OPERATING SYSTEMS

Threads and Concurrency 06

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 2

UNIT 2: Threads and Concurrency

Introduction to Threads, types of threads, Multicore Programming, Multithreading Models, Thread creation, Thread Scheduling, PThreads and Windows Threads, Mutual Exclusion and Synchronization: software approaches, principles of concurrency, hardware support, Mutex Locks, Semaphores. Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts. Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads. Deadlocks: principles of deadlock, tools for detection and Prevention.

OPERATING SYSTEMS

Course Outline - Unit 2

13	4.1,4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3,5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, hardware support	6	
18	6.5,6.6	Mutex Locks, Semaphores	6	
19	6.7.1-6.7.3	Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts	6	
20	6.9	Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6	
21	Handouts	Demonstration of programming examples on process synchronization		
22	7.1-7.3	Deadlocks: principles of deadlock, Deadlock Characterization.	7	
23	7.4	Deadlock Prevention, Deadlock example	7	
24	7.6	Deadlock Detection	7	

Topics Outline

- Process Synchronisation
  - The Critical-Section Problem - Review
  - The Critical-Section Problem - Solution using Synchronisation Criteria
  - Handling of the Critical Section Problem
  - Peterson’s Solution

The Critical Section Problem

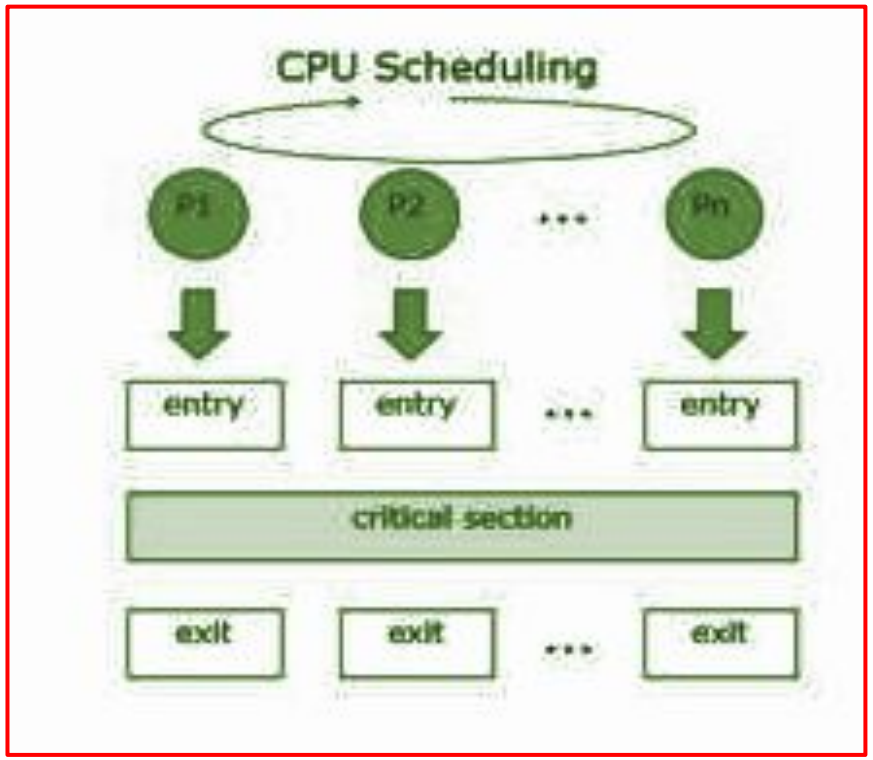
- **Critical Section** is the **part** of a program which **tries** to **access shared resources**.
- That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.
- The critical section **cannot** be executed by **more** than **one** process at the **same time**
- Operating System faces the difficulties in allowing and disallowing the processes from entering the critical section.
- The **solution** to critical section problem is used to design a set of protocols which can ensure that the race condition among the processes will **never arise**.

