UNIT 1
AN INTRODUCTION TO OPERATING SYSTEMS

Application software performs specific task for the user.
System software operates and controls the computer system and provides a platform to run application software.

An **operating system** is a piece of software that manages all the resources of a computer system,both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

Why OS?

1. What if there is no OS?
    a. Bulky and complex app. (Hardware interaction code must be in app's code base)
    b. Resource exploitation by 1 App.
    c. No memory protection.
2. What is an OS made up of?
    a. Collection of system software.

An operating system function -

- Access to the computer hardware.
- interface between the user and the computer hardware
- **Resource management (Aka, Arbitration) (memory, device, file, security, process etc)**
- **Hides the underlying complexity of the hardware. (Aka, Abstraction)**
- facilitates execution of application programs by providing isolation and protection.

User

Application programs
Operating system
Computer hardware

The operating system provides the means for proper use of the resources in the operation of the computer system.

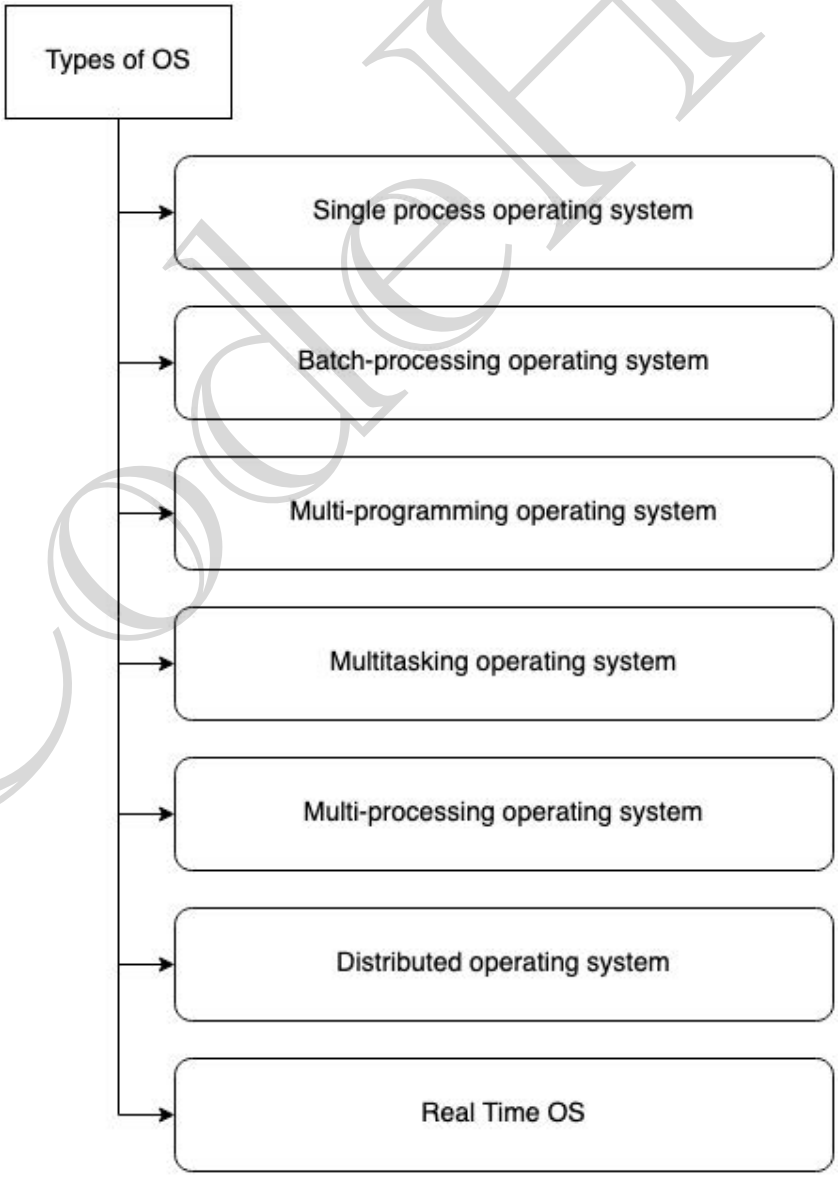
LEC-2: Types of OS

OS goals –

- Maximum CPU utilization
- Less process starvation
- Higher priority job execution

Types of operating systems –

- Single process operating system [MS DOS, 1981]
- Batch-processing operating system [ATLAS, Manchester Univ., late 1950s – early 1960s]
- Multiprogramming operating system [THE, Dijkstra, early 1960s]
- Multitasking operating system [CTSS, MIT, early 1960s]
- Multi-processing operating system [Windows NT]
- Distributed system [LOCUS]
- Real time OS [ATCS]

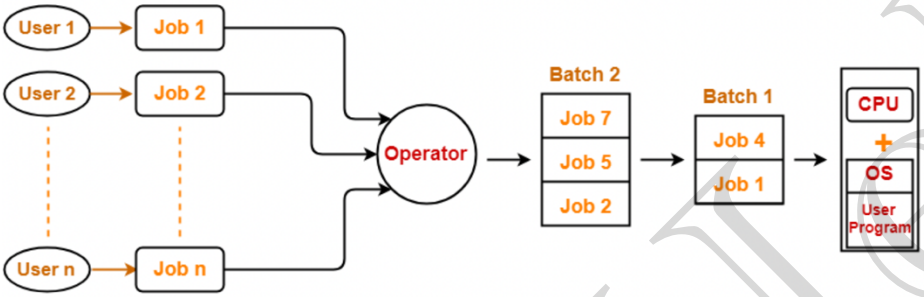


Single process OS, only 1 process executes at a time from the ready queue. [Oldest]

Batch-processing OS,

1. Firstly, user prepares his job using punch cards.
2. Then, he submits the job to the computer operator.
3. Operator collects the jobs from different users and sort the jobs into batches with similar needs.
4. Then, operator submits the batches to the processor one by one.
5. All the jobs of one batch are executed together.

- Priorities cannot be set, if a job comes with some higher priority.
- May lead to starvation. (A batch may take more time to complete)
- CPU may become idle in case of I/O operations.



Multiprogramming increases CPU utilization by keeping multiple jobs (code and data) in the**memory** so that the CPU always has one to execute in case some job gets busy with I/O.

- Single CPU
- Context switching for processes.
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

Multitasking is a logical extension of multiprogramming.

- Single CPU
- Able to run more than one task simultaneously.
- Context switching and time sharing used.
- Increases responsiveness.
- CPU idle time is further reduced.

Multi-processing OS, more than 1 CPU in a single computer.

- Increases reliability, 1 CPU fails, other can work
- Better throughput.
- Lesser process starvation, (if 1 CPU is working on some process, other can be executed on other CPU.

Distributed OS,

- OS manages many bunches of resources, >=1 CPUs, >=1 memory, >=1 GPUs, etc
- **Loosely connected autonomous,** interconnected computer nodes.
- collection of independent, networked, communicating, and physically separate computational nodes.

RTOS

- **Real time** error free, computations within tight-time boundaries.
- Air Traffic control system, ROBOTS etc.

LEC-3: Multi-Tasking vs Multi-Threading

Program: A Program is an executable file which contains a certain set of instructions written to complete the specific job or operation on your computer.

- It's a compiled code. Ready to be executed.
- Stored in Disk

Process: Program under execution. Resides in Computer's primary memory (RAM).

Thread:

- Single sequence stream within a process.
- An independent path of execution in a process.
- Light-weight process.
- Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
- E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads.)

Multi-Tasking	**Multi-Threading**
The execution of more than one task simultaneously is called as multitasking.	A process is divided into several different sub-tasks called as threads, which has its own path of execution. This concept is called as multithreading.
Concept of more than 1 processes being context switched.	Concept of more than 1 thread. Threads are context switched.
No. of CPU 1.	No. of CPU >= 1. (Better to have more than 1)
Isolation and memory protection exists. OS must allocate separate memory and resources to each program that CPU is executing.	**No isolation and memory protection**, resources are shared among threads of that process. OS allocates memory to a process; multiple threads of that process share the same memory and resources allocated to the process.

Thread Scheduling:
Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.

Difference between Thread Context Switching and Process Context Switching:

Thread Context switching	Process context switching
OS saves current state of thread & switches to another thread of same process.	OS saves current state of process & switches to another process by restoring its state.

Doesn’t includes switching of memory address space. (But Program counter, registers & stack are included.)	Includes switching of memory address space.
Fast switching.	Slow switching.
CPU’s cache state is preserved.	CPU’s cache state is flushed.

LEC-4: Components of OS

1. **Kernel:** A **kernel** is that part of the operating system which interacts directly with the hardware andperforms the most crucial tasks.
    a. Heart of OS/Core component
    b. Very first part of OS to load on start-up.
2. **User space:** Where application software runs, apps don't have privileged access to the underlying hardware. It interacts with kernel.
    a. GUI
    b. CLI

A **shell**, also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed.

Functions of Kernel:

1. **Process management:**
    a. Scheduling processes and threads on the CPUs.
    b. Creating & deleting both user and system process.
    c. Suspending and resuming processes
    d. Providing mechanisms for process synchronization or process communication.
2. **Memory management:**
    a. Allocating and deallocating memory space as per need.
    b. Keeping track of which part of memory are currently being used and by which process.
3. **File management:**
    a. Creating and deleting files.
    b. Creating and deleting directories to organize files.
    c. Mapping files into secondary storage.
    d. Backup support onto a stable storage media.
4. **I/O management:** to manage and control I/O operations and I/O devices
    a. Buffering (data copy between two devices), caching and spooling.
        i. Spooling
            1. Within differing speed two jobs.
            2. Eg. Print spooling and mail spooling.
        ii. Buffering
            1. Within one job.
            2. Eg. Youtube video buffering
        iii. Caching
            1. Memory caching, Web caching etc.

Types of Kernels:

1. Monolithic kernel
    a. All functions are in kernel itself.
    b. Bulky in size.
    c. Memory required to run is high.
    d. Less reliable, one module crashes -> whole kernel is down.
    e. High performance as communication is fast. (Less user mode, kernel mode overheads)
    f. Eg. Linux, Unix, MS-DOS.

2. Micro Kernel
    a. Only major functions are in kernel.
        i. Memory mgmt.
        ii. Process mgmt.
    b. File mgmt. and IO mgmt. are in User-space.
    c. smaller in size.
    d. More Reliable
    e. More stable
    f. Performance is slow.
    g. Overhead switching b/w user mode and kernel mode.
    h. Eg. L4 Linux, Symbian OS, MINIX etc.

3. Hybrid Kernel:
    a. Advantages of both worlds. (File mgmt. in User space and rest in Kernel space.)
    b. Combined approach.
    c. Speed and design of mono.
    d. Modularity and stability of micro.
    e. Eg. MacOS, Windows NT/7/10
    f. IPC also happens but lesser overheads
4. Nano/Exo kernels…

Q. How will communication happen between user mode and kernel mode?
Ans. Inter process communication (**IPC**).

1. Two processes executing independently, having independent memory space (Memory protection), But some may need to communicate to work.

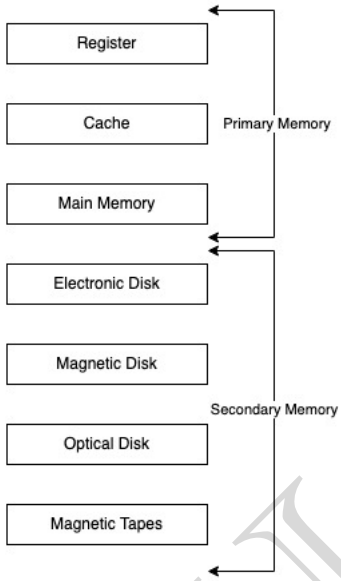
2. Done by shared memory and message passing.

Lec-7: 32-Bit vs 64-Bit OS

1. A 32-bit OS has 32-bit registers, and it can access 2^32 unique memory addresses. i.e., 4GB of physical memory.
2. A 64-bit OS has 64-bit registers, and it can access 2^64 unique memory addresses. i.e., 17,179,869,184 GB of physical memory.
3. 32-bit CPU architecture can process 32 bits of data & information.
4. 64-bit CPU architecture can process 64 bits of data & information.
5. Advantages of 64-bit over the 32-bit operating system:
    a. **Addressable Memory**: 32-bit CPU -> 2^32 memory addresses, 64-bit CPU -> 2^64 memory addresses.
    b. **Resource usage**: Installing more RAM on a system with a 32-bit OS doesn't impact performance. However, upgrade that system with excess RAM to the 64-bit version of Windows, and you'll notice a difference.
    c. **Performance**: All calculations take place in the registers. When you're performing math in your code, operands are loaded from memory into registers. So, having larger registers allow you to perform larger calculations at the same time.
    32-bit processor can execute 4 bytes of data in 1 instruction cycle while 64-bit means that processor can execute 8 bytes of data in 1 instruction cycle.
    (In 1 sec, there could be thousands to billons of instruction cycles depending upon a processor design)
    d. **Compatibility**: 64-bit CPU can run both 32-bit and 64-bit OS. While 32-bit CPU can only run 32-bit OS.
    e. **Better Graphics performance**: 8-bytes graphics calculations make graphics-intensive apps run faster.

Lec-8: Storage Devices Basics

What are the different memory present in the computer system?



