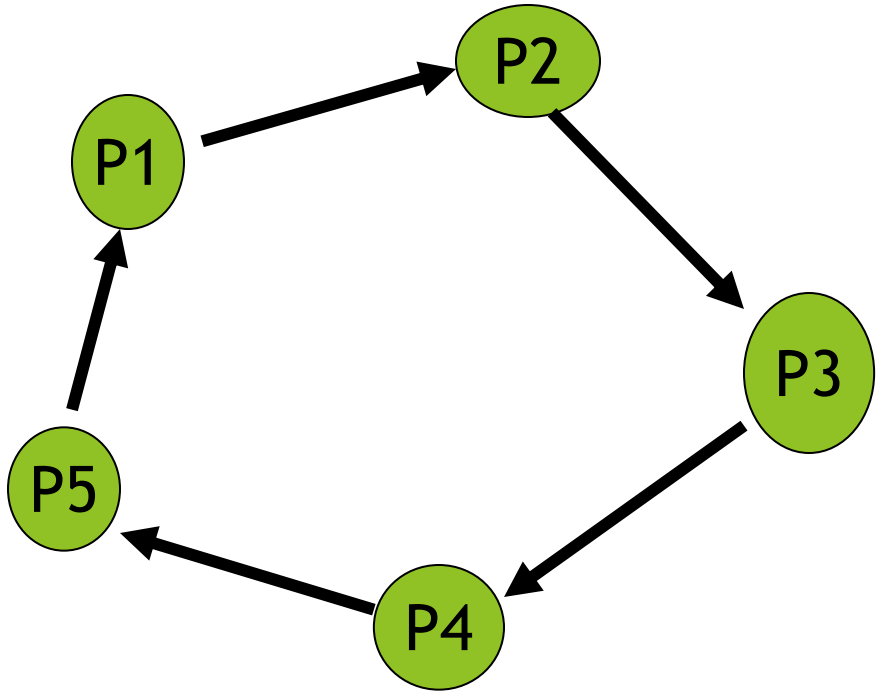
- Wait-for graph.
- Resource-allocation graph.

