OPERATING SYSTEMS

Mutual Exclusion & Synchronization:Software

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

Course Syllabus - Unit 2

12 Hours

Unit 2: Threads & 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.

Course Outline

13	Introduction to Threads, types of threads, Multicore Programming, Multithreading Models	4.1 - 4.3	42.8
14	Thread creation, Thread Scheduling	5.4	
15	Pthreads and Windows Threads	4.4	
16	Mutual Exclusion and Synchronization: software approaches,	6.1-6.2	
17	principles of concurrency, hardware support	6.3-6.4	
18	Mutex Locks, Semaphores	6.5, 6.6	
19	Classic problems of Synchronization: Bounded-Buffer Problem, Readers-Writers problem	6.7-6.8	
20	Dining-Philosophers Problem	6.8	
21	Synchronization Examples: Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6.9	
22	Deadlocks: principles of deadlock, Deadlock Characterization	7.1-7.3	
23	Deadlock Prevention, Deadlock example	7.4-7.5	
24	Deadlock Detection, Algorithm	7.6	

Topic Outline

- The Critical-Section Problem
- Software Solution
- Peterson’s Solution

Background

- Processes can execute concurrently
- May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

The Producer - Consumer Problem

- One of the most common task structures in concurrent systems is illustrated by the producer-consumer problem.

- In this problem, threads or processes are divided into two relative types:

  - A producer thread / process is responsible for performing an initial task that ends with creating some result

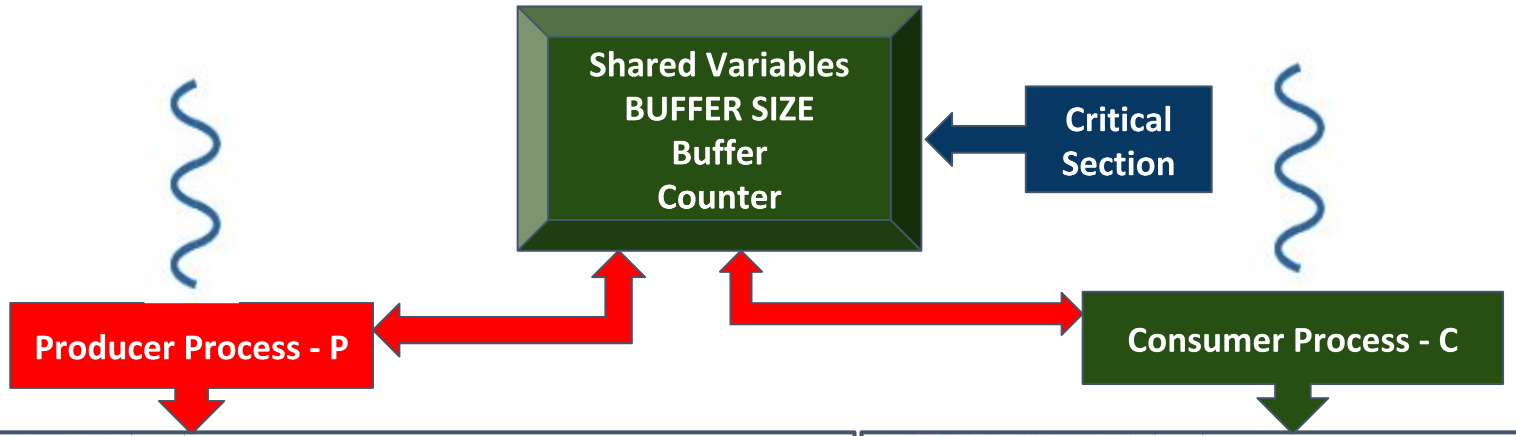
  - A consumer thread that takes that initial result for some later task.

- Between the threads or processes, there is a shared array or queue that stores the results being passed.

- One key feature of this problem is that the consumer removes the data from the queue and "consumes" it by using it in some later purpose.