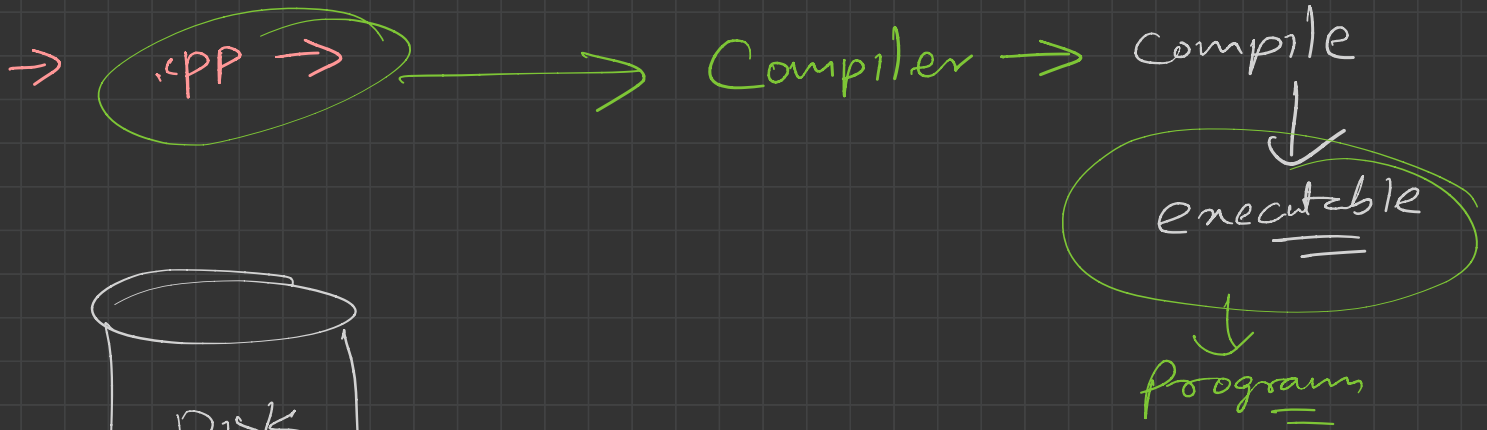
1. **Register**: Smallest unit of storage. It is a part of CPU itself.
   A register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).
   Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU.
2. **Cache**: Additional memory system that temporarily stores frequently used instructions and data for quicker processing by the CPU.
3. **Main Memory**: RAM.
4. **Secondary Memory**: Storage media, on which computer can store data & programs.

Comparison

1. **Cost**:
   a. Primary storages are costly.
   b. Registers are most expensive due to expensive semiconductors & labour.
   c. Secondary storages are cheaper than primary.
2. **Access Speed**:
   a. Primary has higher access speed than secondary memory.
   b. Registers has highest access speed, then comes cache, then main memory.
3. **Storage size**:
   a. Secondary has more space.
4. **Volatility**:
   a. Primary memory is volatile.
   b. Secondary is non-volatile.

Lec-9

Process? What?

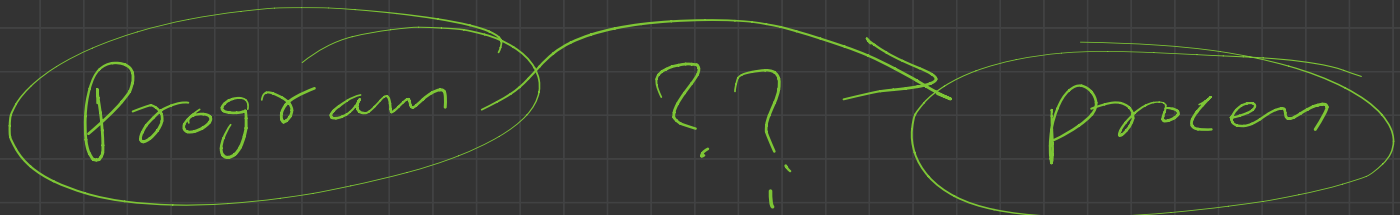


Program e.g. TikTok → OS → Program → Process

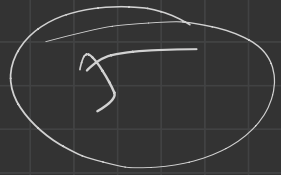
→ Program Under execution → Process

Why? Process ↦

↳ User – Work. → way



How OS creates a process



1. Load the program & static data to memory
   - ↓ Used for initialization

```
char *name = "Lakshay";

int a = 0;
```

(2) Allocate Runtime stack.

- part of memory used for local variable, fn argument & return value

(3) Allocate heap :-

→ part of memory used for dynamic allocation

(9) 10 tasks :-

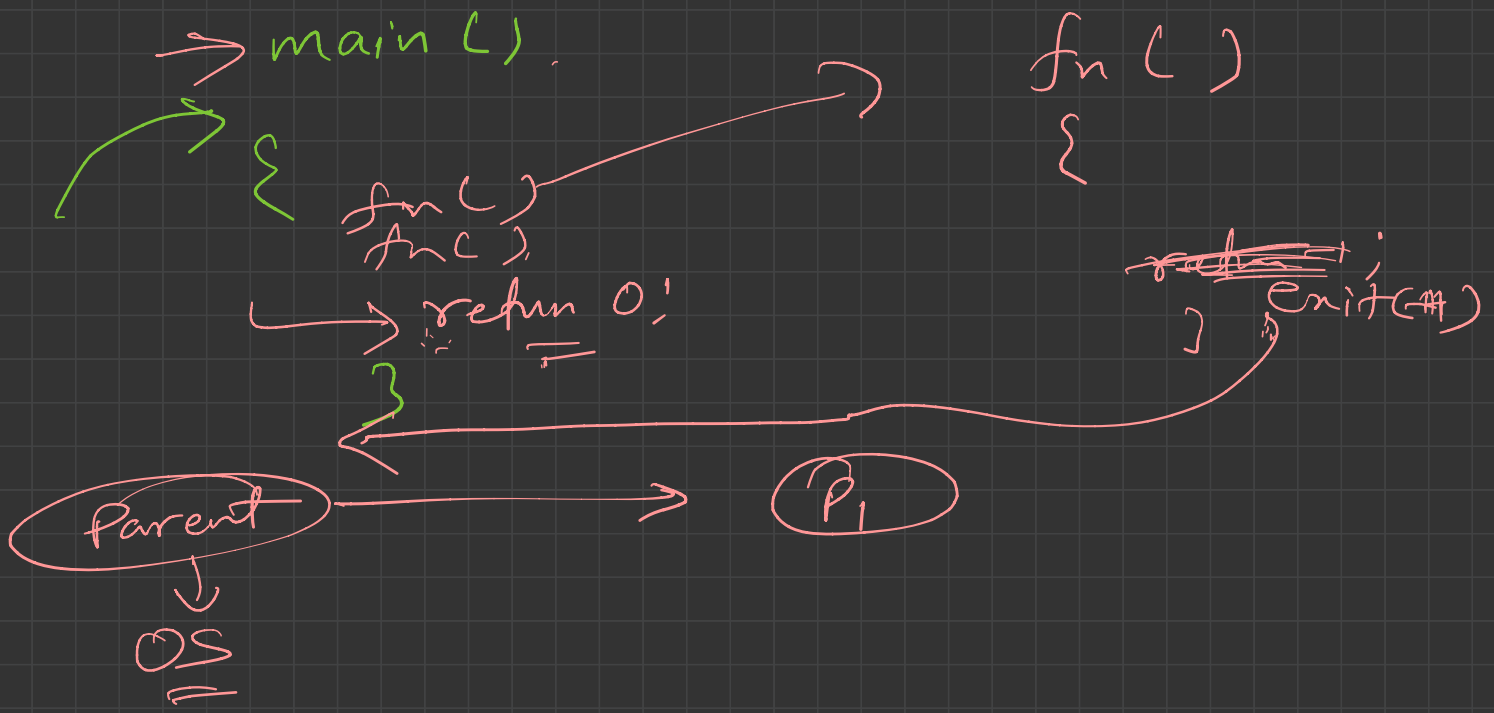
Unix :- i/p → handle →

o/p → o/p →

error :- → handle

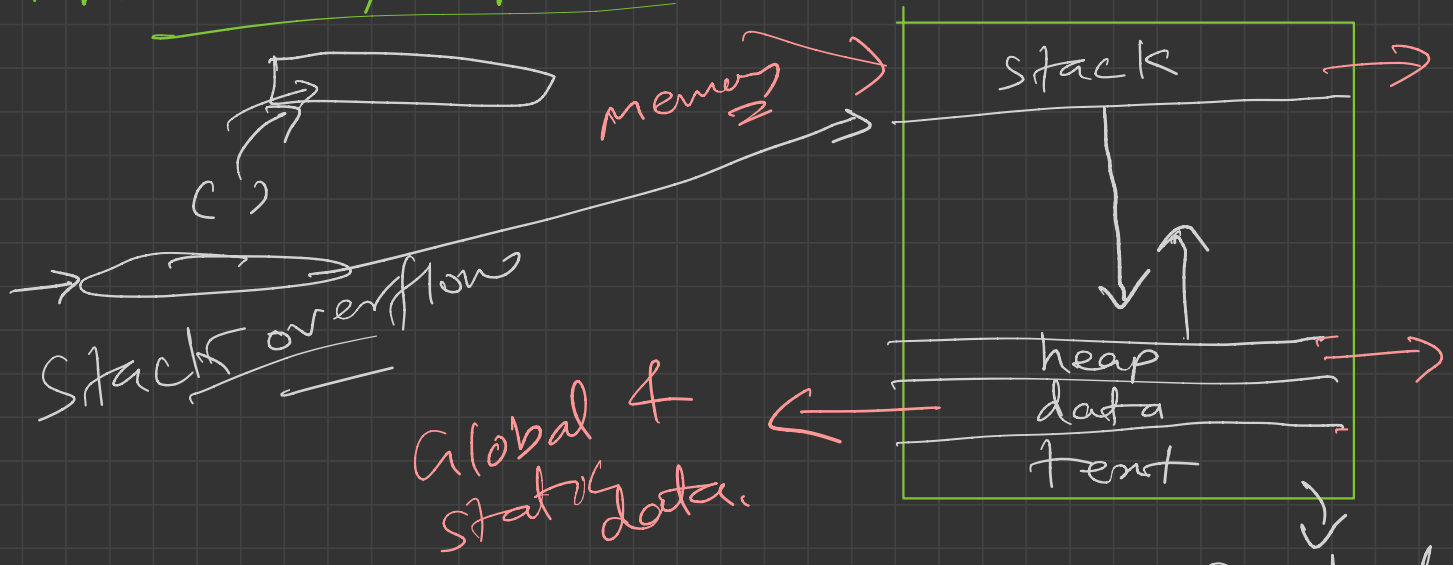
fprintf(stderr, "hang");

(5) OS handoff Control to main ();



Process → memory

*Arch. of Process :-



① Stack overflow — Base case set.

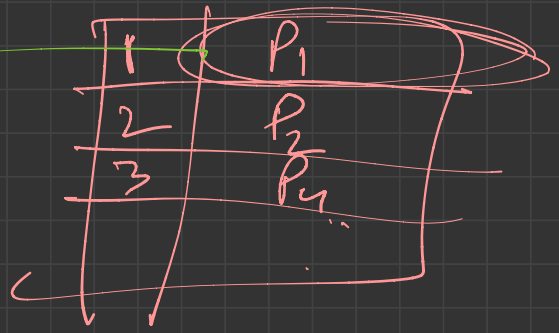
② Out of memory — deallocate unnecessary object.