Wait-for graph

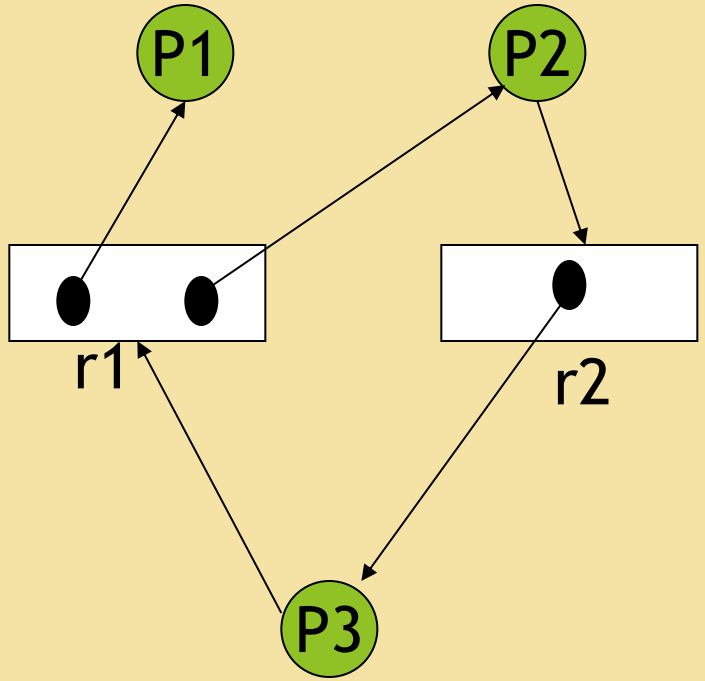
P1 is waiting for R
R is held by P2



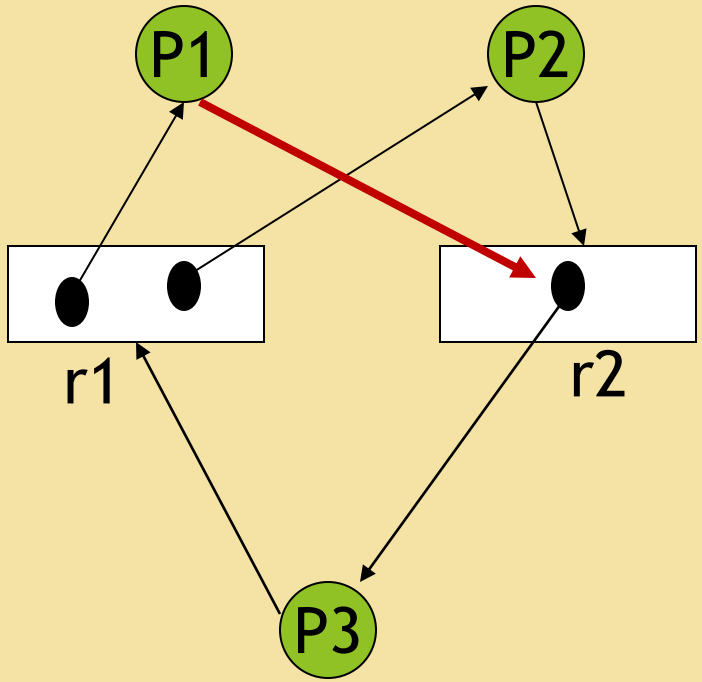
Wait-for graph



Resource allocation graph



Resource allocation graph
Without deadlock

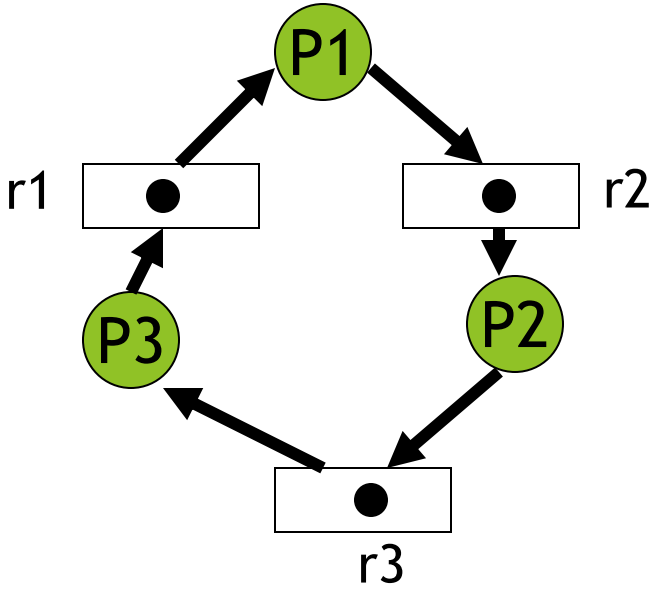


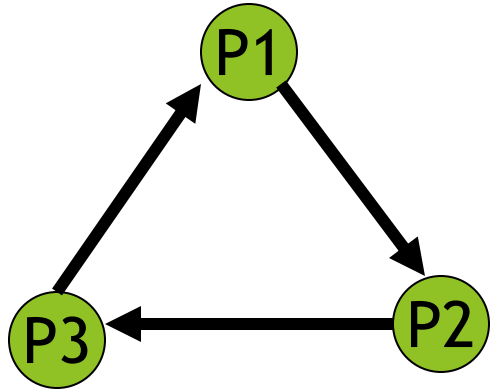
With deadlock

Wait-for graph and Resource-allocation graph conversion

- Any resource allocation graph with a single instance of resources can be transferred to a wait-for graph.

A cycle in Wait-for-graph denotes Deadlock.





Banker's Algorithm

- Banker's Algorithm is used to determine whether a process's request for allocation of resources be safely granted immediately.

Or

- The grant of request be deferred to a later stage.

Banker's Algorithm

•For the banker's algorithm to operate, each process has to a priori specify its **maximum requirement of resources**.

• A process is admitted for execution only if its maximum requirement of resources is within the system capacity of resources.

The Banker's algorithm is an example of resource allocation policy that avoids deadlock.