- There is no way for the consumer threads or processes to repeatedly access data in the queue.

The Producer - Consumer Problem




```
while (true)
{
  /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
              /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true)
{
    while (counter == 0)
           ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
     counter--;
    /* consume the item in next consumed */
}
```

The Producer - Consumer Problem

- Both Producer and Consumer can run Concurrently either with True Concurrency or Pseudo Concurrency or both as deemed fit
- The P and C order of execution is not guaranteed

Schedule 1	P	C	P	C	P	P
Schedule 2	P	P	P	C	C	C
Schedule 3	C	C	C	C	P	P
Schedule 4	P	P	C	C	P	C
Schedule 5	P	C	P	C	P	C

The Producer - Consumer Problem: The Race Condition

- **counter++**

could be implemented as
S0=>register1 = counter
S1=>register1 = register1 + 1
S2=>counter = register1

- counter--

could be implemented as
S0=>register2 = counter
S1=>register2 = register2 - 1
S2=>counter = register2

Consider this execution interleaving with "count = 5" initially:

P=>S0: Producer execute register1 = counter {register1 = 5}

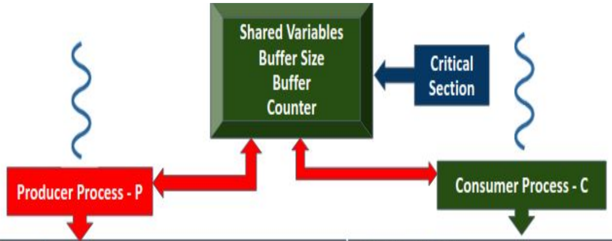
P=>S1: Producer execute register1 = register1 + 1 {register1 = 6}

C=>S0: Consumer execute register2 = counter {register2 = 5}

C=>S1: Consumer execute register2 = register2 – 1 {register2 = 4}

P=>S2: Producer execute counter = register1 {counter = 6 }

C=>S2: Consumer execute counter = register2 {counter = 4}




```
while (true)
{
  /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
              /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true)
{
   while (counter == 0)
         ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
    counter--;
   /* consume the item in next consumed*/
}
```

The Producer - Consumer Problem: Non Occurrence of Race Condition by Luck

- **counter++**

could be implemented as
S0=>register1 = counter
S1=>register1 = register1 + 1
S2=>counter = register1

P

- counter--

could be implemented as
S0=>register2 = counter
S1=>register2 = register2 - 1
S2=>counter = register2

C

Consider this execution interleaving with "count = 5" initially:

P=>S0: Producer execute register1 = counter {register1 = 5}

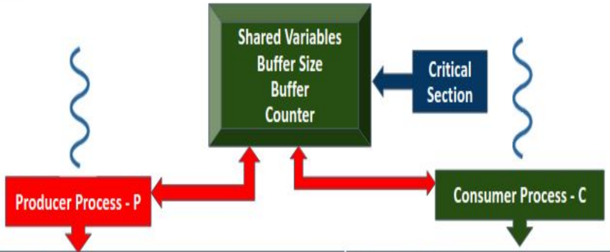
P=>S1: Producer execute register1 = register1 + 1 {register1 = 6}

P=>S2: Producer execute Counter = register1 {Counter = 6}

C=>S0: Consumer execute register2 = Counter {register2 = 6}

C=>S1: Consumer execute register2 = register2- 1 {register2 = 5}

C=>S2: Consumer execute counter = register2 {counter = 5}




```
while (true)
{
  /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
            /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true)
{
   while (counter == 0)
        ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
   /* consume the item in next consumed*/
}
```