Attributes of process :-

P_1 P_2 P_3 P_4 → Process Table

PCB ⟸

Process control Block



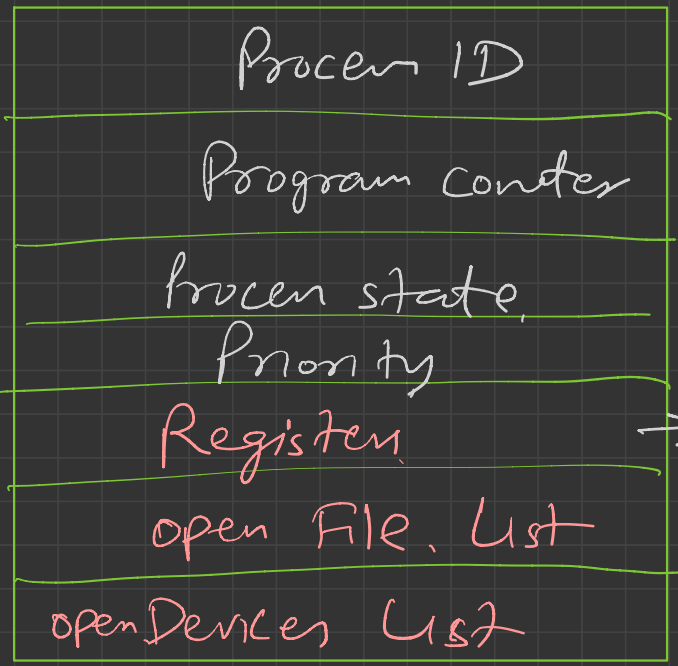
SP → stack pointer

BP → Base pointer

CR →

CPU

D D D



→ unique identifier

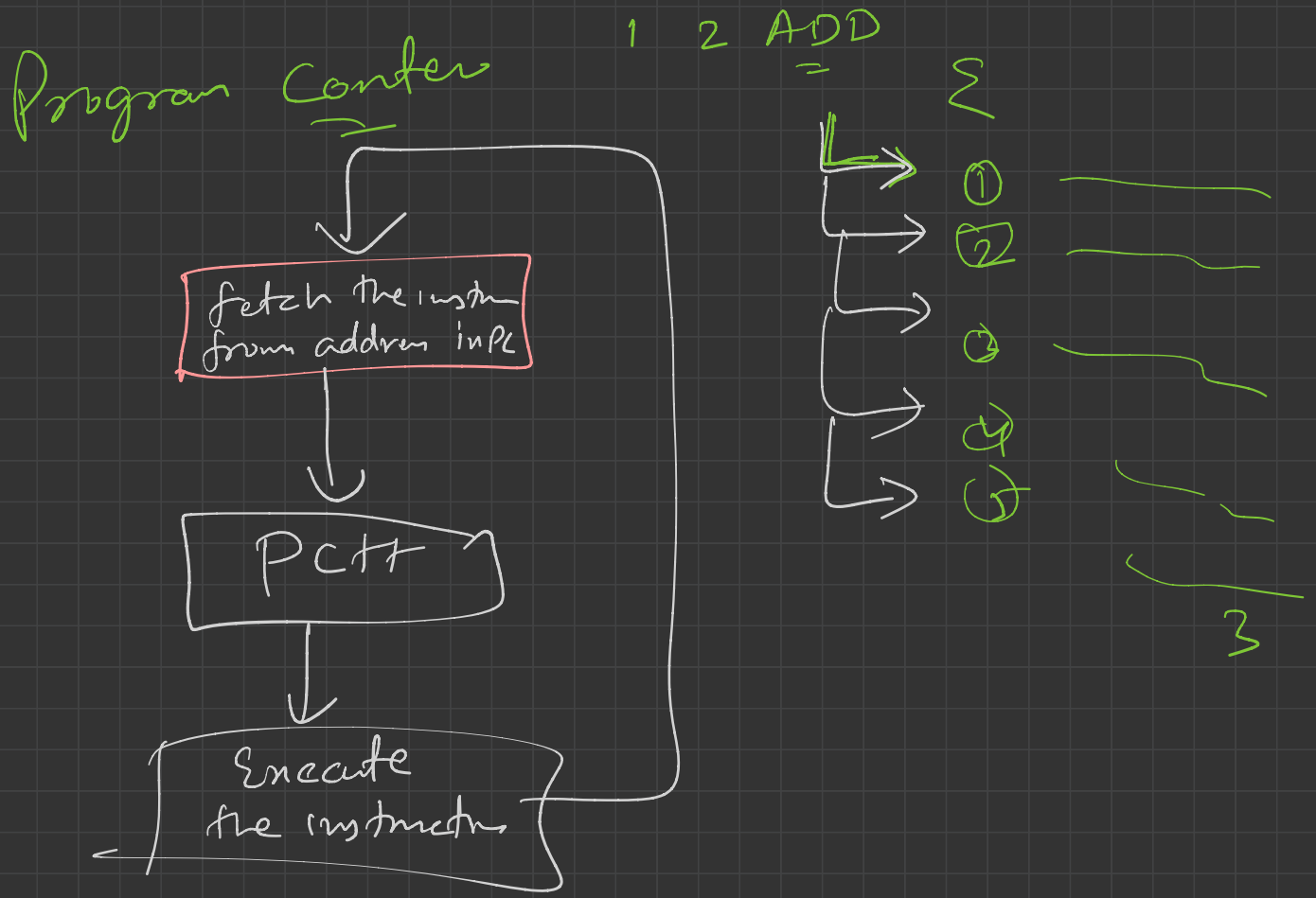
→

New, wait, RUN.

→ Priority.

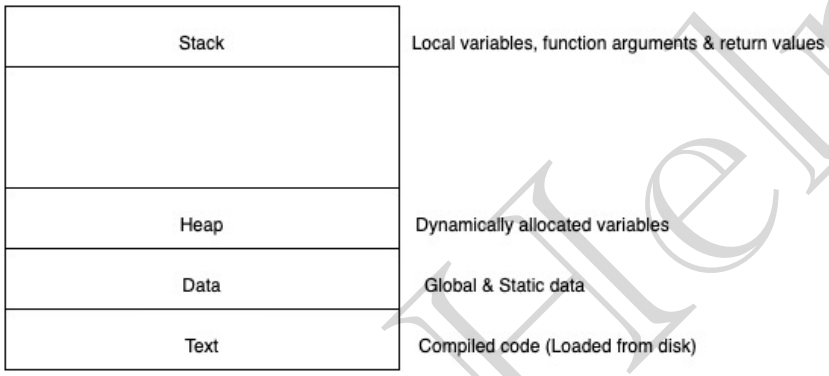
→ Saves the Register CPU.

PCB

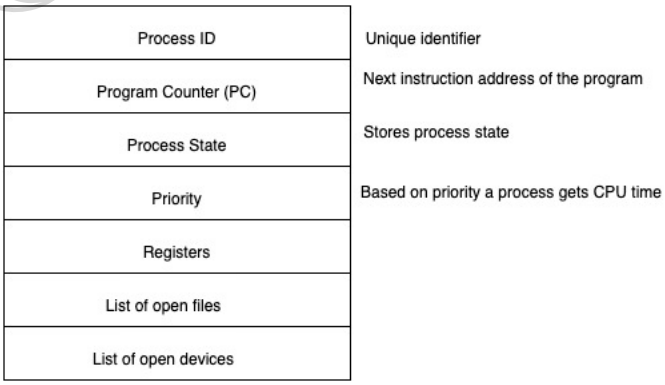

Program Counter
1 2 ADD
=
Σ
fetch the instruction from address in PC
PC++
Execute the instruction
1
2
3
4
5
3

Lec-9: Introduction to Process

1. What is a program? Compiled code, that is ready to execute.
2. What is a process? Program under execution.
3. How OS creates a process? Converting program into a process.
   STEPS:
   a. Load the program & static data into memory.
   b. Allocate runtime stack.
   c. Heap memory allocation.
   d. IO tasks.
   e. OS handoffs control to main ().
4. **Architecture** of process:



5. **Attributes** of process:
   a. Feature that allows identifying a process uniquely.
   b. Process table
      i. All processes are being tracked by OS using a table like data structure.
      ii. Each entry in that table is process control block (PCB).
   c. PCB: Stores info/attributes of a process.
      i. Data structure used for each process, that stores information of a process such as process id, program counter, process state, priority etc.
6. **PCB structure**:



Registers in the PCB, it is a data structure. When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.

Lec-10

Process state :- — Life cycle.

→ Generation → Termination.

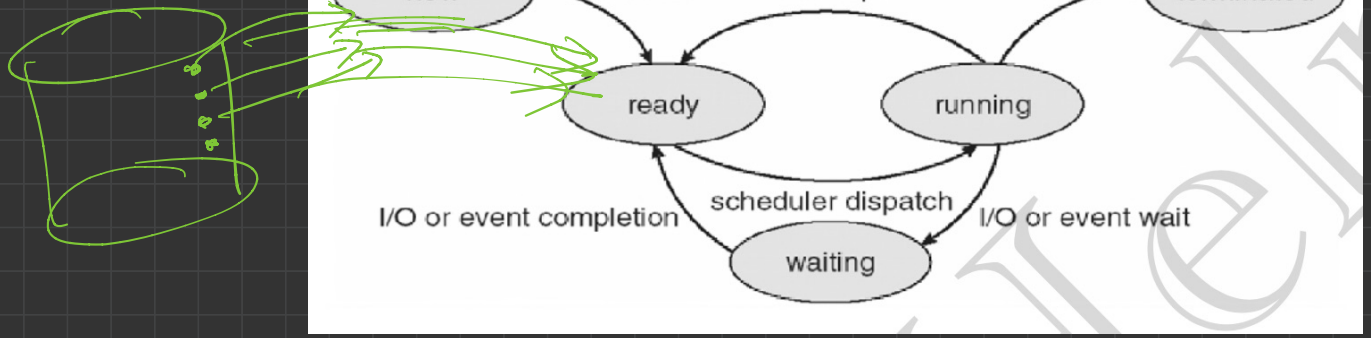
① New :- Program → Process Being done

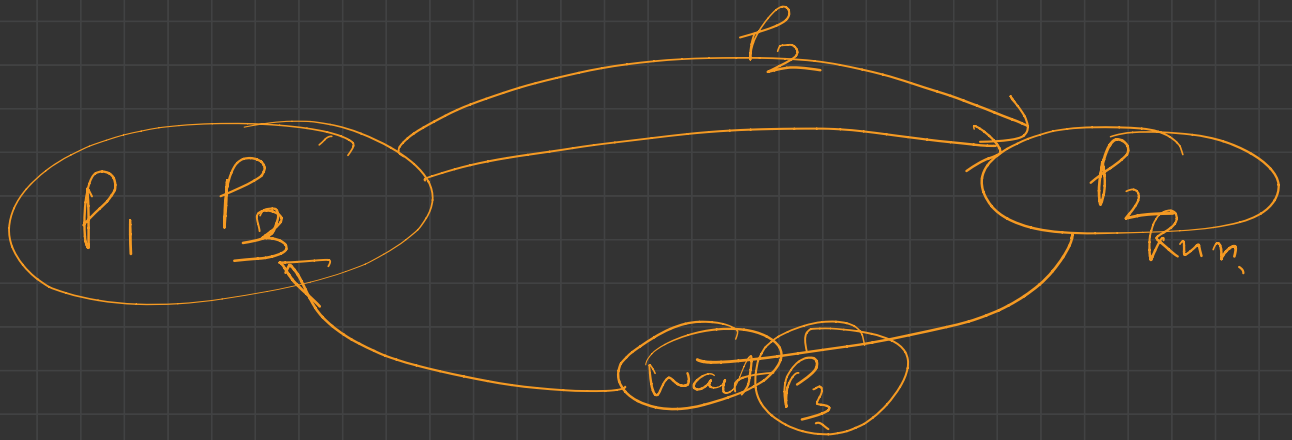
② Ready :- — Process is in Memory
— Ready Queue.
—

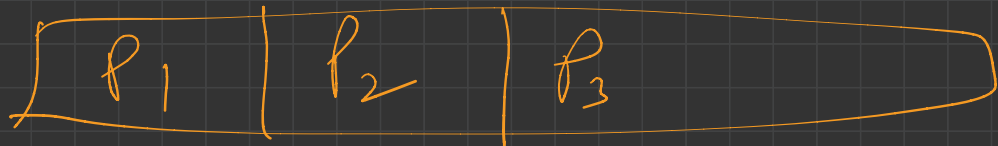
③ Running :→ P_1 → CPU allocate

④ Waiting :→ waiting I/O completion

⑤ Terminated :→ Process finishes


new
admitted
interrupt
exit
terminated
ready
running
I/O or event completion
scheduler dispatch
I/O or event wait
waiting


P1
P3
P2
P2
Run.
Wait
P3



Job que.

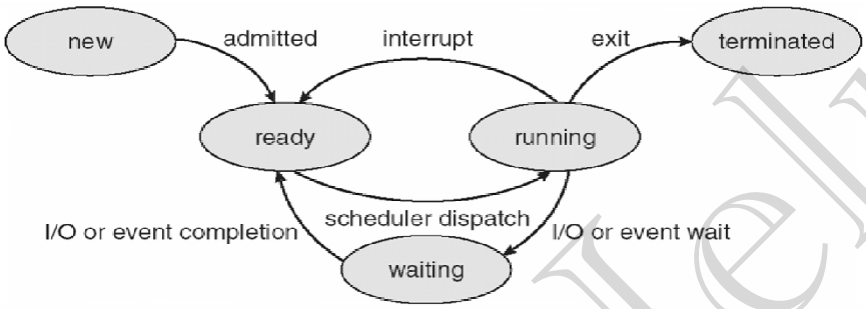
New

① Job scheduler (LTS)

② CPU scheduler (STS)

Lec-10: Process States | Process Queues

1. **Process States**: As process executes, it changes state. Each process may be in one of the following states.
    a. **New**: OS is about to pick the program & convert it into process. OR the process is being created.
    b. **Run**: Instructions are being executed; CPU is allocated.
    c. **Waiting**: Waiting for IO.
    d. **Ready**: The process is in memory, waiting to be assigned to a processor.
    e. **Terminated**: The process has finished execution. PCB entry removed from process table.

