CS & IT ENGINEERING

Operating System

REVISION

Process Synchronization (Part - 01)

Lecture No.- 05

By- Dr. Khaleel Khan Sir

Recap of Previous Lecture

Topic CPU Scheduling

Topics to be Covered

- **Topic** Need for Synchronization
- **Topic** Critical Section Problem
- **Topic** Peterson Solution
- **Topic** Hardware Synchronization

What is Synchronization? Why is it needed? <Need for Synchroniz.>

How to achieve a Synchronized IPC Environment

I.P.C

<Interprocess Communication>

Synchronization =

agreed protocol b/w the Processes involved in Communication, So as to ensure that there are no observable problems

Need for Synchronization ~ Problems that arise due to Lack of Synch. in IPC Environ.

(i) No Inconsistency (Incorrectness / wrong results)

(ii) Data loss

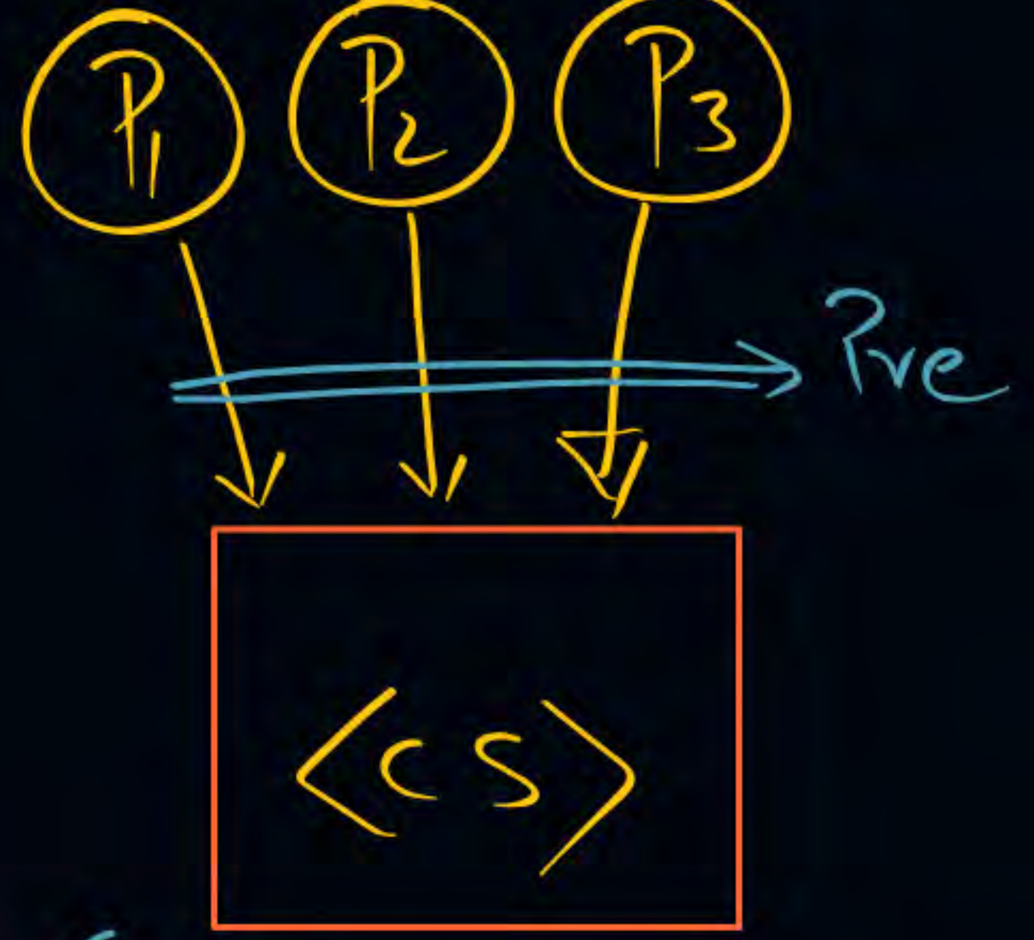
(iii) Deadlocks | Livelocks

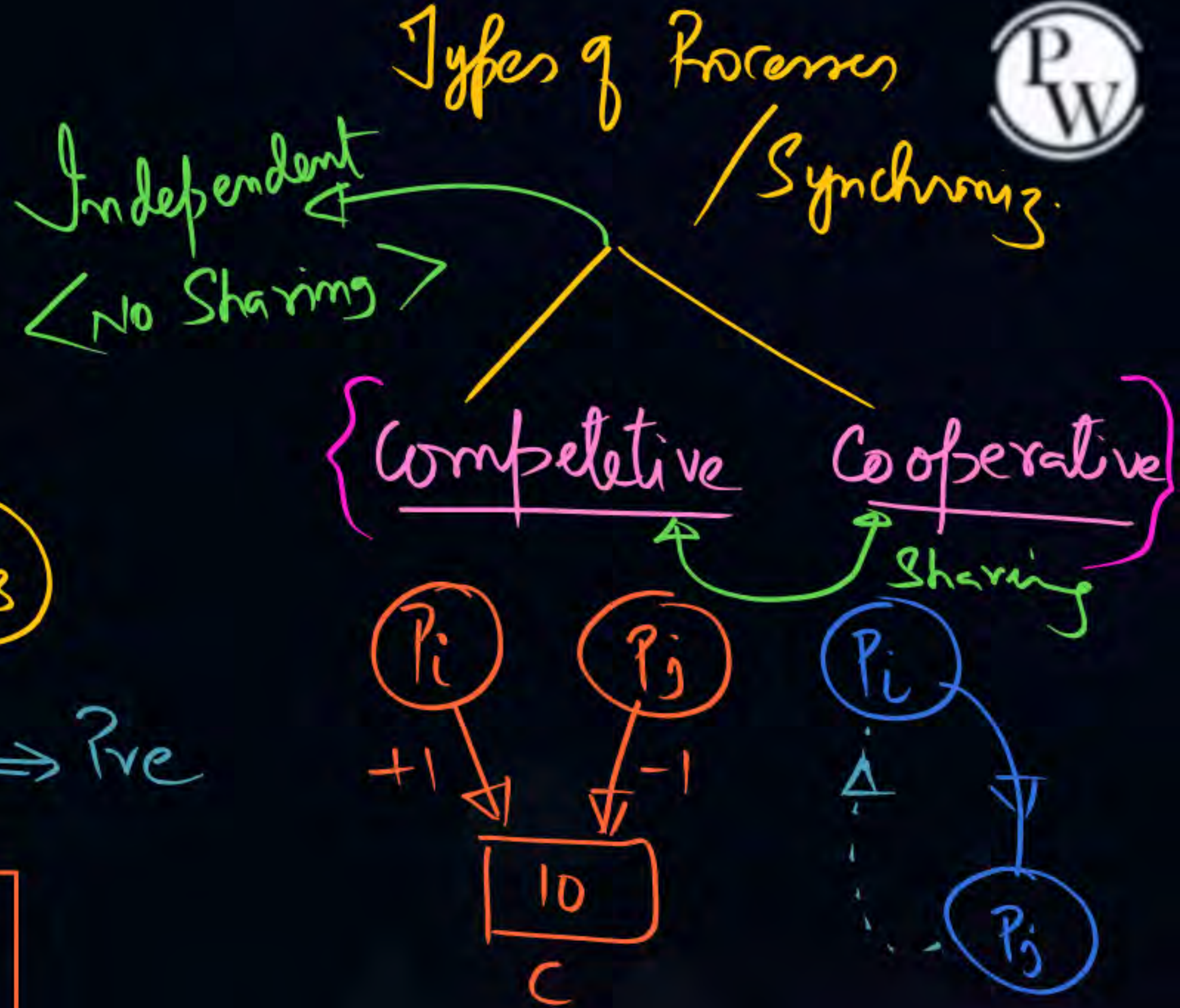
Necessary Conditions:

(i) Presence of Shared resources (Critical Section)