The Critical Section Problem

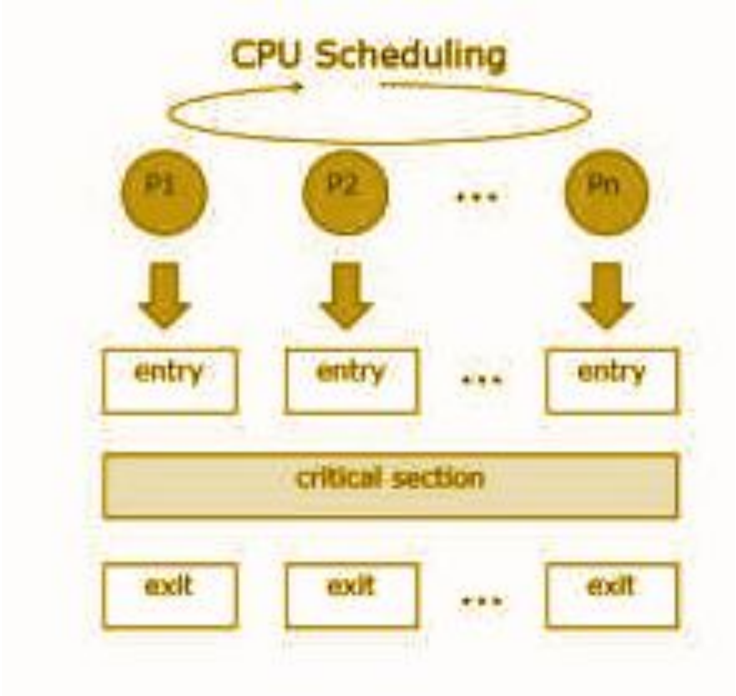
- Critical Section is the part of a program which tries to access shared resources.
- That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.
- The critical section cannot be executed by more than one process at the same time
- Operating System faces the difficulties in allowing and disallowing the processes from entering the critical section.
- The 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 =>Synchronization Criteria

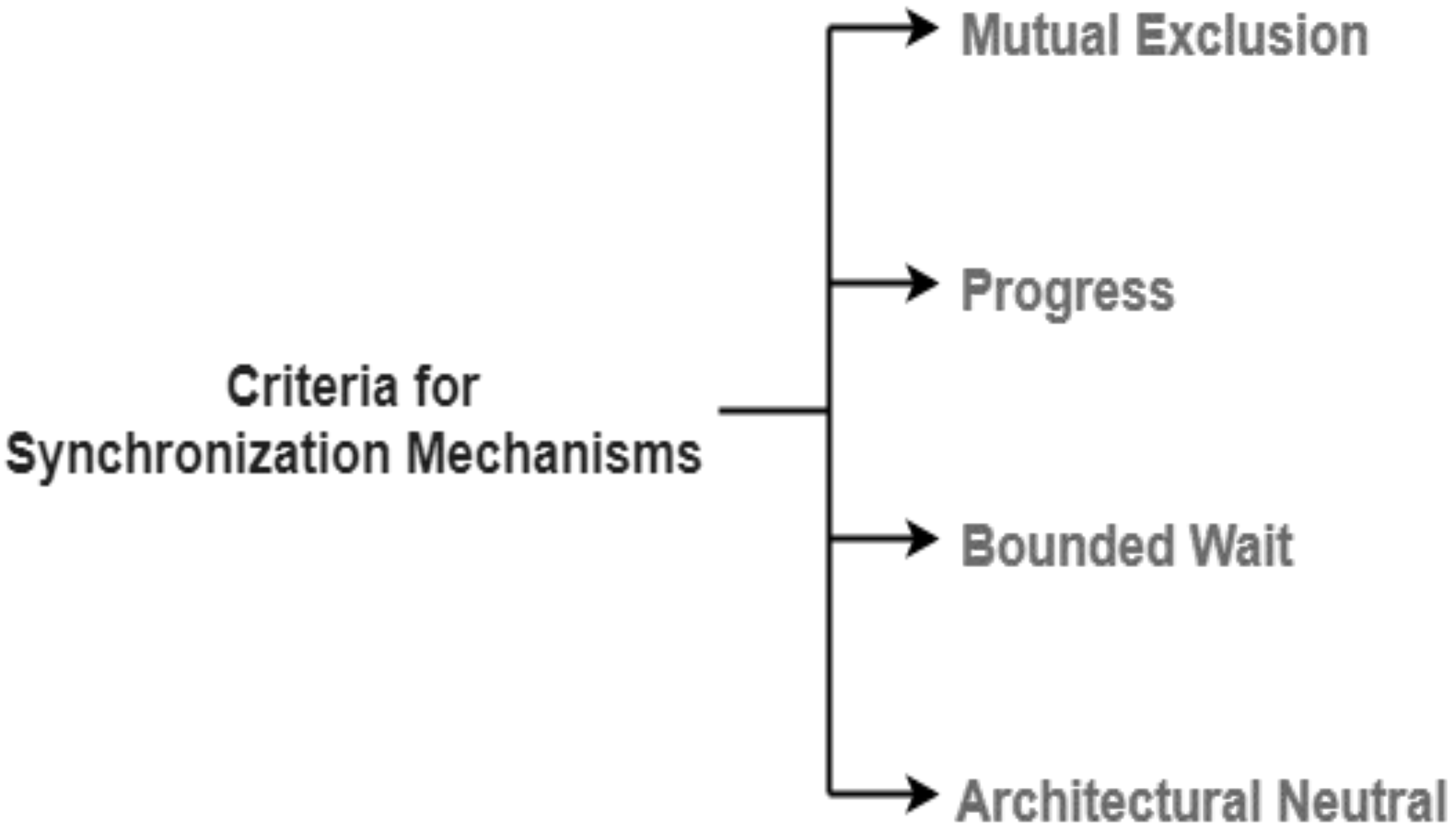
- In order to synchronize the cooperative processes, One's main task is to solve the critical section problem.