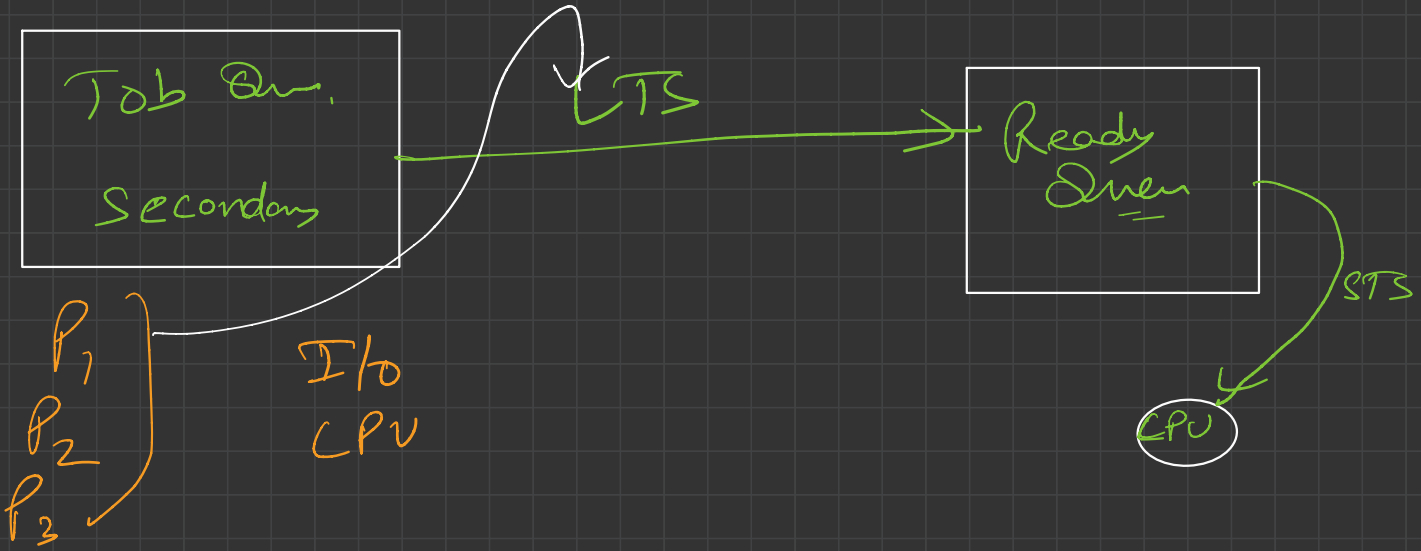


2. **Process Queues:**
    a. Job Queue:
        i. Processes in new state.
        ii. Present in secondary memory.
        iii. **Job Schedular (Long term schedular (LTS))** picks process from the pool and loads them into memory for execution.
    b. Ready Queue:
        i. Processes in Ready state.
        ii. Present in main memory.
        iii. **CPU Schedular (Short-term schedular)** picks process from ready queue and dispatch it to CPU.
    c. Waiting Queue:
        i. Processes in Wait state.
3. **Degree of multi-programming**: The number of processes in the memory.
    a. **LTS** controls degree of multi-programming.
4. **Dispatcher:** The module of OS that gives control of CPU to a process selected by **STS**.

Lec-11

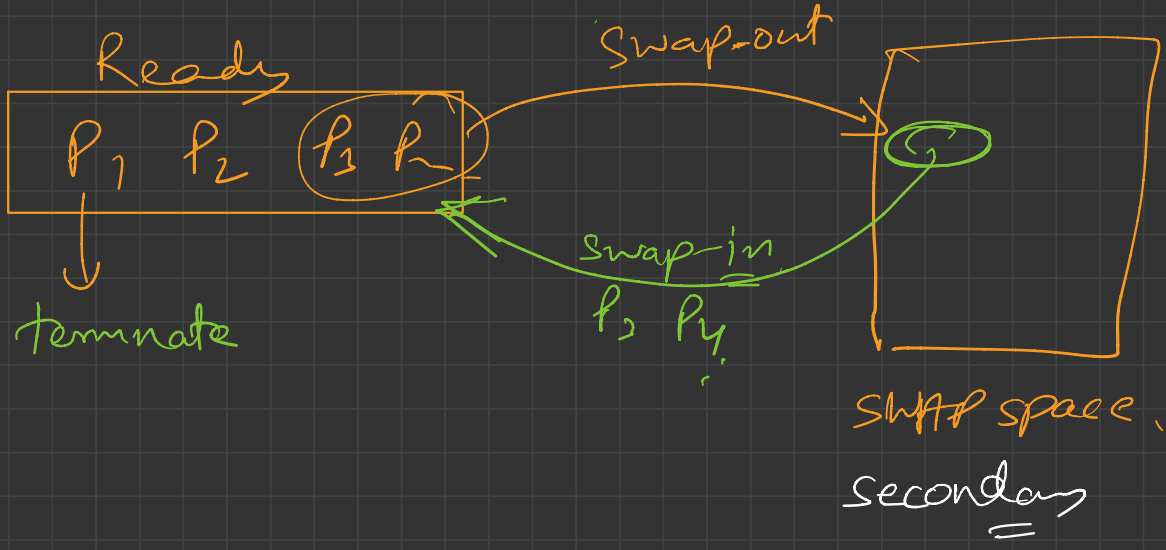
STS → CPU sch.

LTS → Job sch.

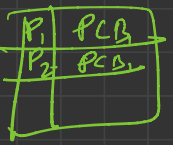


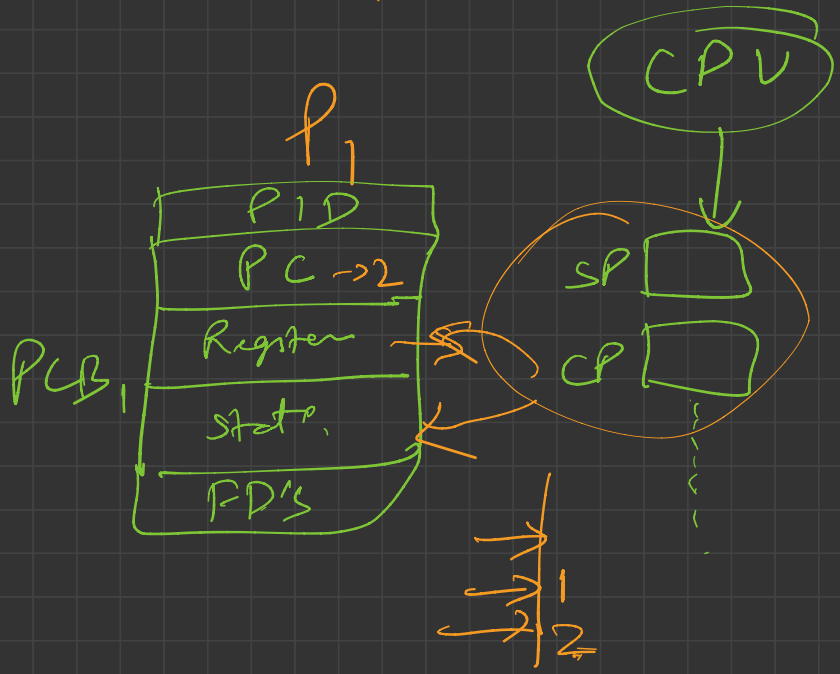
MTS → Medium term scheduler.