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	4	6	4	4
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

1. Compute NEED Matrix.
2. Is the system in safe state? Justify.

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	4	6	4	4
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	4	6	4	4
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

P1 completes execution and releases it's resources

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									4 0	6 0	4 1	4 2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									4 0	6 0	4 1	4 2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									4 0	6 0	4 1	4 2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									4 0 **2** **0**	6 0 **3** **3**	4 1 **5** **3**	4 2 **4** **2**
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	**2**	**3**	**5**	**4**	4	3	5	6				
P5	**0**	**3**	**3**	**2**	0	6	5	2				

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									4 0 2 0	6 0 3 3	4 1 5 3	4 2 4 2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				

Example -1 Banker's Algo

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									4	6	4	4
									0	0	1	2
									2	3	5	4
									0	3	3	2
									2	0	0	0
									0	0	3	4

No Deadlock

Example -2

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	2	2	2	2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Example -2

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	2	2	2	2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Example -2

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
									2 0	2 0	2 1	2 2
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

Deadlock Detected

Don’t have sufficient resources to execute any other process except P1

Strategies for handling deadlocks

- **Deadlock prevention**

  Prevents deadlocks by restraining requests made to ensure that at least one of the four deadlock conditions cannot occur.

- **Deadlock avoidance**

  Dynamically grants a resource to a process if the resulting state is "**safe**".

  A state is safe if there is at least one execution sequence that allows all processes to run to completion.

Strategies for handling deadlocks

- **Deadlock detection and recovery**

  Allows deadlocks to form; then finds and breaks them.

- **Deadlock Ignorance**

  Feasible in general end user systems like out PC to avoid the execution overload.

Detection and Recover

Deadlock Detection can be done with

- Banker's Algorithm for multi instance.
- Wait for graph

Recovery from Deadlock

- 1. Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- 2. Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked

Recovery from Deadlock

- 3. Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning

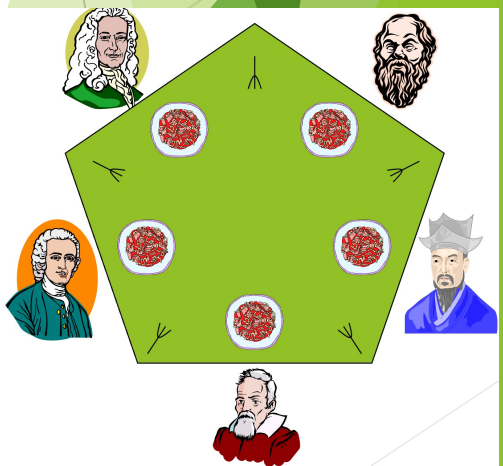
Deadlock Prevention

- We can prevent Deadlock by eliminating any of the four conditions.
  - Mutual Exclusion
  - Hold and Wait
  - No preemption
  - Circular wait

Deadlock Prevention - hold and wait

A process acquires all the needed resources simultaneously before it begins its execution,
therefore breaking the hold and wait condition.

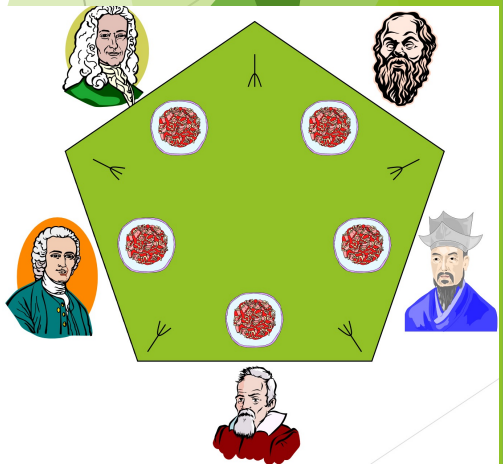
- E.g.
  - In the dining philosophers' problem, each philosopher is required to pick up both forks at the same time. If he fails, he has to release the fork(s) (if any) he has acquired.
- Drawback: over-cautious.