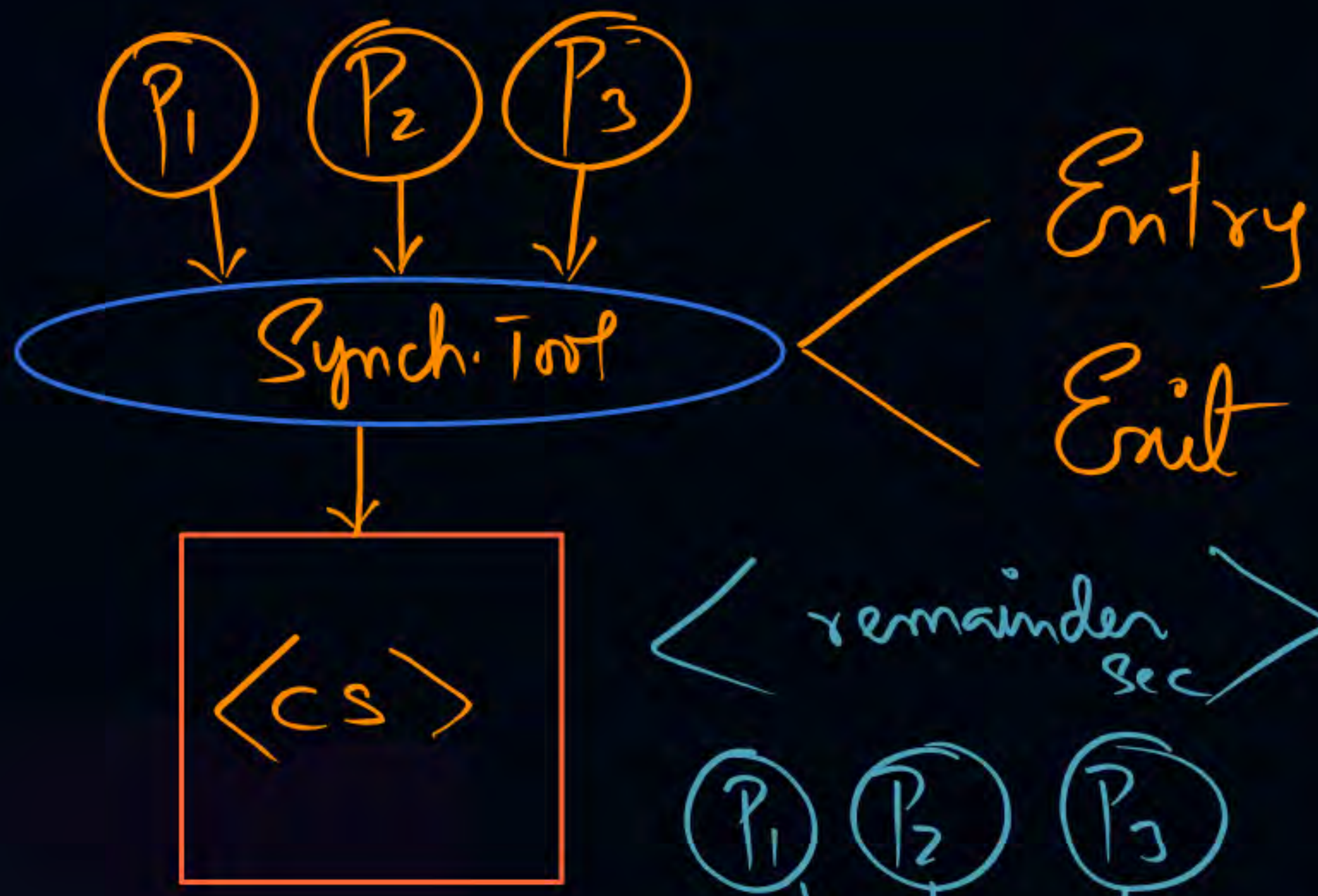
(ii) Race Conditions

(iii) PreEmption

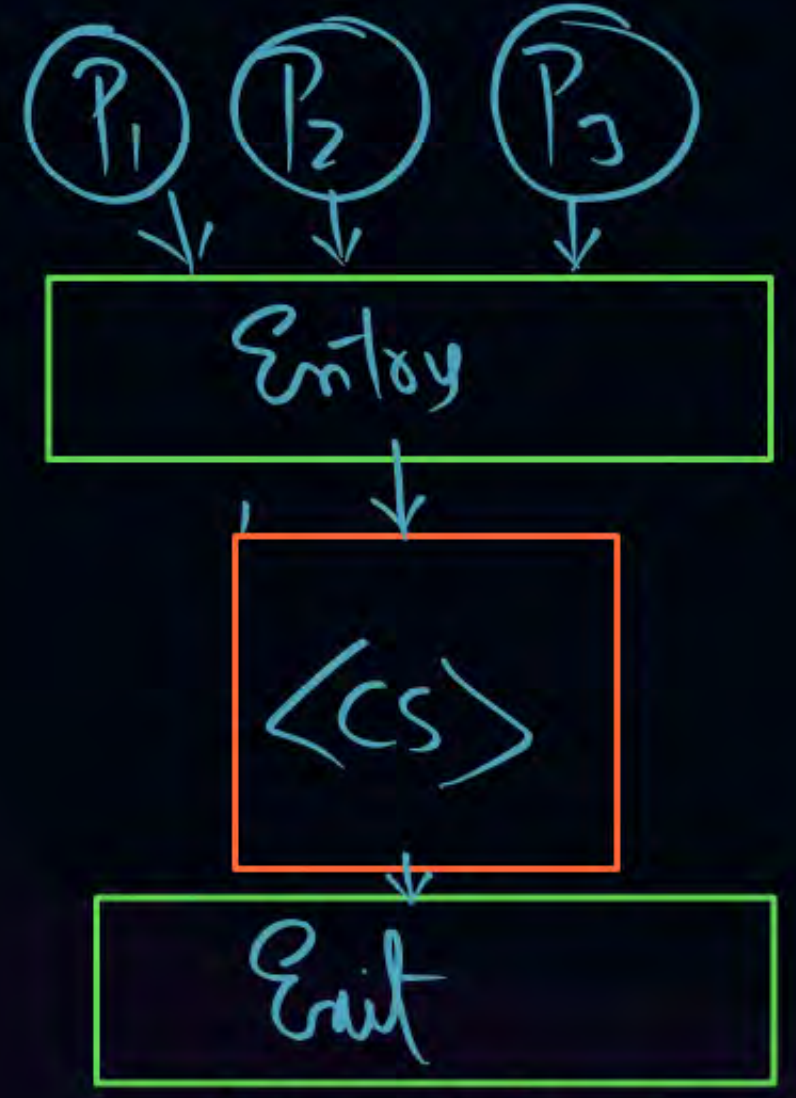




(Critical Section Problem) (CSP) $\Longrightarrow$ Synchr. Mech (Soln) / Tool (Algo)



< remainder sec >



<remainder Sec>

Process$_i$

{

Non-CS
<remainder Sec>

<CS>

}

Topic : Objectives

- Describe the Critical-Section problem and illustrate a Race condition.
- Illustrate Hardware solutions to the Critical-Section problem using Memory Barriers, Compare-and-Swap operations, and Atomic Variables.
- Demonstrate how Mutex Locks, Semaphores, Monitors, and Condition variables can be used to solve the Critical Section problem.