degree of multiprogrm ↑ → LTS



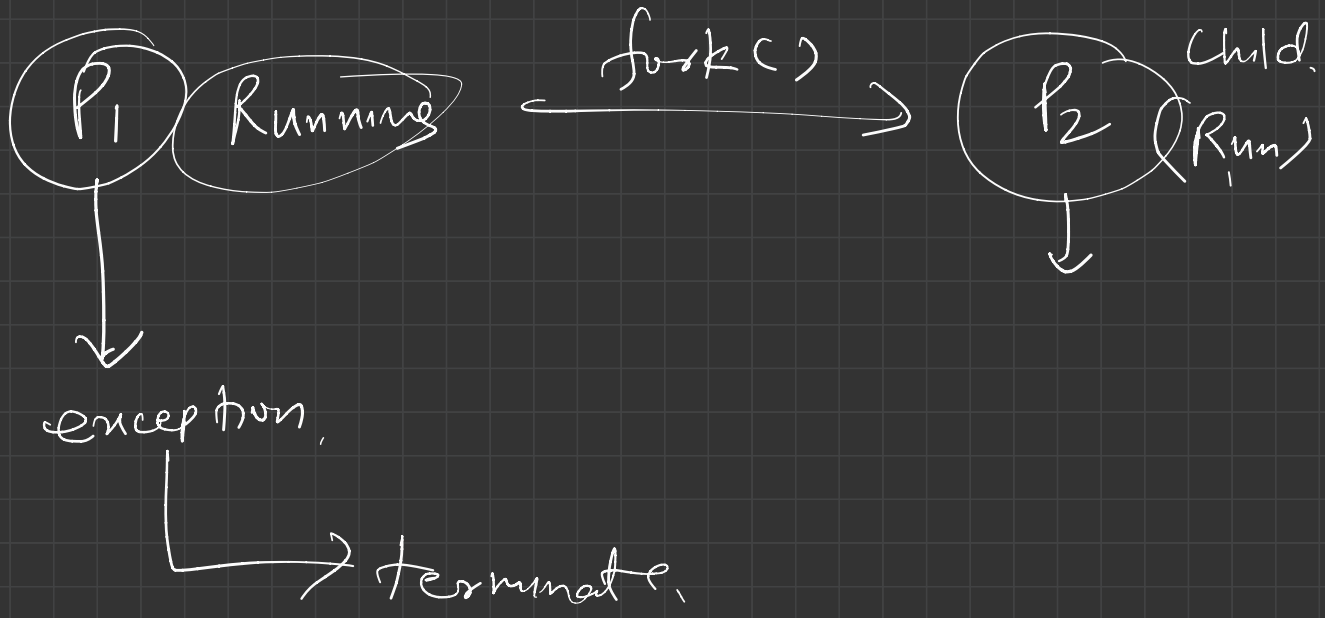
* Context Switching :-

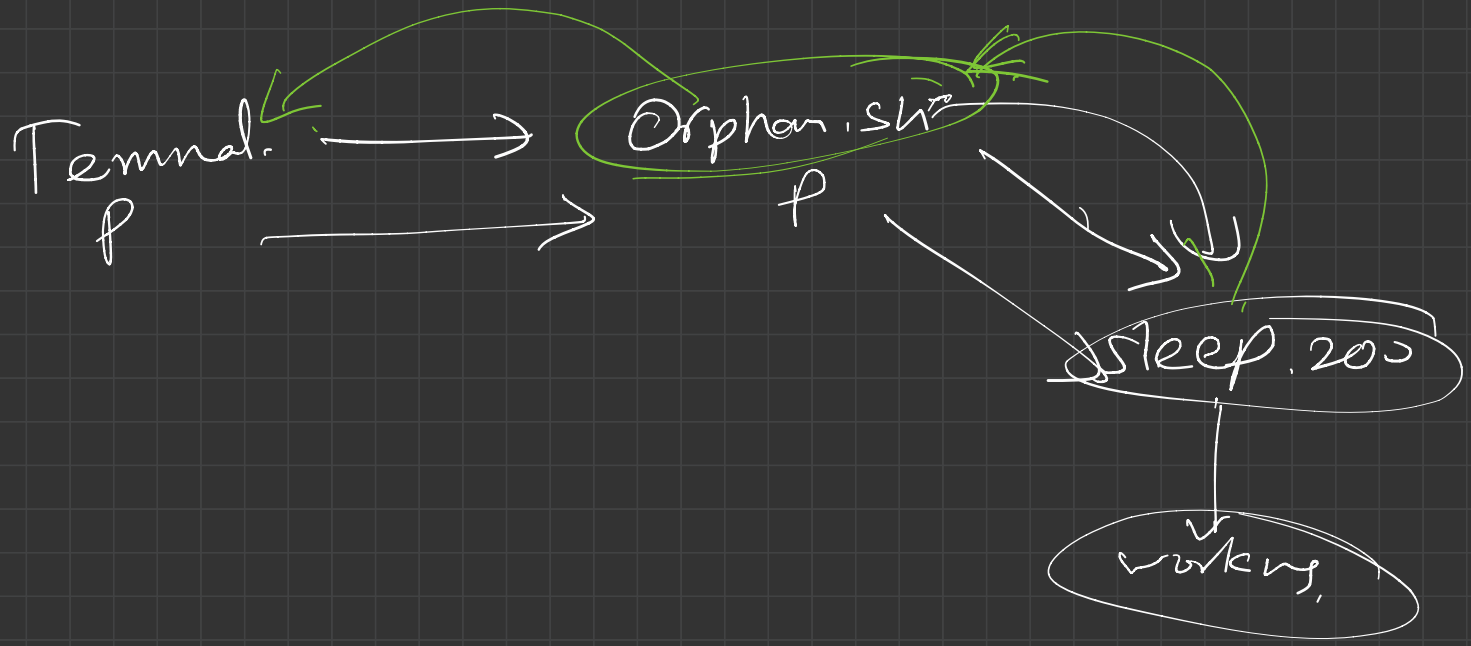


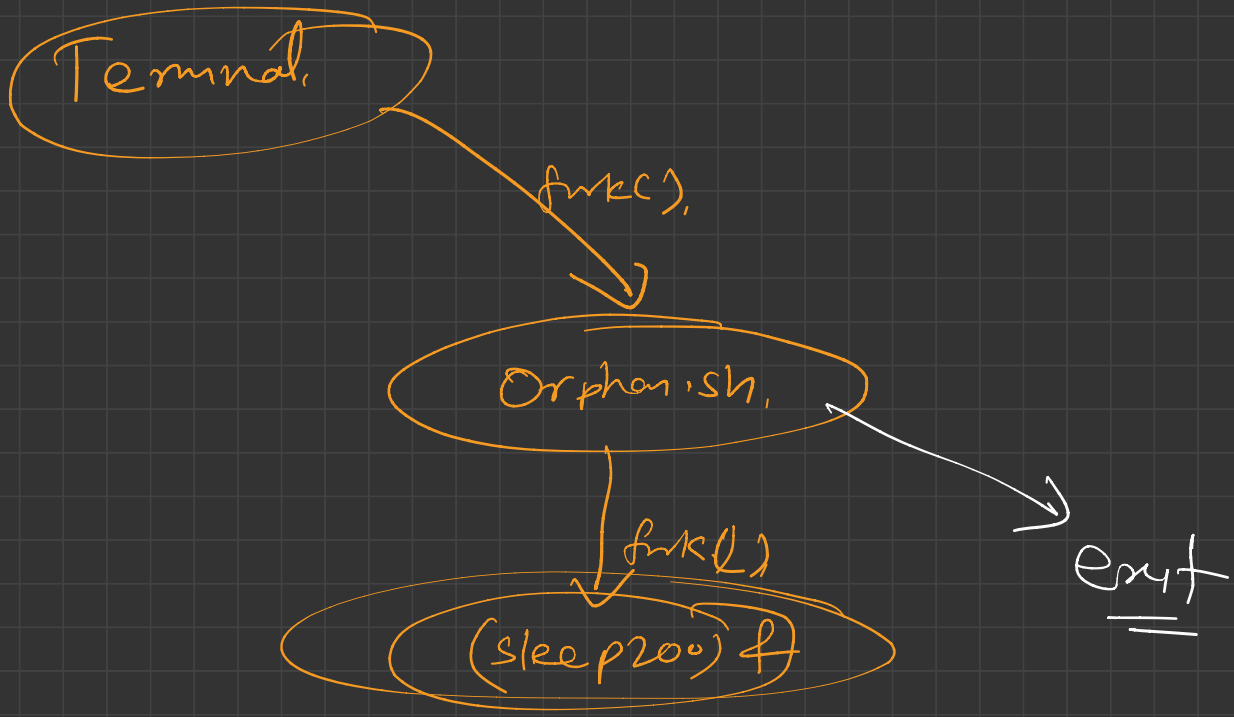


→ Pure overhead !

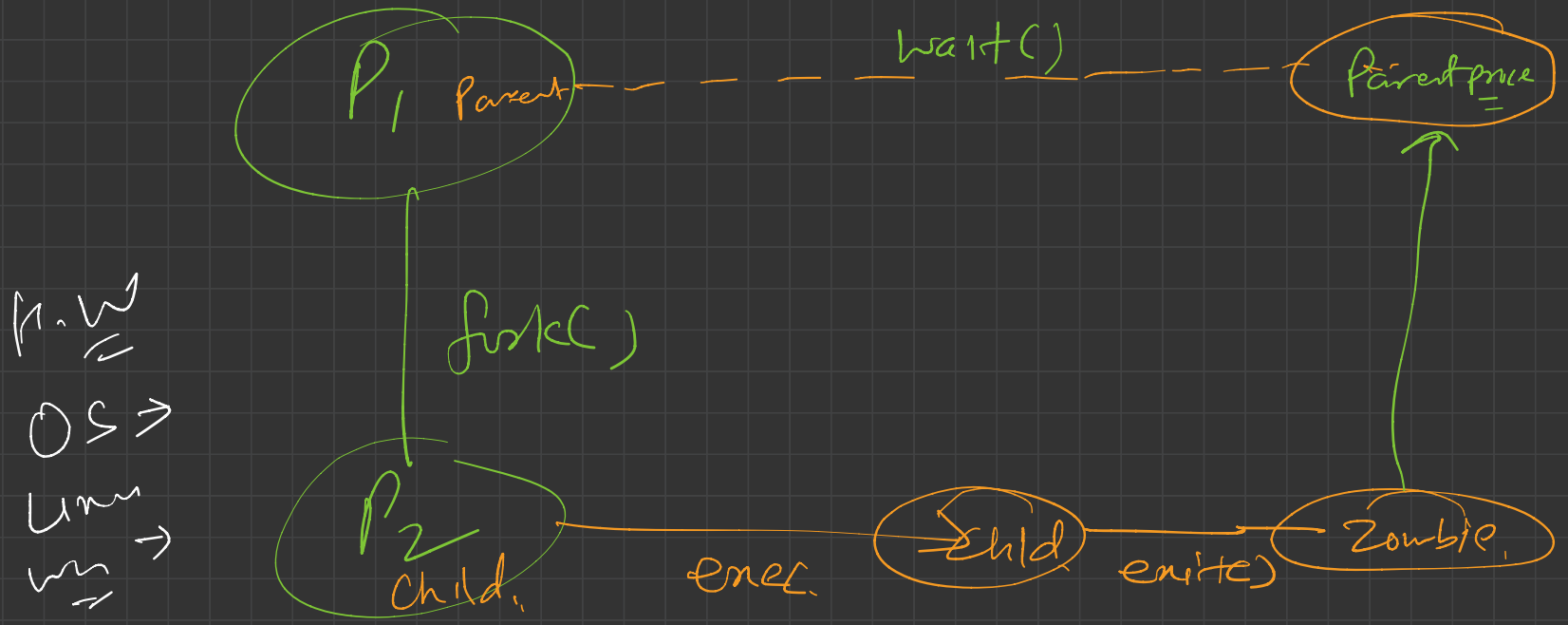
(3) ORPHAN Process :-

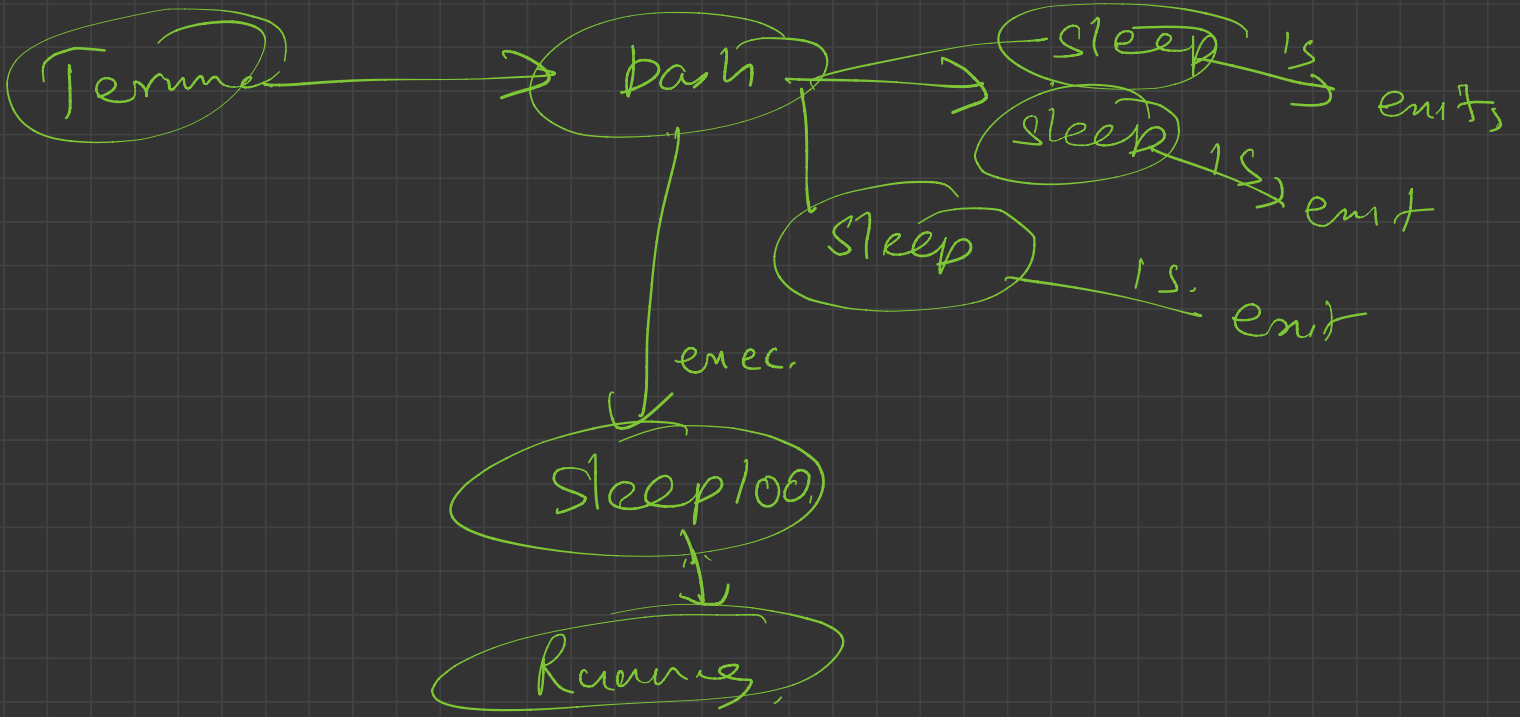



Terminal
P
Orphan.sh
P
sleep 20
working


Terminal
fork()
Orphan.sh
fork()
(sleep200) &
exit

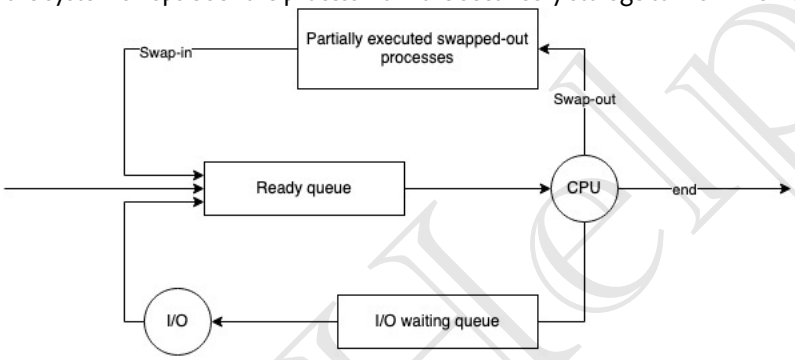
(4) Zombie process




Termine
Bash
sleep
is
emits
sleep
is
emit
sleep
is
exit
exec.
sleep100
Running

LEC-11: Swapping | Context-Switching | Orphan process | Zombie process

1. **Swapping**
   - **a.** Time-sharing system may have medium term schedular (MTS).
   - **b.** Remove processes from memory to reduce degree of multi-programming.
   - **c.** These removed processes can be reintroduced into memory, and its execution can be continued where it left off. This is called **Swapping.**
   - **d.** Swap-out and swap-in is done by MTS.
   - **e.** Swapping is necessary to improve process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.
   - f. Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

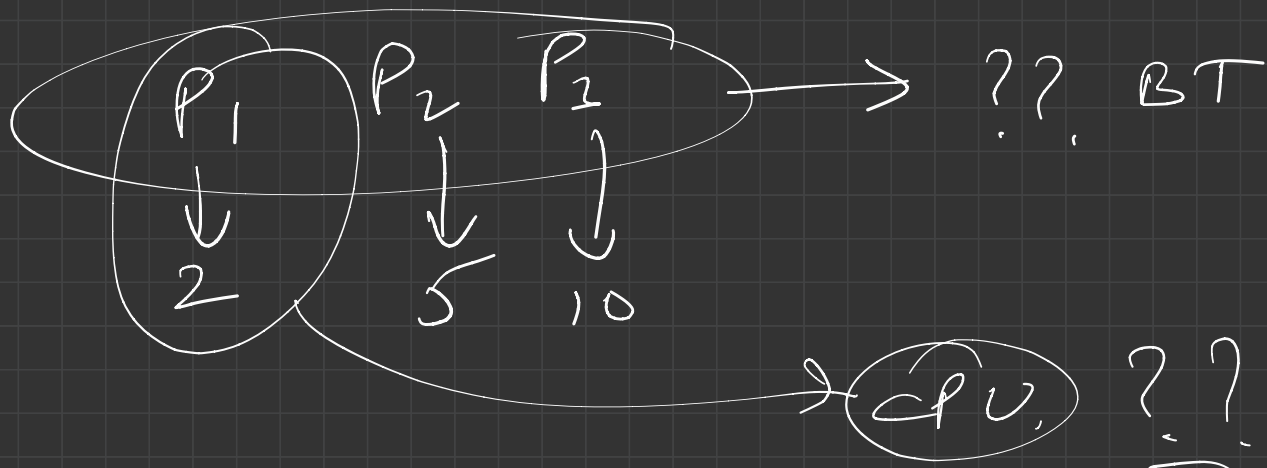


2. **Context-Switching**
   - a. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
   - b. When this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
   - c. It is pure overhead, because the system does no useful work while switching.
   - d. Speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.
3. **Orphan process**
   - **a.** The process whose parent process has been terminated and it is still running.
   - **b.** Orphan processes are adopted by init process.
   - **c.** Init is the first process of OS.
4. **Zombie process / Defunct process**
   - a. A zombie process is a process whose execution is completed but it still has an entry in the process table.
   - b. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as **reaping** the zombie process.
   - c. It is because parent process may call wait () on child process for a longer time duration and child process got terminated much earlier.
   - d. As entry in the process table can only be removed, after the parent process reads the exit status of child process. Hence, the child process remains a zombie till it is removed from the process table.

Lec-13

① SJF (shortest Job first) (Non-preemptive)

– Process with least BT will get CPU.



P	AT	BT	CT	TAT	WT
P_1	0	8	8	8	0
P_2	1	4	12	11	7
P_3	2	9	26	24	15
P_4	3	5	17	14	9

Avg WT 7.75 s.