Deadlock Prevention - circular wait condition

All resources are assigned unique numbers.
A process may request a resource with a unique number “i” only if it is not holding a resource with a number less than or equal to “i” and therefore breaking the **circular wait condition.**

- E.g.
  - In the dining philosophers problem, each philosopher is required to pick a fork that has a larger id than the one he currently holds. That is, philosopher Pi should pick up fork Fi followed by Fi-1.
- Drawback: over-cautions.

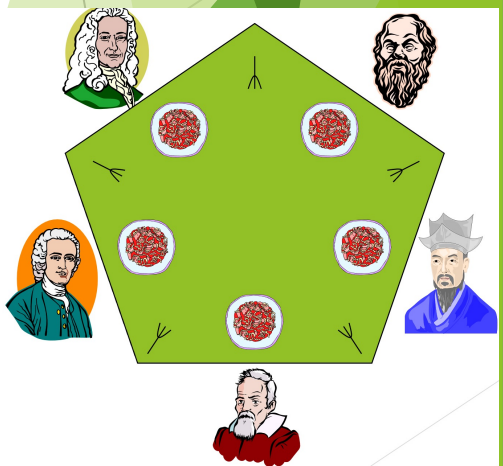


Deadlock Prevention - non-preemption condition

Each process is assigned a unique priority number.

The priority numbers decide whether process Pi should wait for process Pj and therefore break the non-preemption condition.

- E.g.
  - Assume that the philosophers' priorities are based on their ids, i.e.,
    - Pi has a higher priority than Pj if i <j.
    - In this case Pi is allowed to wait for Pi+1 for i=1,2,3,4.
    - P5 is not allowed to wait for P1. If this case happens, P5 has to abort by releasing its acquired fork(s) (if any).
- Drawback: starvation.
  - The lower priority one may always be rolled back. Solution is to raise the priority every time it is victimized.

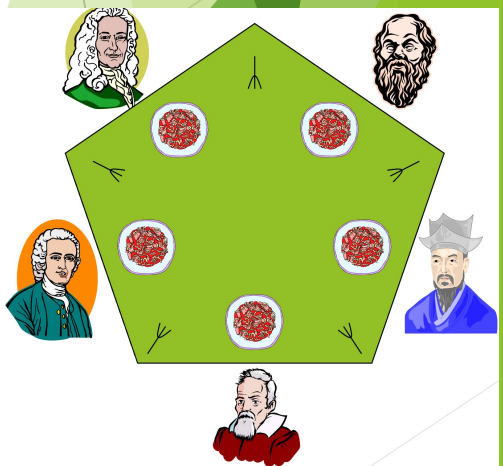


Deadlock Prevention – mutual exclusion

- Practically it is impossible to provide a method to break the mutual exclusion condition since most resources are intrinsically non-sharable,

e.g.,

- two philosophers cannot use the same fork at the same time.
- A printer resource



Deadlock Avoidance

The deadlock Avoidance method is used by the operating system in order to check whether the system is in a safe state or in an unsafe state and in order to avoid the deadlocks.

The process must need to tell the operating system about the **maximum number of resources a process can request** in order to complete its execution.

Deadlock avoidance can be done with

- Banker's Algorithm for multi instance.
- Wait for Graph

Deadlock Avoidance

Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe state

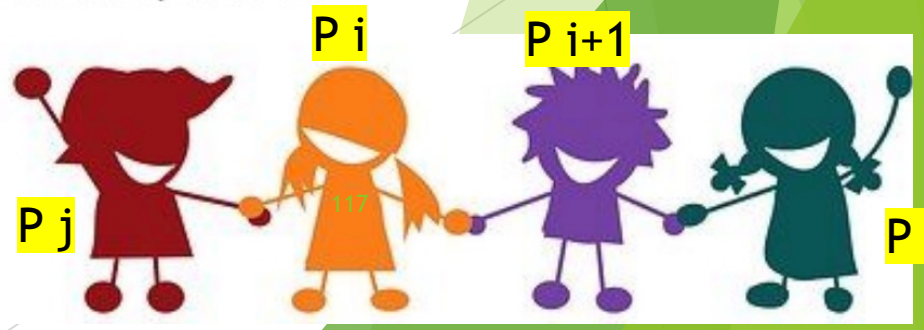
- A state is safe if the system can allocate required resources to each process in some order and still avoid a deadlock.
- Formally, a system is in a safe state only, if there exists a safe sequence.
- In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock may occurs.