The Critical Section Problem => Solution -> Synchronisation Criteria

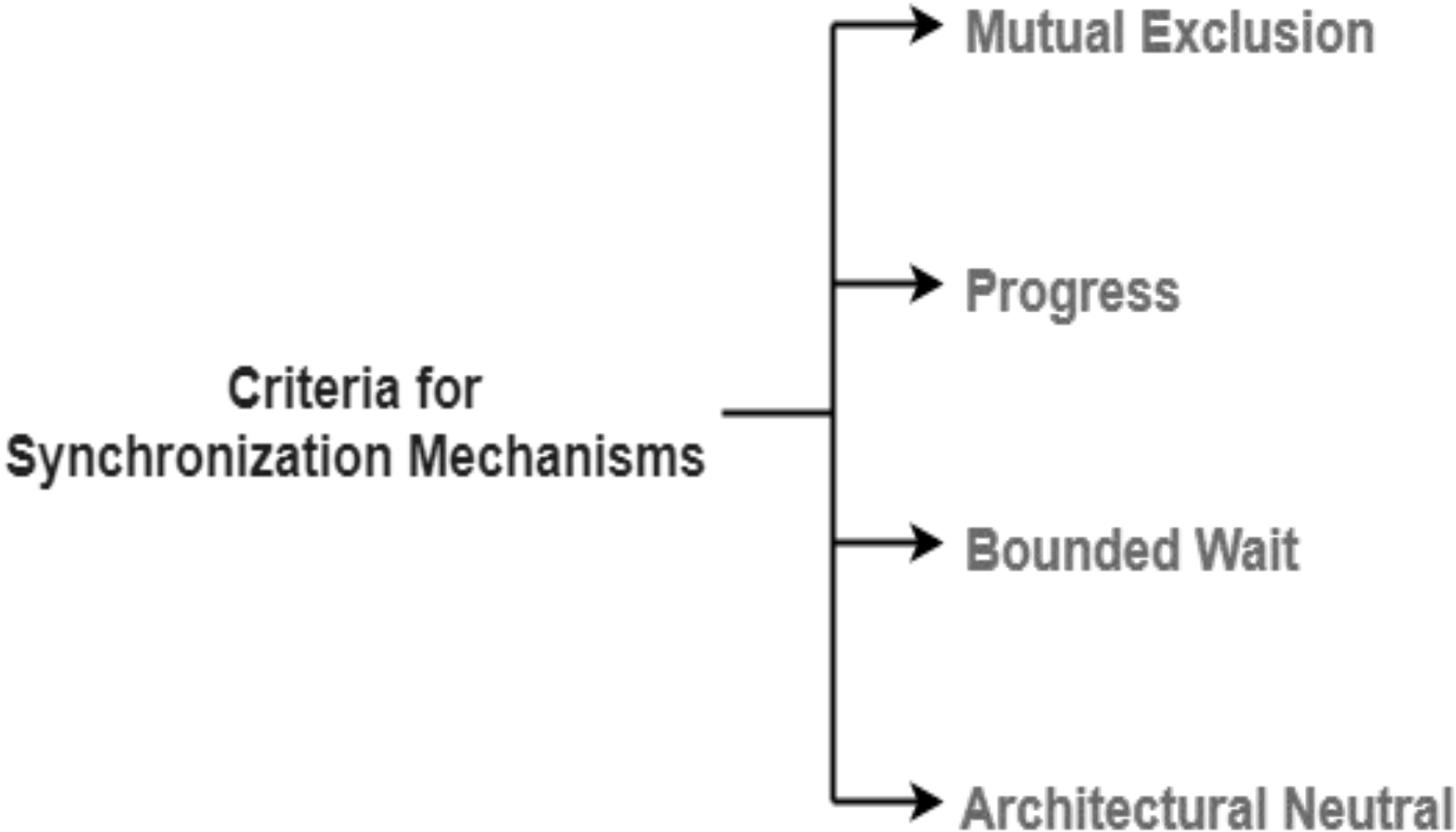
- In order to synchronize the cooperative processes, One's main task is to solve the critical section problem.

- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

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

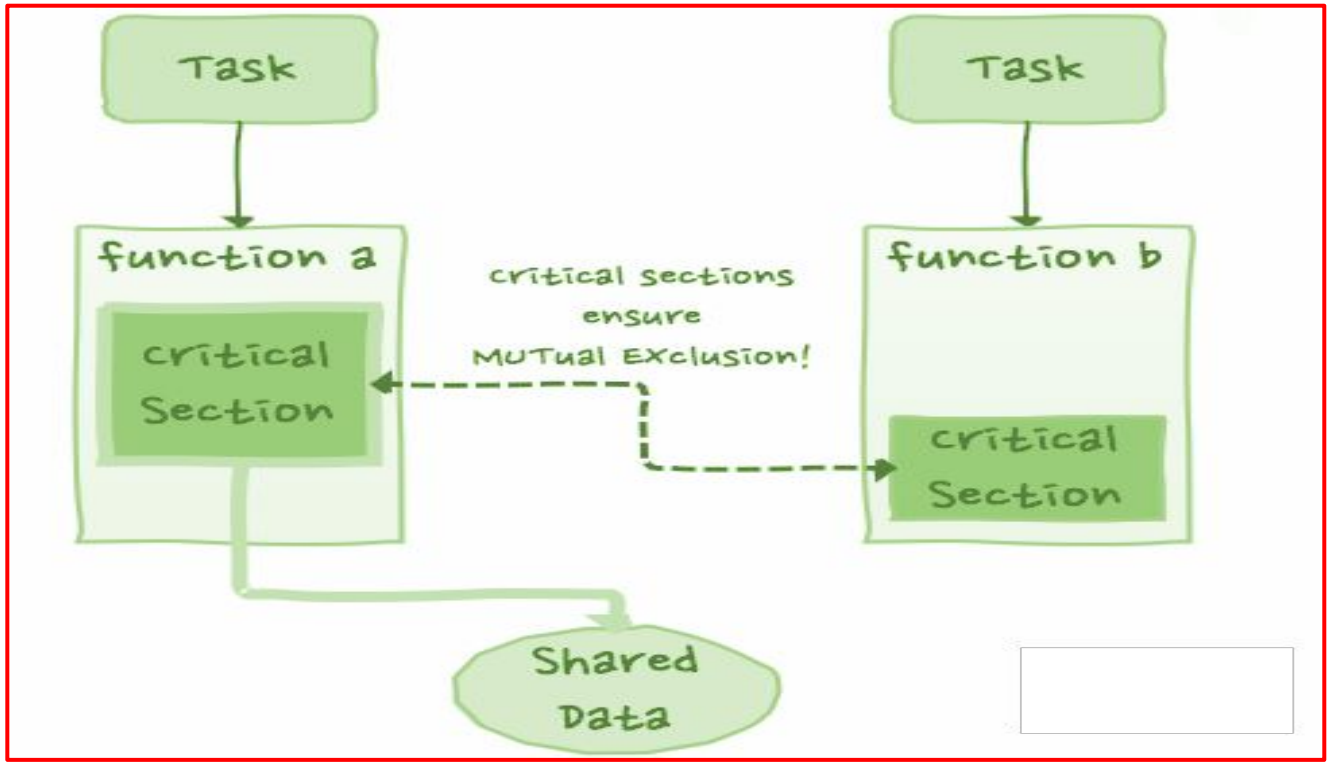


The Critical Section Problem => Solution -> Synchronisation Criteria



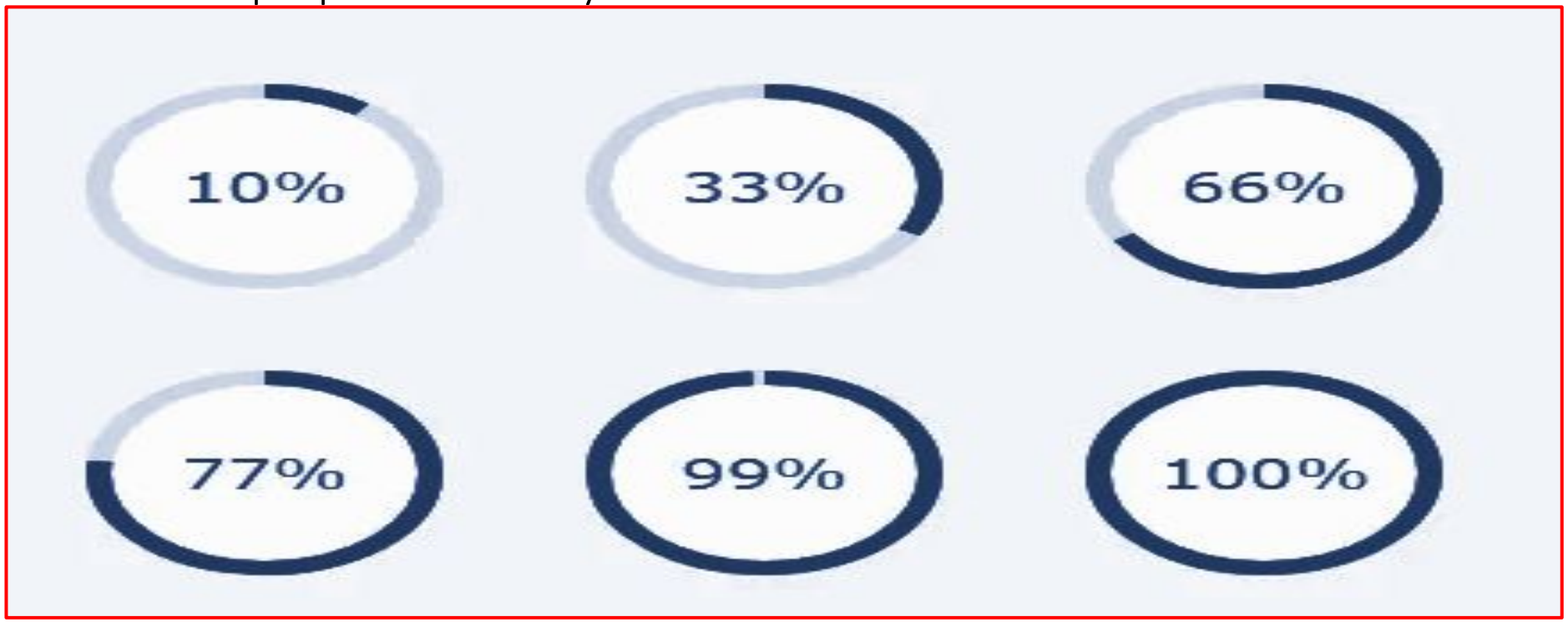
The Critical Section Problem => Solution -> Synchronisation Criteria

- In order to synchronize the cooperative processes, main task is to solve the critical section problem.
- One needs to provide a solution in such a way that the **following conditions** can be **satisfied**.
- **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections



The Critical Section Problem => Solution -> Synchronisation Criteria

- In order to synchronize the cooperative processes, main task is to solve the critical section problem.
- One needs to provide a solution in such a way that the **following conditions** can be **satisfied**.
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely



The Critical Section Problem => Solution -> Synchronisation Criteria

- In order to synchronize the cooperative processes, main task is to solve the critical section problem.
- One needs to provide a solution in such a way that the **following conditions** can be **satisfied**.
- **Bounded Waiting -** A bound must exist 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
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the n processes

The Critical Section Problem => Solution -> Synchronisation Criteria

- In order to synchronize the cooperative processes, main task is to solve the critical section problem.
- One needs to provide a solution in such a way that the **following conditions** can be **satisfied**.
- **Architecture Neutral:** The mechanism should ensure
  - It can run on any architecture without any problem.
  - There is no dependency on the architecture.