Topic : Background

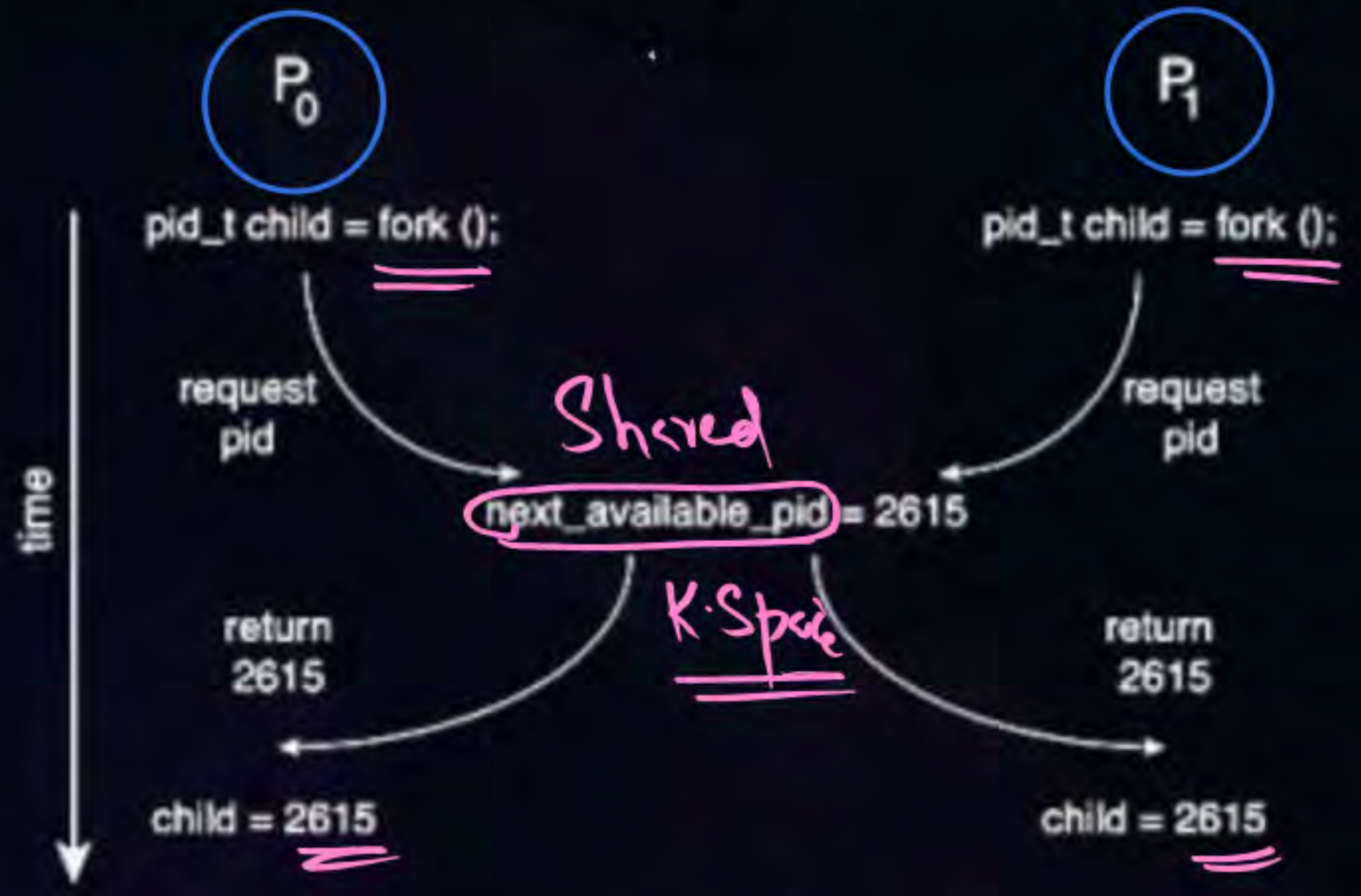
- Processes can execute concurrently

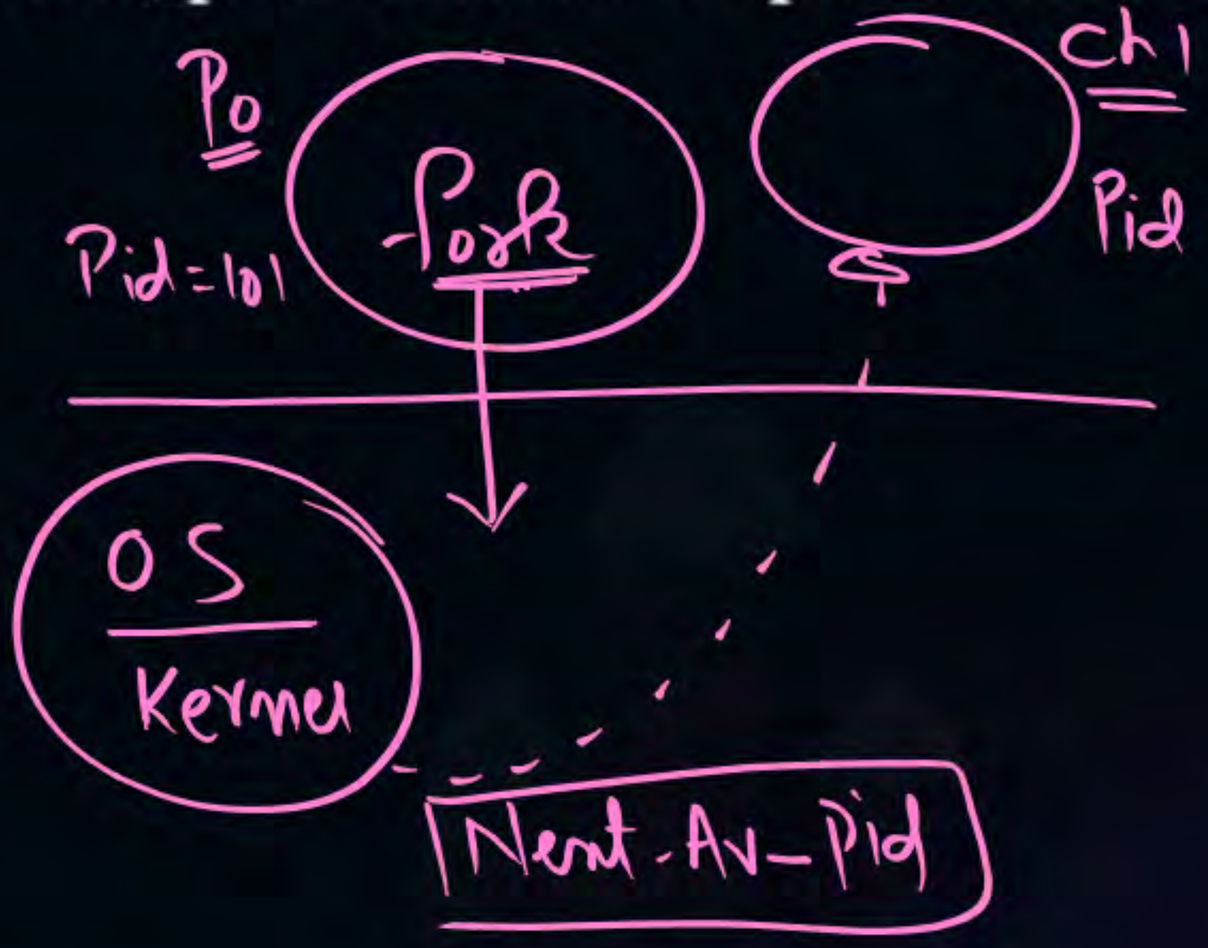
  (Pre Emption)

  - 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
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.

Topic : Race Condition

- Processes P0 and P1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid)





- Unless there is a mechanism to prevent P0 and P1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!

Topic : Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

Topic : Critical Section

- General structure of process Pi

```
while (true) {
    entry section
    < critical section >
    exit section
    < remainder section >
}
```

Requirements of C.S Problem

(i) Mutual Exclusion } Mandatory

(ii) Progress

(iii) Bounded wait } Starvation

Topic : Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. 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

Synchronization Tools

- loops → **Busy-waiting**
- if-then-else → **Non-Busy-waiting / Blocking**

S/W

→ Lock Var
→ Str. Alt
→ Peterson
→ Dekker's
→ Bakery Algo

Impl'td @ U.M
(No H/w + No OS Support)

H/W

→ Interrupt
→ TSL
→ Swap (CAS)

(Spl. Arch based | Supported Instr's)