Safe state

System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on

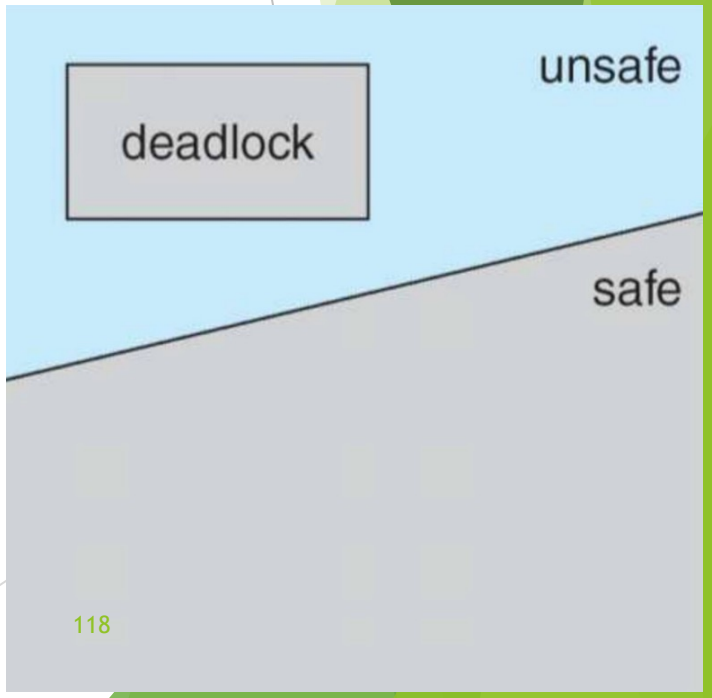
j < i.



Deadlock Avoidance

- Safe state ➔ No deadlock.
- If a system in not in safe sate ➔ Possibility of deadlock.

- Avoidance
  - Ensure that system will never enter an unsafe state, thus preventing getting into deadlock



Deadlock Ignorance

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- Including UNIX and WINDOWS, all the operating system ignore the deadlock first assuming that the user is restricted to one process.

System failure, compiler error, programming bugs, hardware crashes that occur once a week should be paid more attention rather than deadlock problem that occur once in years

(we can shut down in case of deadlock, but not in case of a flight navigation system)

END OF UNIT-III

Questions from past University papers (Unit 2 and Unit 3)