The Critical Section Problem => Solution -> Synchronisation Criteria

- In order to synchronize the cooperative processes, main task is to solve the critical section problem.
- One needs to provide a solution in such a way that the **following conditions** can be **satisfied**.
- **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting -** A bound must exist 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
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the n processes
- **Architecture Neutral:** The mechanism should ensure
    - It can run on any architecture without any problem.
    - There is no dependency on the architecture.

The Critical Section Problem => Solution -> Synchronisation Criteria

- **Mutual Exclusion and Progress** are the **mandatory** criteria. They must be fulfilled by all the synchronization mechanisms.

- **Bounded waiting and Architectural neutrality** are the **optional** criteria. However, it is recommended to meet these criteria as far as possible.

Handling of Critical Section in OS

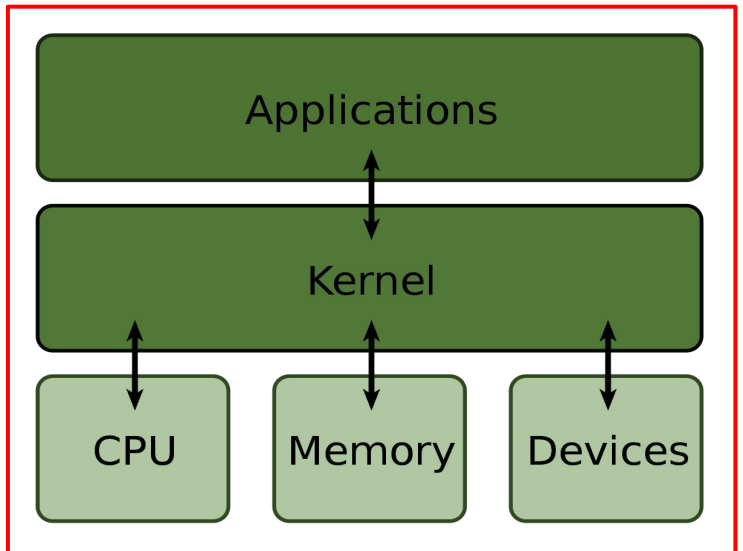
- Two approaches for handling CS in OS depends on if kernel is **preemptive** or **non- preemptive**

- **Preemptive Kernel** => allows preemption of process when running in kernel mode

- **Non-preemptive Kernel** => runs until exits kernel mode, blocks, or voluntarily yields CPU. Essentially free of race conditions in kernel mode

Software Solution to Critical Section Problem - Peterson's Solution

- It is the solution for Two processes
- Good algorithmic description of solving the two process critical section problem
- Because the load and store machine-language instructions are at a **lower layer** of **abstraction** they are always atomic w.r.t OS ; that is, **cannot** be interrupted, **independent** of the **OS** in **Preemptive** Kernel Mode or **Non-Preemptive** Kernel Mode



Software Solution to Critical Section Problem - Peterson's 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 Pi is ready!

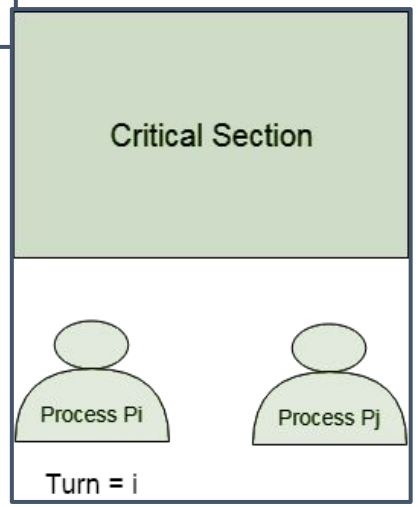
Software Solution to Critical Section Problem - Peterson's Solution