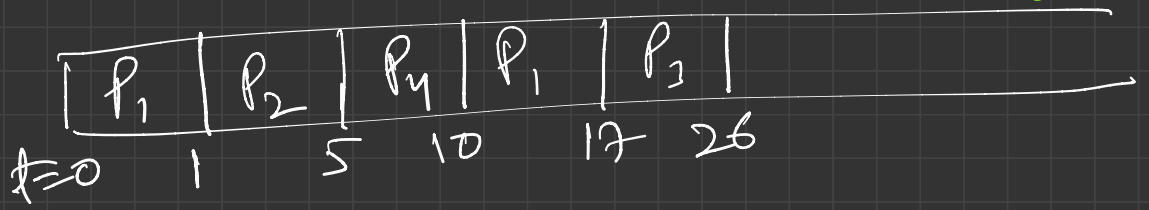
Non-Pr.

Criteria: AT + BT

* Preemptive SJF

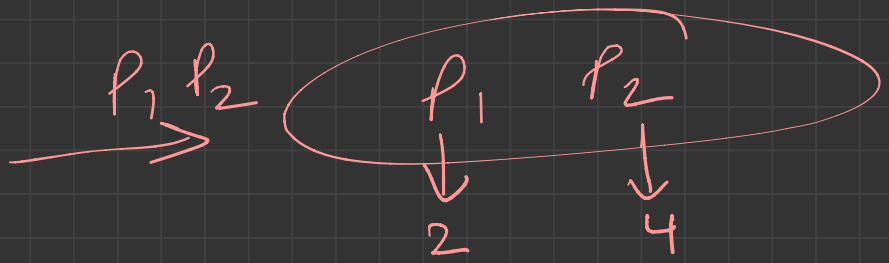
P	AT	BT	CT	TAT	WT
P_1	0	~~8~~ 7	17	17	9
P_2	1	4	5	4	0
P_3	2	9	26	24	15
P_4	3	5	10	7	2

Avg WT :⇒ 6.5 ms



* Priority scheduling :-

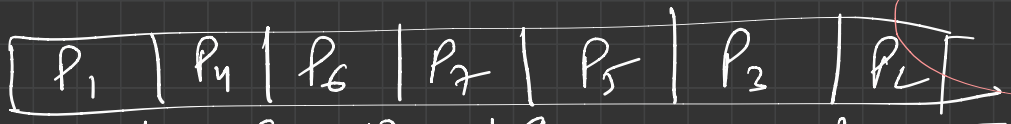
* → Assign priority to each Job



① Non - Preemptive :-

P	Priority	AT	BT	CT	TAT	WT
1	2	0	4	4	4	0
2	4	1	2	25	24	22
3	6	2	3	23	22	19
4	10	3	5	9	6	1
5	8	4	1	20	16	15
6	12	5	4	13	8	4
7	9	6	6	19	13	7

Avg. 9.71 ms



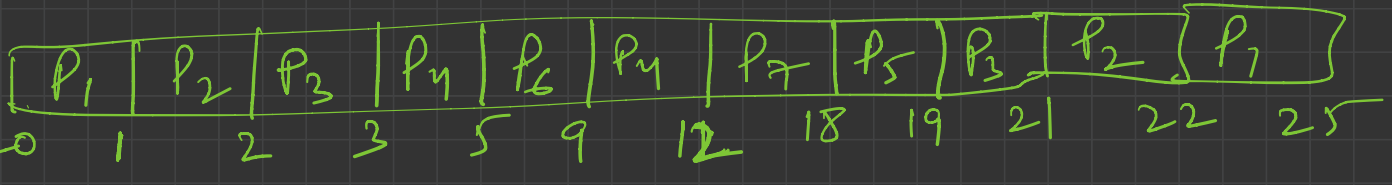
t=0

* Preemptive Priority scheduling

Pno.	P*	AT	BT	CT
1	2	0	13	25
2	4	1	2	22
3	6	2	2	21
4	10	3	5	12
5	8	4	1	19
6	12	5	4	9
7	9	6	6	18

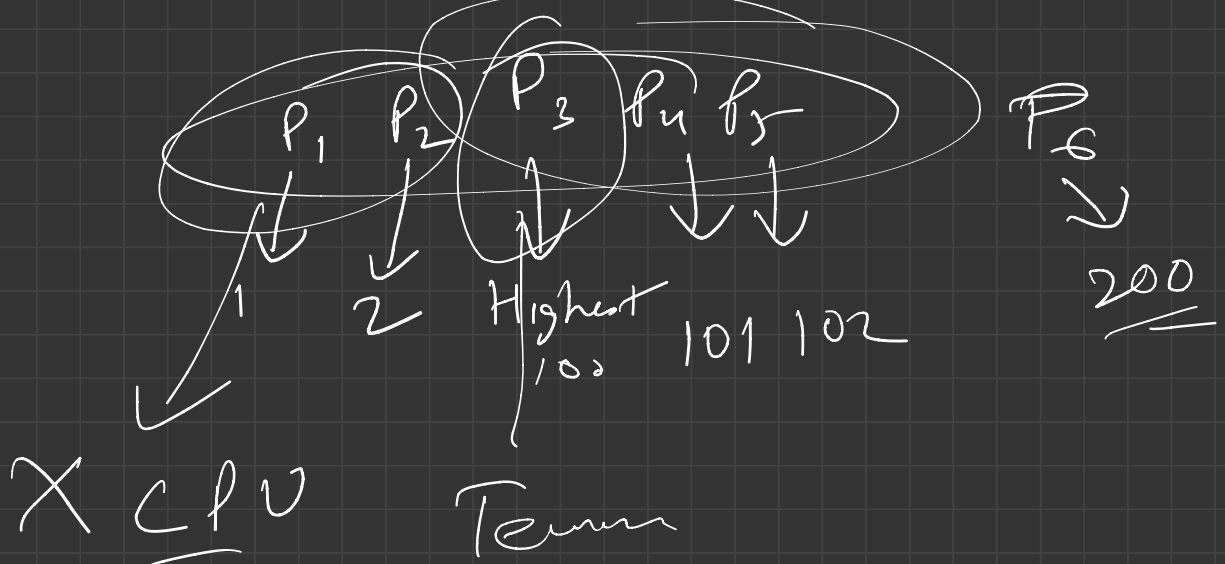
H.W
Avg WT

Avg - 11.4



Biggest Drawback ⇒ Non Pre Preemb

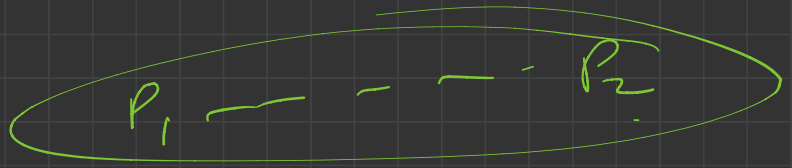
Indefinite waiting or Extreme starvation



Rumor :-

IBM 7094 at MIT

ON → 1967 → Job submit



1973 → check

Lowest Points → Reads

Solution to indefinite waiting

Ageing :->

Gradually Tuning Priorities of Lowest Jobs

15 min → Lowest Priorities Jobs

Priority +1

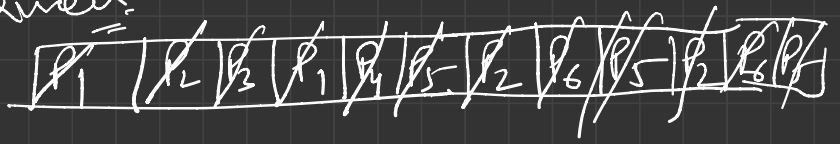
* Round-Robin (RR)
- Most Popular
- FCFS (Preemptive) version
- Criteria: AT + TQ, BTX
- design Time-Sharing
- Easy to implement.

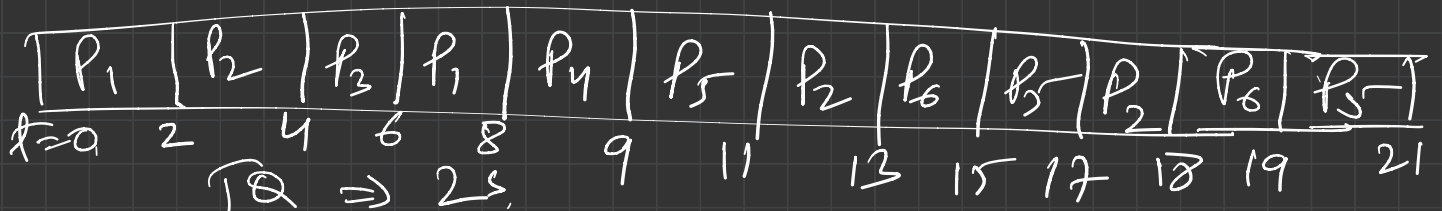
P	AT	BT
1	0	4
2	1	5
3	2	2
4	3	1
5	4	6
6	6	3

TQ = 2s

TQ = 1s

Queue:

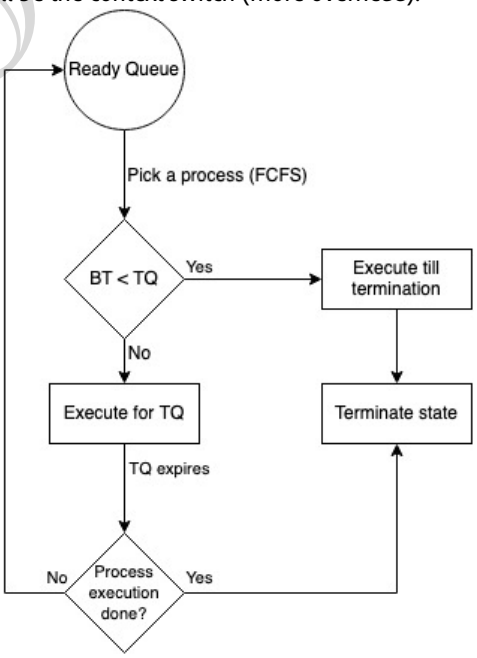




TQ ⇒ 2s.

LEC-13: CPU Scheduling | SJF | Priority | RR

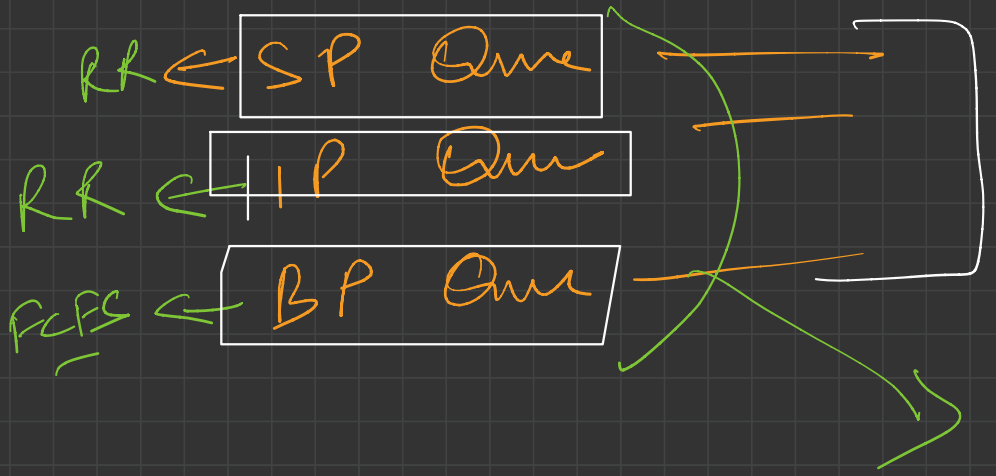
1. Shortest Job First (SJF) [Non-preemptive]
    a. Process with least BT will be dispatched to CPU first.
    b. Must do estimation for BT for each process in ready queue beforehand, Correct estimation of BT is an impossible task (ideally.)
    c. Run lowest time process for all time then, choose job having lowest BT at that instance.
    d. This will suffer from convoy effect as if the very first process which came is Ready state is having a large BT.
    e. Process starvation might happen.
    f. Criteria for SJF algos, AT + BT.
2. SJF [Preemptive]
    a. Less starvation.
    b. No convoy effect.
    c. Gives average WT less for a given set of processes as scheduling short job before a long one decreases the WT of short job more than it increases the WT of the long process.
3. Priority Scheduling [Non-preemptive]
    a. Priority is assigned to a process when it is created.
    b. SJF is a special case of general priority scheduling with priority inversely proportional to BT.
4. Priority Scheduling [Preemptive]
    a. Current RUN state job will be preempted if next job has higher priority.
    b. May cause indefinite waiting (Starvation) for lower priority jobs. (Possibility is they won't get executed ever). (True for both preemptive and non-preemptive version)
        i. Solution: Ageing is the solution.
        ii. Gradually increase priority of process that wait so long. E.g., increase priority by 1 every 15 minutes.
5. Round robin scheduling (RR)
    a. Most popular
    b. Like FCFS but preemptive.
    c. Designed for time sharing systems.
    d. Criteria: AT + time quantum (TQ), Doesn't depend on BT.
    e. No process is going to wait forever, hence very low starvation. [No convoy effect]
    f. Easy to implement.
    g. If TQ is small, more will be the context switch (more overhead).