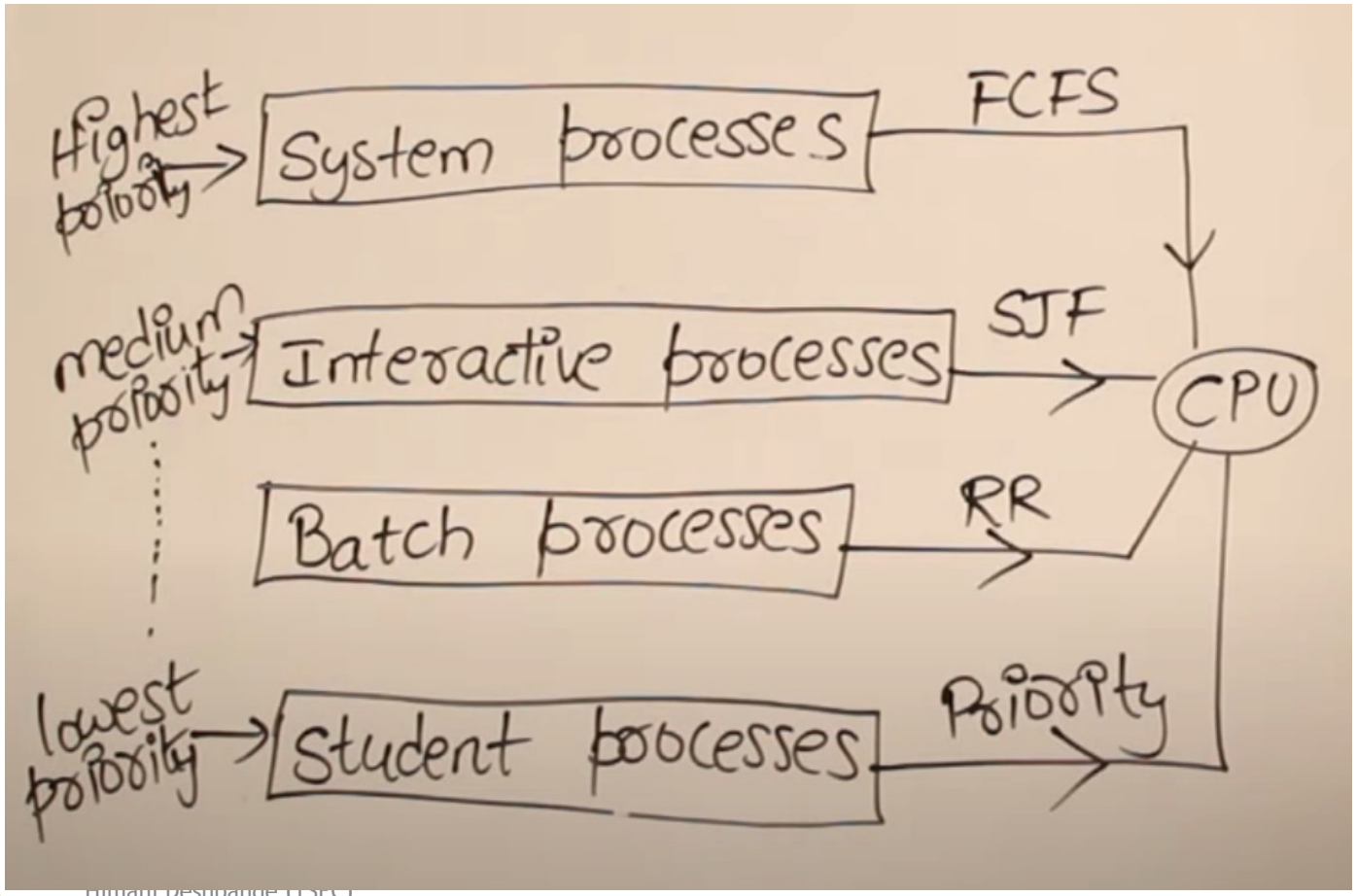
Dec 18 (IT)

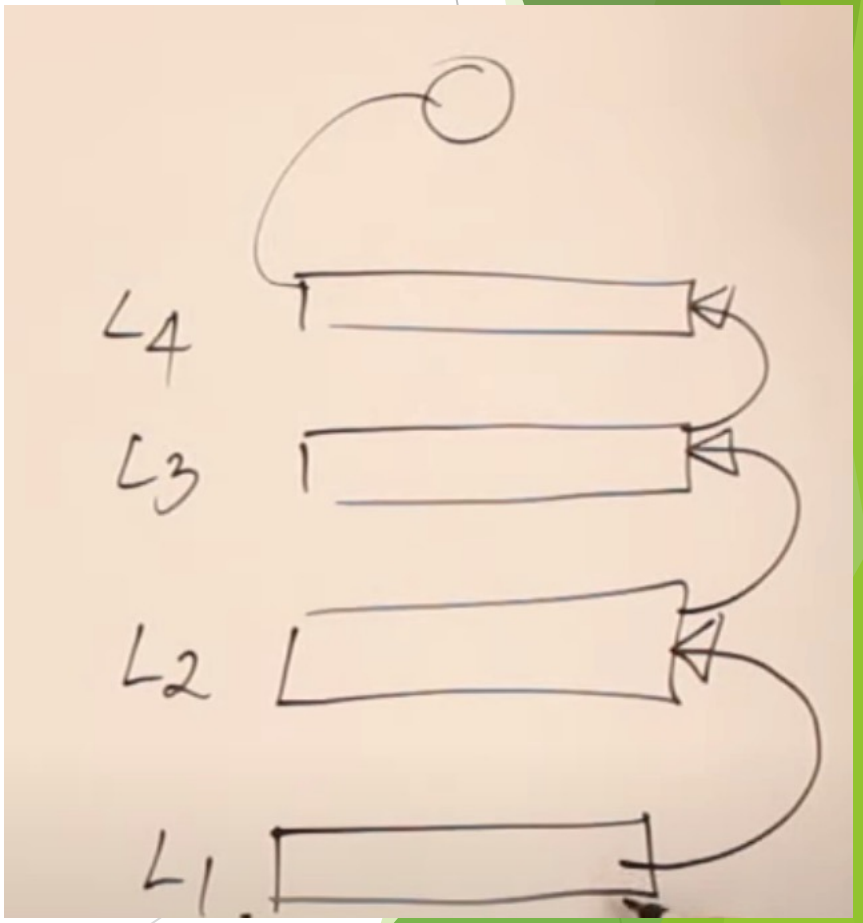
1. What is context switch? Describe the actions taken by a kernel to context switch between processes. (10)