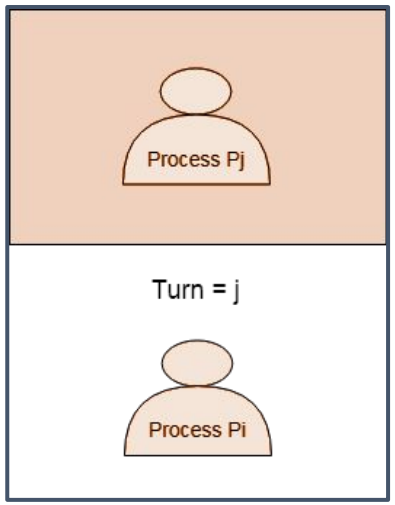
```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

Pi

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

Pj

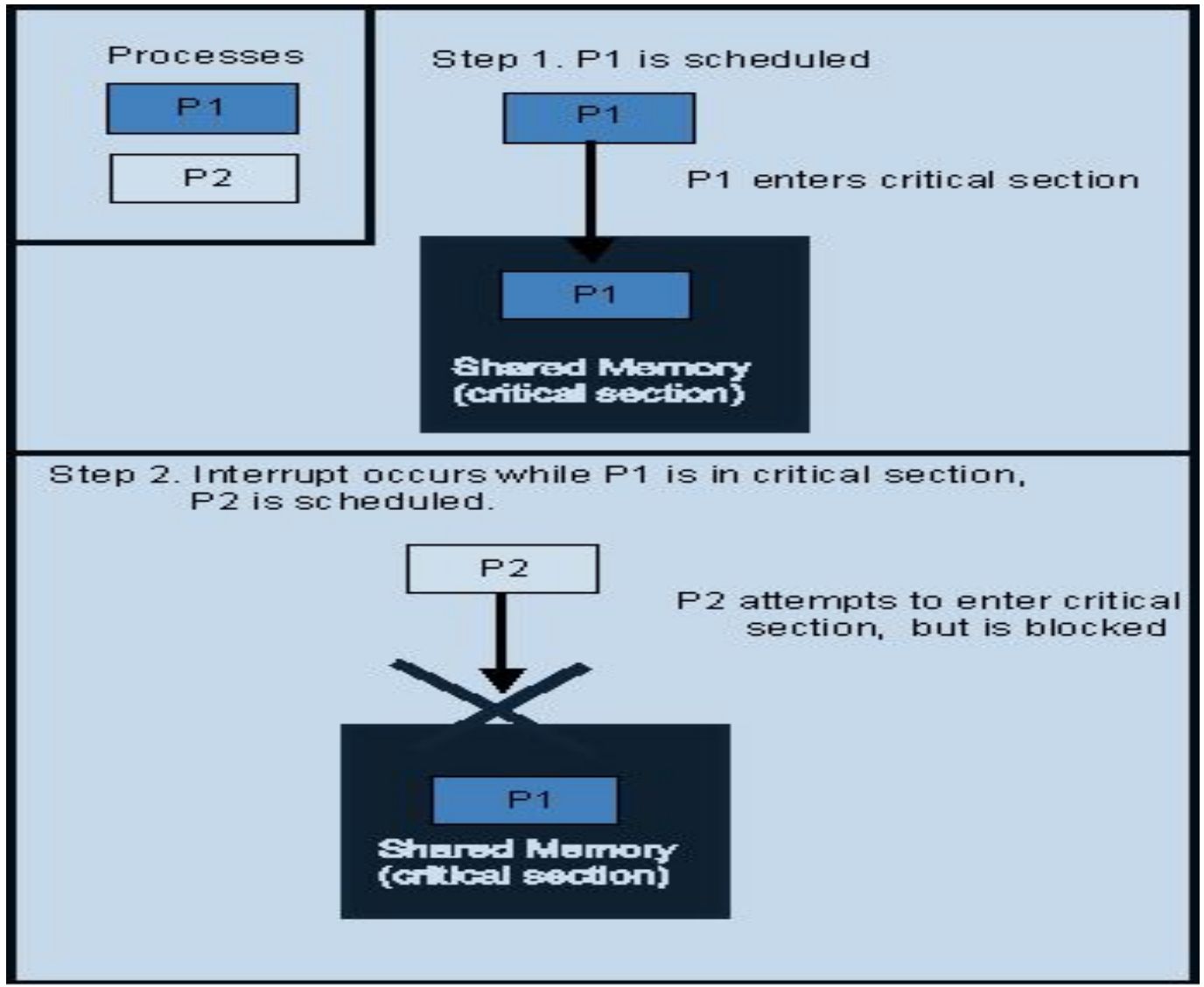




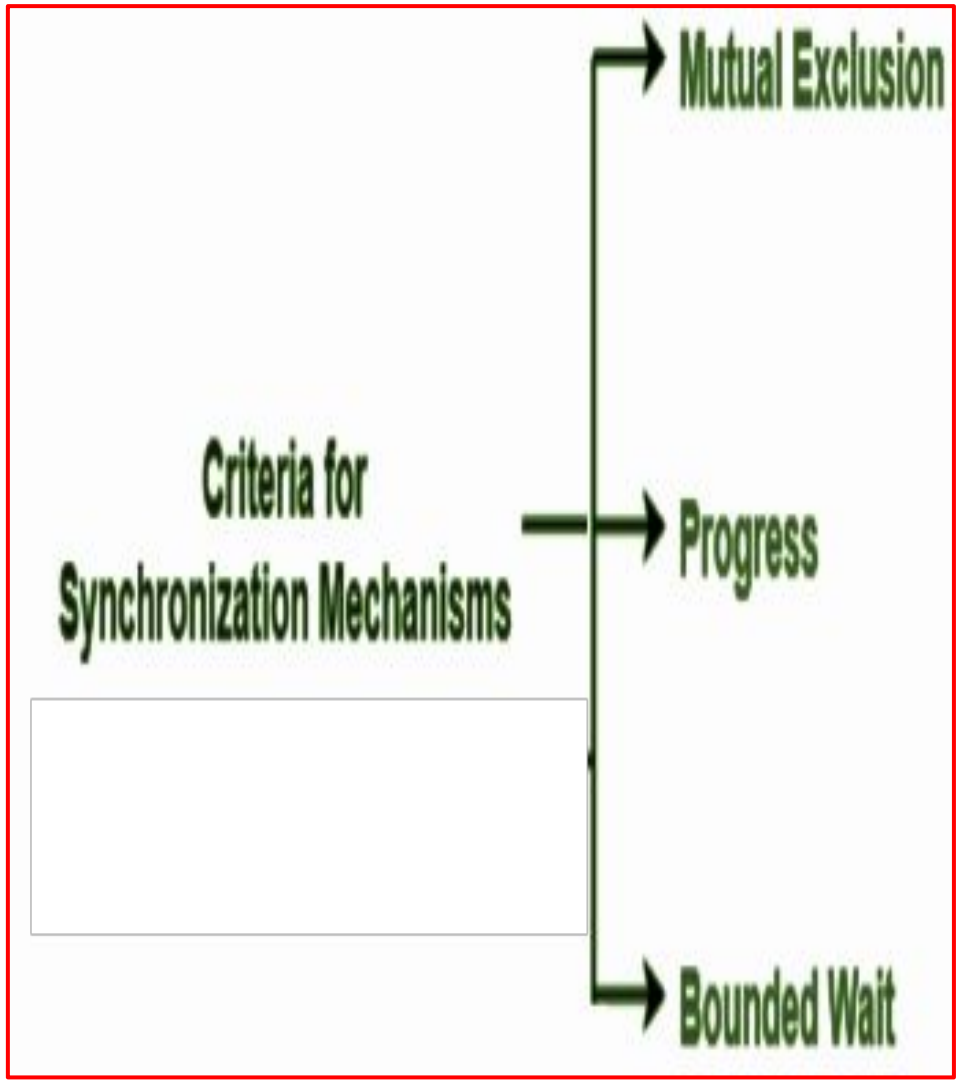
Software Solution to Critical Section Problem - Peterson's Solution

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

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



Software Solution to Critical Section Problem - Peterson's Solution



- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Topics Uncovered in this Session

- Process Synchronisation
  - The Critical-Section Problem - Review
  - The Critical-Section Problem - Solution using Synchronisation Criteria
  - Handling of the Critical Section Problem
  - Peterson’s Solution

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com