- 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);
```



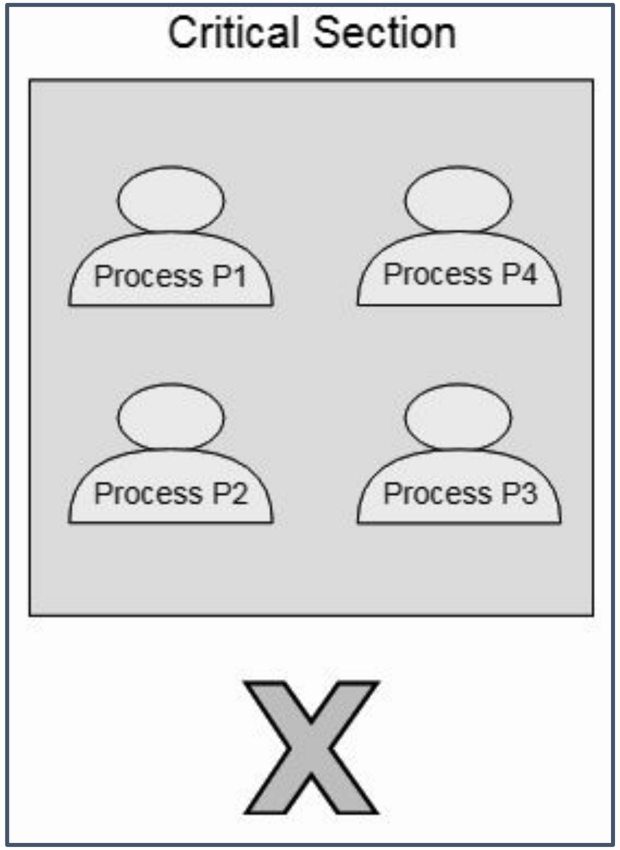
The Critical Section Problem => Solution =>Synchronization Criteria

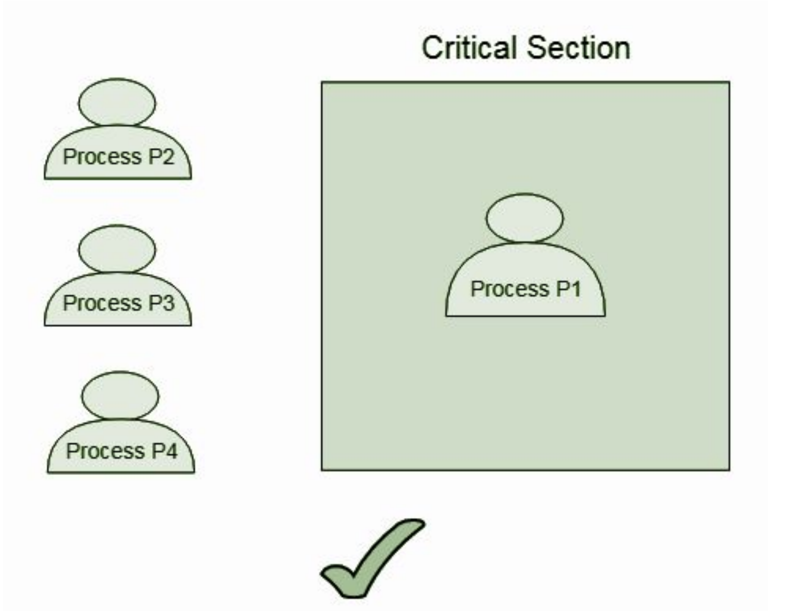


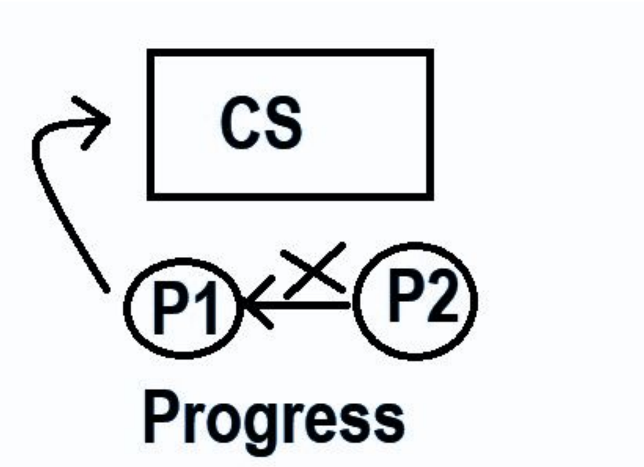
The Critical Section Problem => Solution =>Synchronization Criteria

- In order to synchronize the cooperative processes, One’s 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 =>Synchronization Criteria







The Critical Section Problem => Solution =>Synchronization 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