2. Explain the differences in how the following scheduling algorithms discriminate between processes. (10)
   1. FCFS
   2. RR
   3. Multilevel feedback queue

3. What is deadlock? What are the essential conditions for deadlock to occur? (10)

Multi level queue scheduling





Dec 18 (IT) cont….

4. Describe how the Swap( ) instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement. (10)

5. Consider the following set of processes, with the length of the CPU burst given in milli seconds. The processes are assumed to have arrived in order P_1, P_2, P_3, P_4, P_5 all at time 0. **10M**

Process	*Burst Time*	*Priority*
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Calculate the average turnaround time and maximum waiting time for pre-emptive priority scheduling algorithm.

Dec 19 (IT)

1. What are the four conditions that create deadlock. Explain deadlock and avoidance techniques. (10)
2. What is scheduling ? Explain short term, mid term and long term scheduling. (10)
3. Why there is need for communication between two processes? Explain various modes of communication.
4. What are preemptive and non-preemptive algorithms? Explain any two with example.

Consider the following set of processes, with the arrival times and the CPU burst times given in milliseconds. **10**

Process	Burst Time	Arrival Time
PI	15	0
P2	5	0
P3	13	0

Draw Gantt chart, calculate Turnaround Time, Waiting Time, Average Turnaround Time and Average Waiting Time for:
i) First-Come First-Served.
ii) Shortest Job First.

May 18 (IT)

1. Explain with example which of the following algorithms could result in starvation?
    1. FCFS
    2. SJF
    3. RR
    4. Priority

2. Show that , if the wiat() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

3. Short Note
    1. Deadlock avoidance in distributed system
    2. OS schedulers

Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	ABCD	*ABCD*	*ABCD*
Po	0012	0012	1520
pl	1000	1750	
p2	1354	2356	
p3	0632	0652	
p4	0014	0656	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need?*

b. Is the system in a safe state?

c. If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

May 19 (IT)

1. Explain race condition with example. (05)

2. Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock-free if the following two conditions hold:
   a) The maximum need of each process is between 1 and m resources
   b) The sum of all maximum needs is less than m + n. (10)

3. Define critical section. What are the requirements to solve critical-section problem? (05)

4. What is the critical section problem? What requirement should a solution to critical section problem satisfy? State Peterson's solution and indicate how it satisfies the above requirements. (10)

5. Explain the Distributed Processing in Operating Systems. What are the necessary conditions for deadlock? (10)

Dec18 (comps)

1. What is Mutual exclusion? Explain its significance. (5 marks)
2. Explain various process states with diagram. (5 Marks)
3. Explain various types of scheduler. (5 Marks)
4. What is deadlock? Explain the necessary and sufficient conditions for deadlock. (10)
5. Short Note:
    1. Resource allocation graph
    2. Reader and writer problem using semaphore

b Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time. Draw the Gantt chart and find the average waiting time using the FCFS and SJF (Non-Pre-emptive) scheduling algorithm. **10**

process	Burst time
P0	21
P1	3
P2	6
P3	2

10