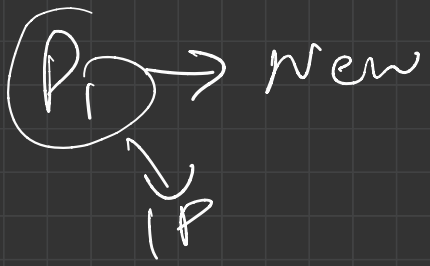
Lec-14

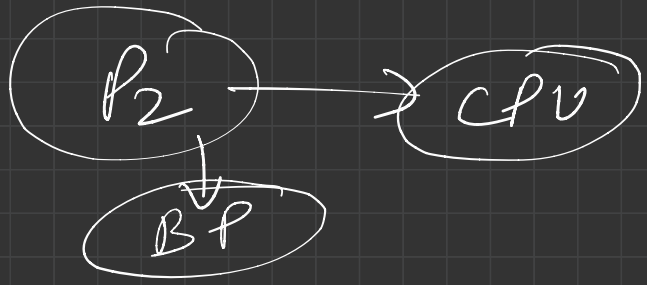
(1) Multi-Level Queue scheduling (MLQ)

1. System process : Created by OS
2. Interactive process :- User input required.
   (foreground) :-
3. Batch process :- X i/p (No i/p)
   (Background)


RR
SP Queue
RR
IP Queue
FCFS
BP Queue
own scheduling
??? schedule?

SP > IP > BP


P1
New
IP


P2
CPU
BP

* Multi-level feedback Queue scheduling

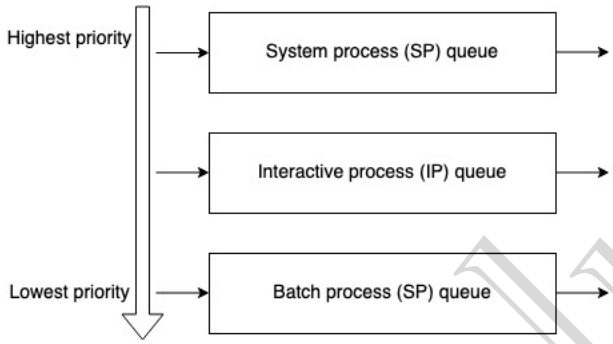
- Multiple sub queues
- Inter queue movement is allowed.
- separate process based on BT.
  - BT ↑ → Lower queues
  - I/O bound & interative process higher priority.
  - Ageing method Lower priority process priority ↑.
- flexible
- Configurable OS design

Design of MLFQ :

1. No. of Queues.
2. Scheduling Algo in each Qu.
3. Method to upgrade a process to a higher Queue.
4. demote a process to Lower priority
5. Process P_i → which Queue will it be Pushed.

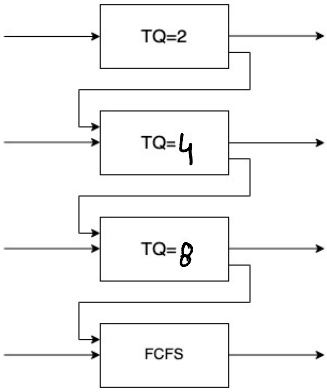
LEC-14: MLQ | MLFQ

1. Multi-level queue scheduling (**MLQ**)
    a. Ready queue is divided into multiple queues depending upon priority.
    b. A process is permanently assigned to one of the queues (inflexible) based on some property of process, memory, size, process priority or process type.
    c. Each queue has its own scheduling algorithm. E.g., SP -> RR, IP -> RR & BP -> FCFS.



    d. System process: Created by OS (Highest priority)
    Interactive process (Foreground process): Needs user input (I/O).
    Batch process (Background process): Runs silently, no user input required.
    e. Scheduling among different sub-queues is implemented as **fixed priority preemptive** scheduling. E.g., foreground queue has absolute priority over background queue.
    f. If an interactive process comes & batch process is currently executing. Then, batch process will be preempted.
    g. Problem: Only after completion of all the processes from the top-level ready queue, the further level ready queues will be scheduled.
    This came starvation for lower priority process.
    h. Convoy effect is present.

2. Multi-level feedback queue scheduling (**MLFQ**)
    a. Multiple sub-queues are present.
    b. Allows the process to move between queues. The idea is to separate processes according to the characteristics of their BT. If a process uses too much CPU time, it will be moved to lower priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queue.
    In addition, a process that waits too much in a lower-priority queue may be moved to a higher priority queue. This form of ageing prevents starvation.
    c. Less starvation then MLQ.
    d. It is flexible.
    e. Can be configured to match a specific system design requirement.
    Sample MLFQ design:

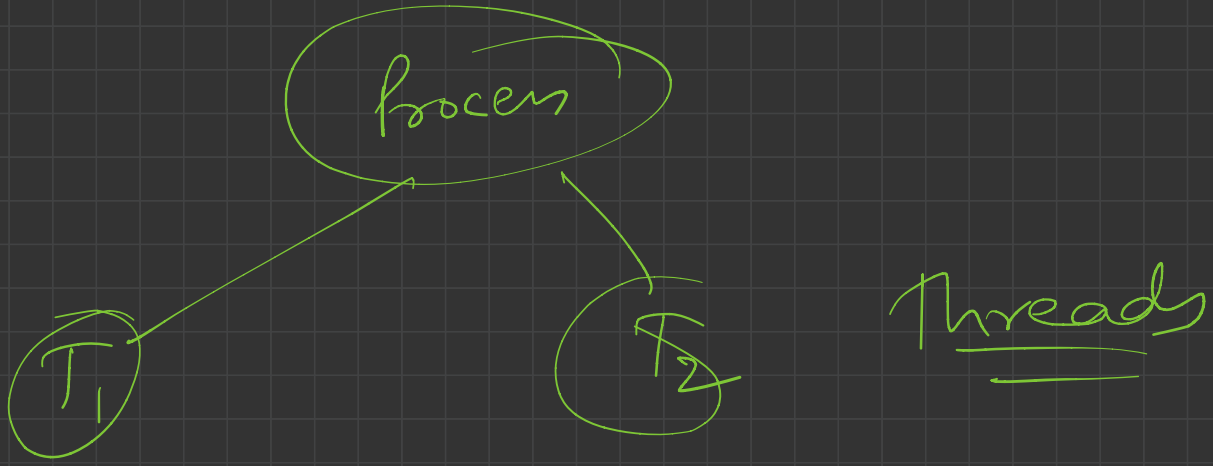


3. Comparison:

	FCFS	SJF	PSJF	Priority	P-Priority	RR	MLQ	MLFQ
Design	Simple	Complex	Complex	Complex	Complex	Simple	Complex	Complex
Preemption	No	No	Yes	No	Yes	Yes	Yes	Yes
Convoy effect	Yes	Yes	No	Yes	Yes	No	Yes	Yes
Overhead	No	No	Yes	No	Yes	Yes	Yes	Yes

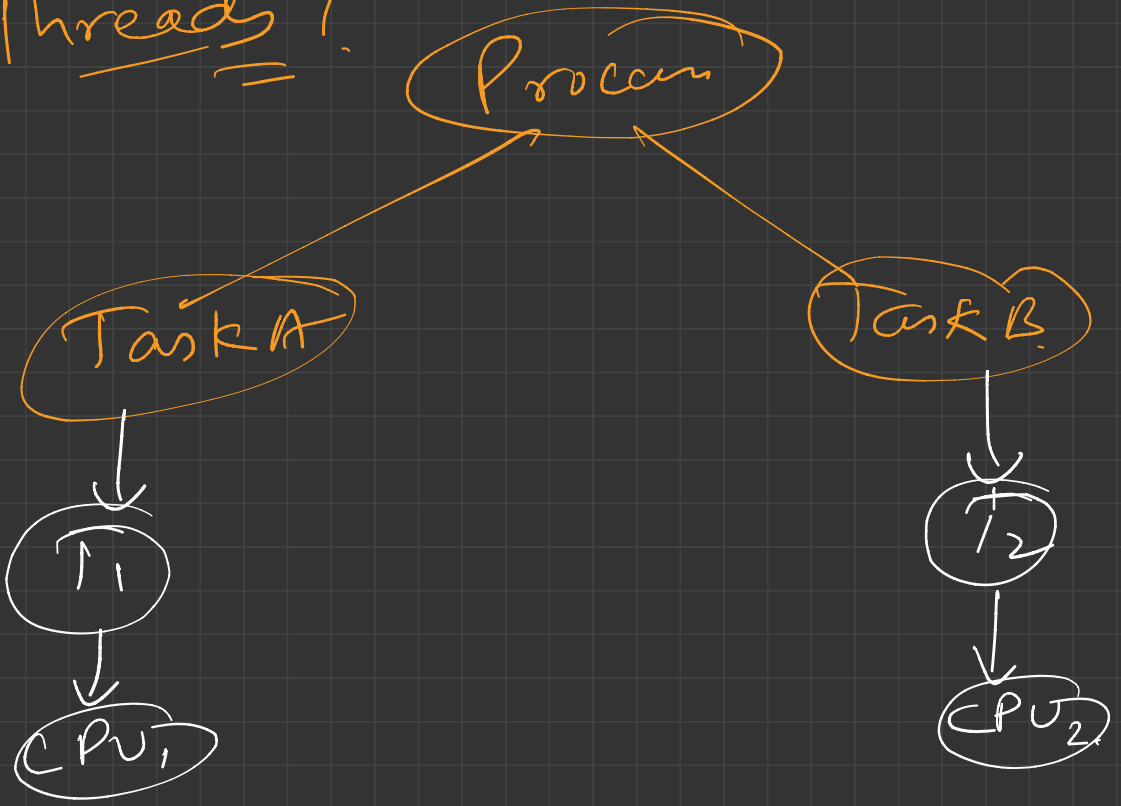
Lec-15

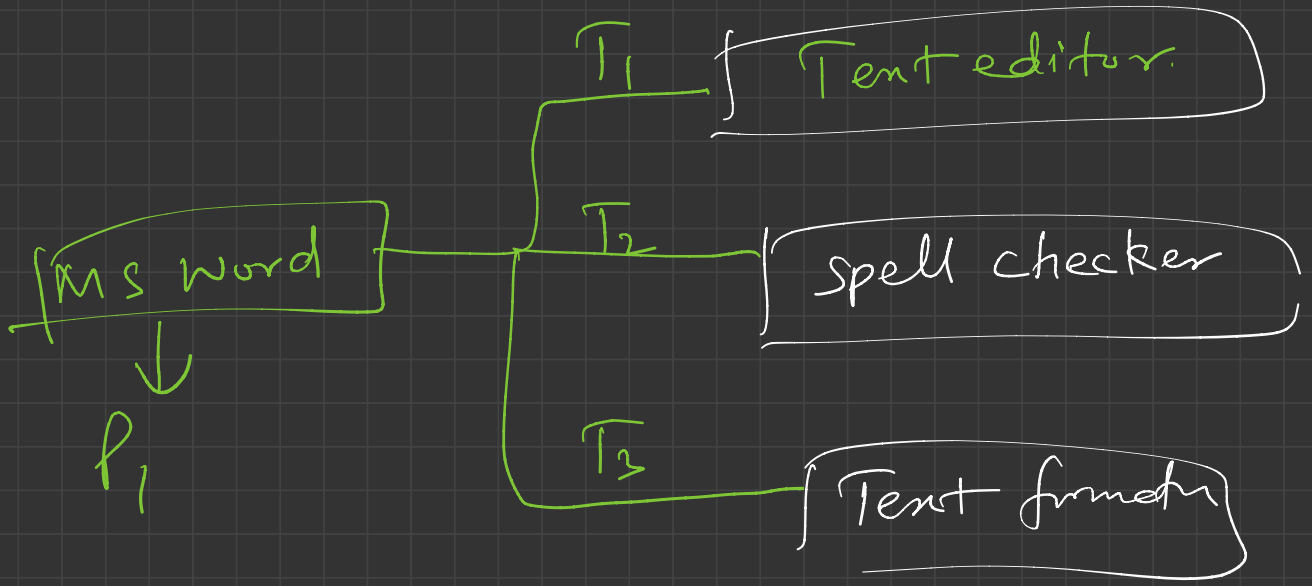
Concurrency :- Multiple instruction at the same time.



Threads

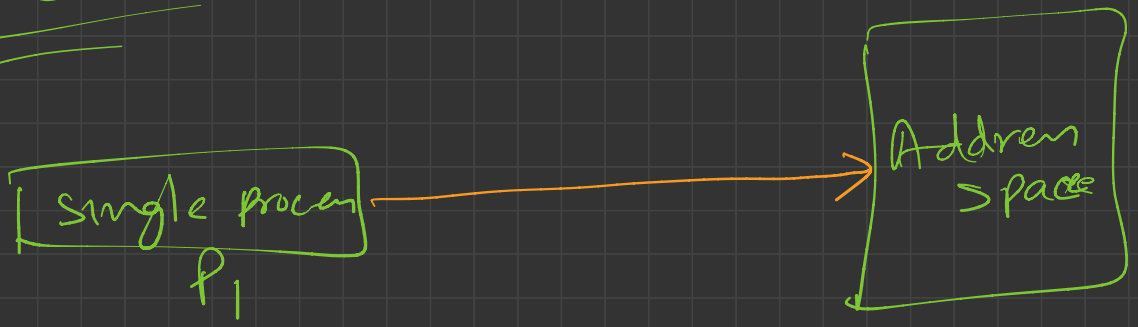
Threads ?