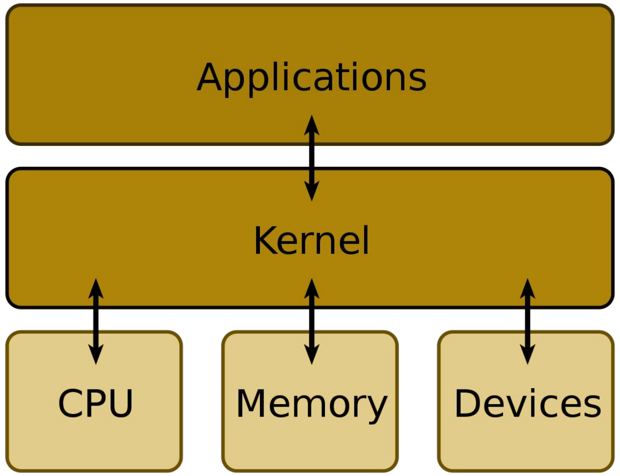
- Two approaches for handling CS in OS depends on if kernel is preemptive or non- preemptive
- **Preemptive Kernel** => allows preemption of process when running in kernel mode
- **Non-preemptive Kernel** => runs until exits kernel mode, blocks, or voluntarily yields CPU. Essentially free of race conditions 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]

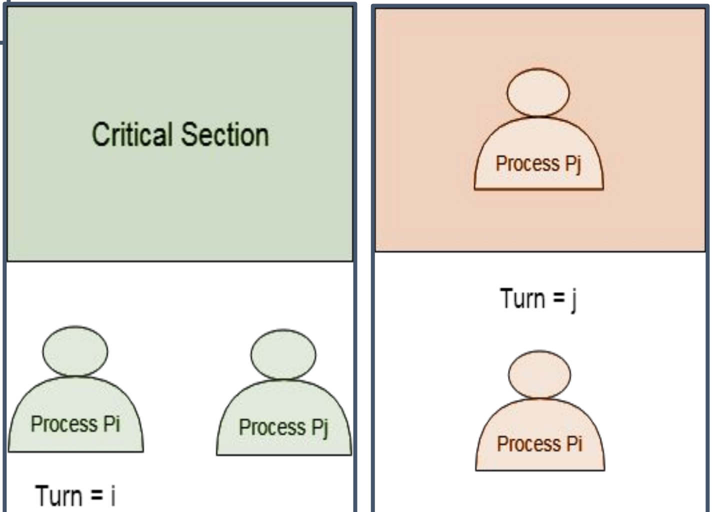
- The variable turn indicates whose turn it is to enter the critical section

- 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);
```