OS - Based

Sleep - wakeup
Semaphores
< Monitor > PL Support

Part | Supported | Impl'td
Within OS Kernel

Topic : Interrupt-based Solution

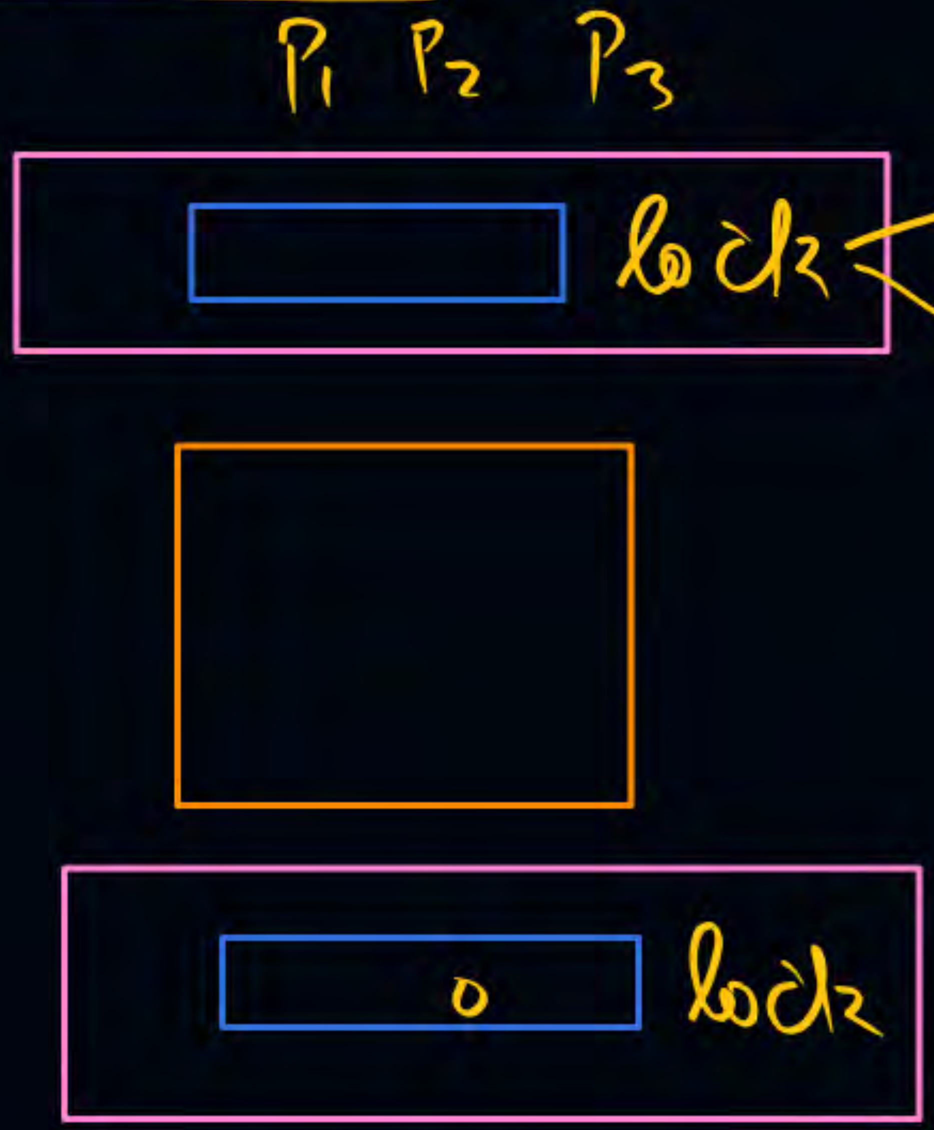
Problem
↓
PreEmption
↓
Interrupt

- Entry section: Disable Interrupts : Non-Impl. @ UM
- Exit section: Enable Interrupts
- Will this solve the problem?
  - What if the critical section code that runs for an hour?
  - Can some processes starve – never enter their critical section. P_i P_j P_k
  - What if there are two CPUs?

Entry $\boxed{\text{Disable-Interrupt}}$
↓
$\boxed{\text{<cs>}}$

Exit $\boxed{\text{Enable Interrupt}}$

(i) Lock variable :



lock:
- 0 → CS is free
- 1 → CS is in use

→ M.E : ✗

→ Progress : ✓

→ Bounded wait : ✗

Topic : Software Solution 1

Strict Alternation :

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - int turn;
- The variable turn indicates whose turn it is to enter the critical section
- initially, the value of turn is set to i

Topic : Algorithm for Process Pi

```
while (true){
    while (turn = = j);
        /* critical section */
    turn = j;
    /* remainder section */
}
```

→ Mut-Exclusion : ✓ (always)

→ Progress : ✗

P0 P1

0 turn

P1 : - - - <cs>

P0 : I am not Intrstd in c.s
<remainder Sec>

<cs>

Topic : Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  turn = i

  and turn cannot be both 0 and 1 at the same time

- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Topic : Peterson's Solution

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- 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 P_i is ready!

Topic : Algorithm for Process P_i

$a = b + c;$
$d = a * f;$

Pri

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

Entry

Exit

→ Mut. Exclusion : ✓

→ Progress : ✓

→ Bounded wait : ✓

→ 2-process Soln

→ Busy-waiting (Wastage of CPU Time)

→ May have Problem when implemented on Modern Arch.

2 mins Summary

Topic	One	Need for Synchronization
Topic	Two	Nec. Conditions
Topic	Three	CS Problem
Topic	Four	Requirements
Topic	Five	Software Solution

53

THANK - YOU