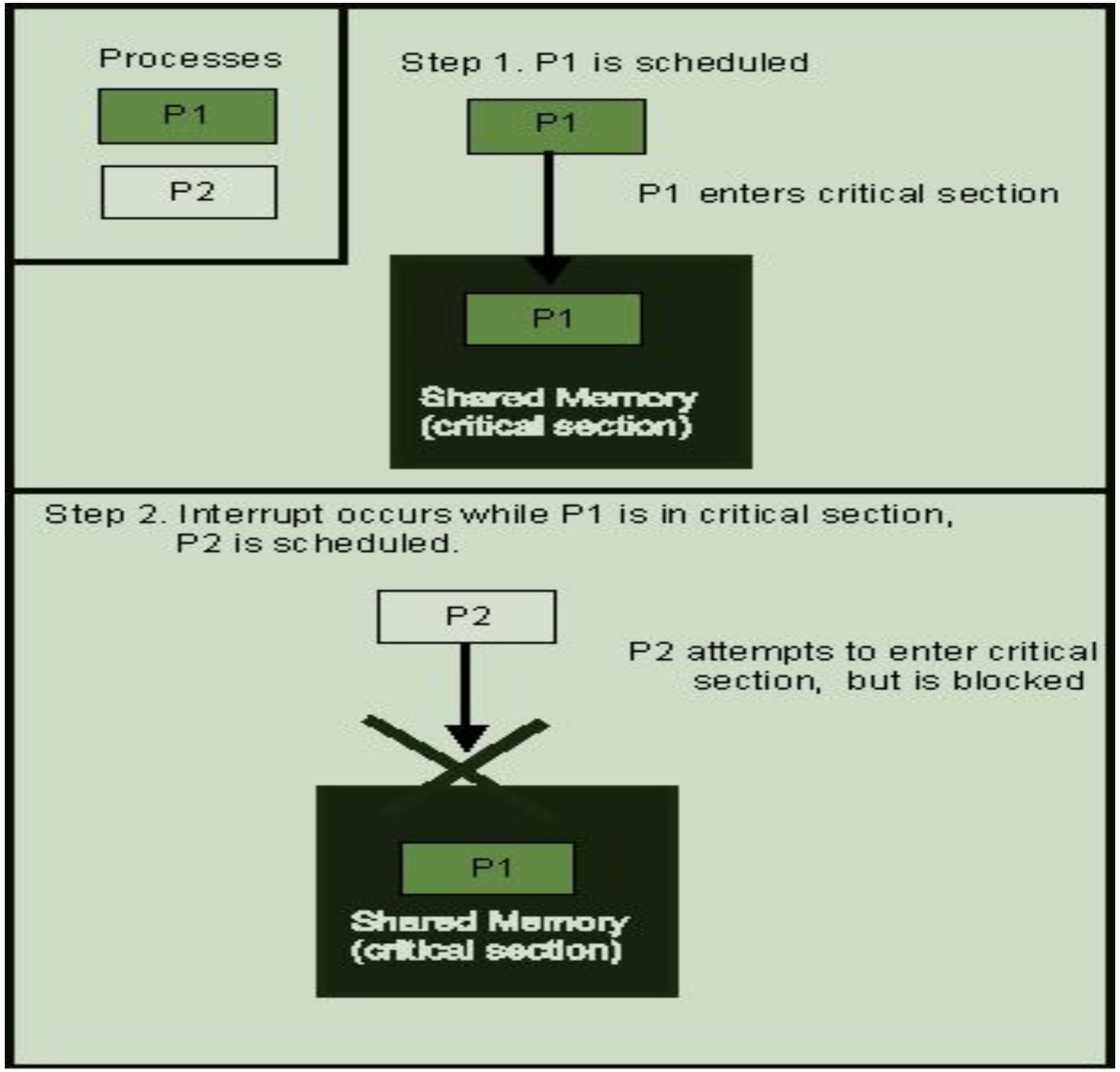
Pj

OPERATING SYSTEMS

Software Solution to Critical Section Problem - Peterson’s Solution

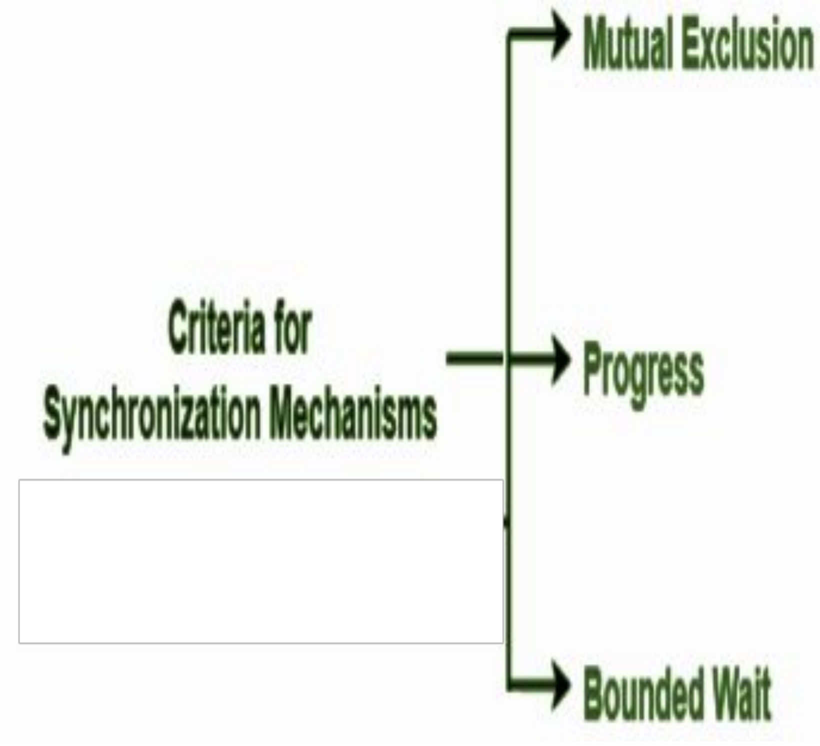
```
do {
      flag[i] = true;
      turn = j;
      while (flag[j] && turn = = j);
            critical section
      flag[i] = false;
            remainder section
 } while (true);
```

```
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

Proof of Peterson's Solution meeting the criteria of Solution to Critical Section Problem



- 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

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