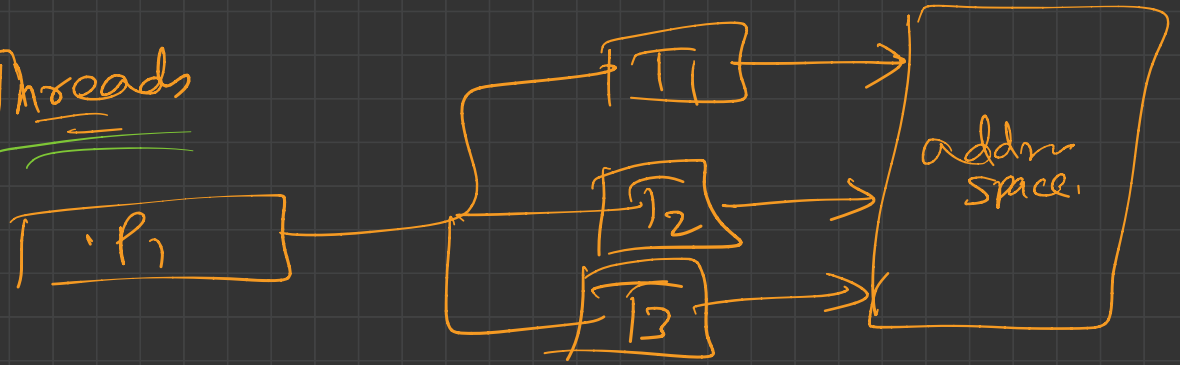


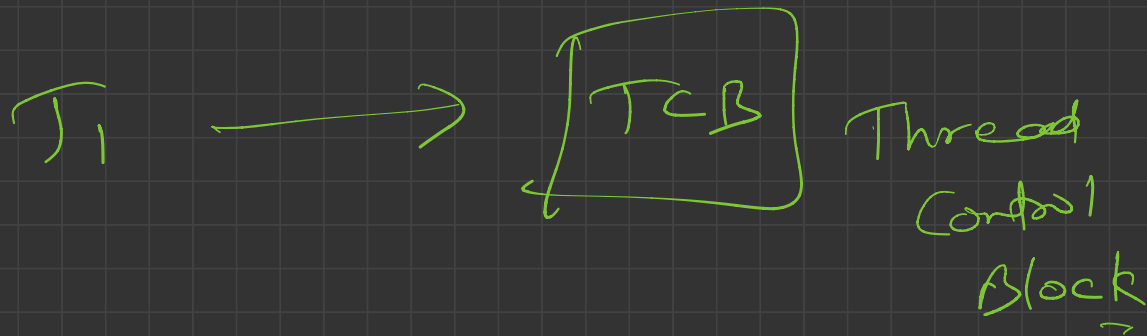
Text editor → spell checker → Text formatter

Process

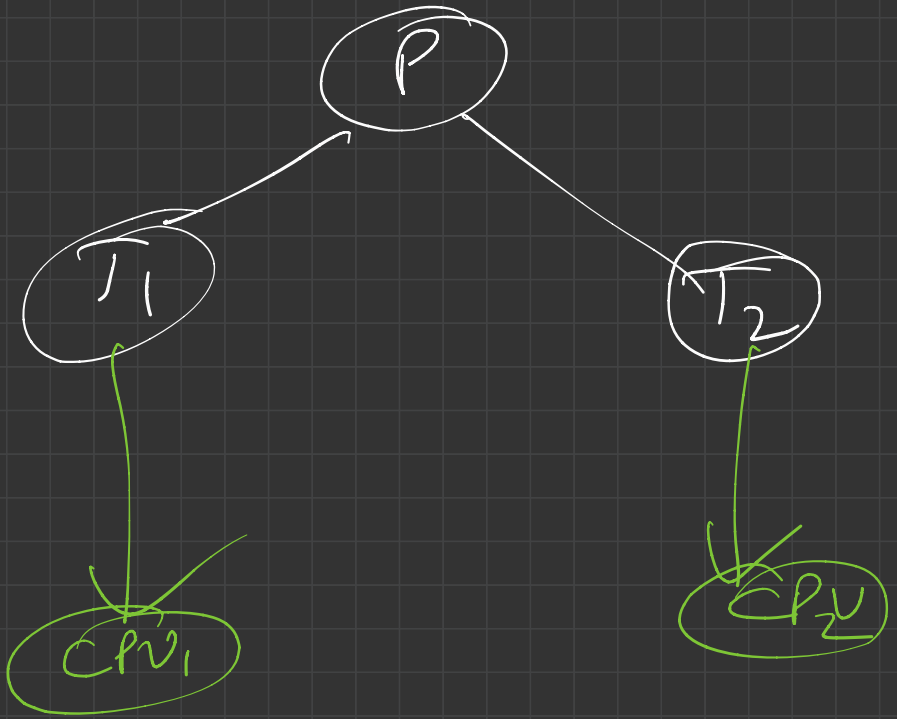


Threads

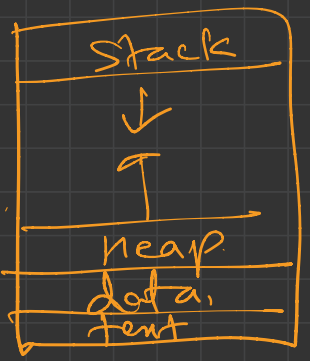




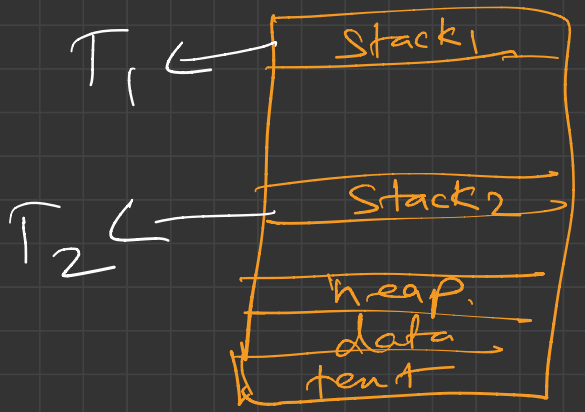
Single CPU → Multi threading ?
Benefit


P
T1
T2
CPU1
CPU2

Process



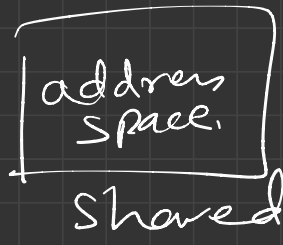
Thread



Benefits of MT

(1) Responsiveness = Interactive

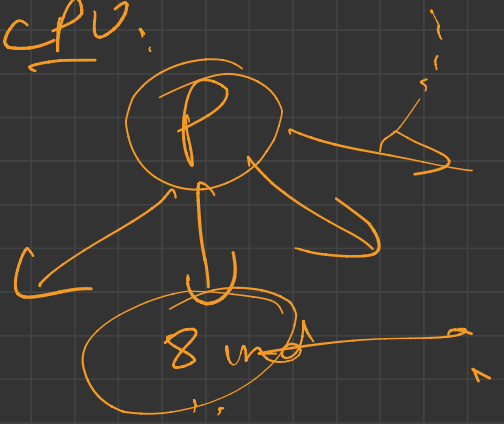
(2) Resource sharing →

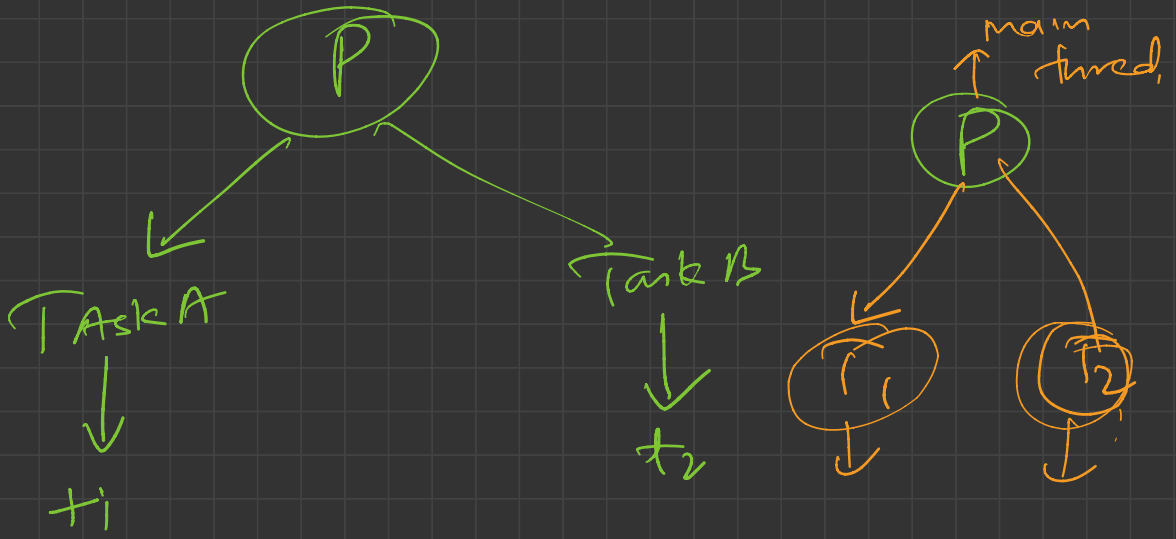


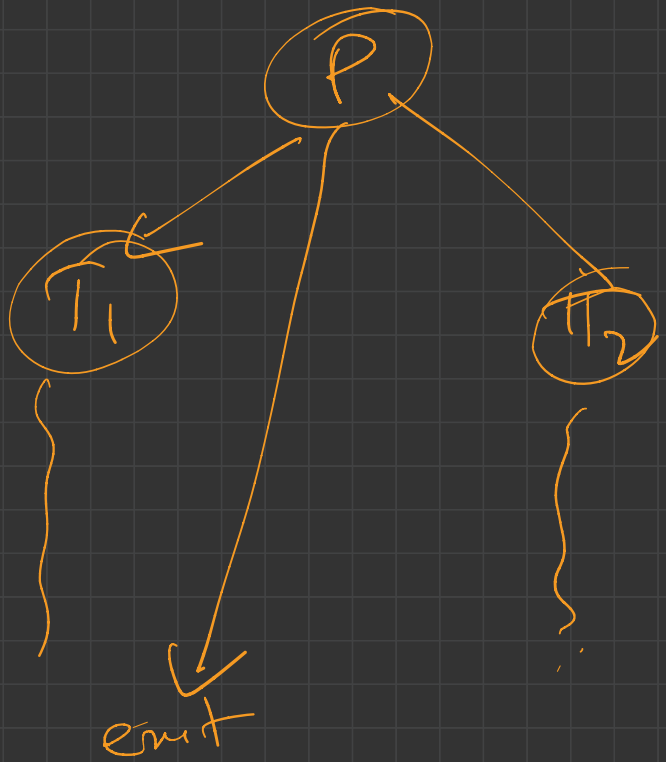
③ Economy :→

④ Threads Better utilizes Multi Core CPU.

8 →




P
Task A
t1
Task B
t2
main
thread
P
T1
T2

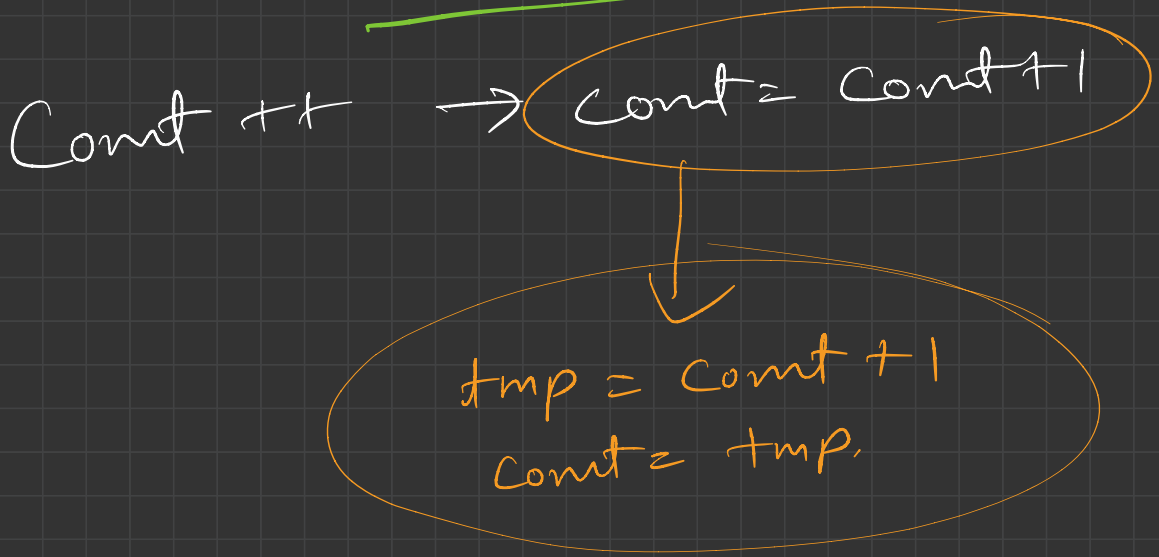

P
T1
T2
exit

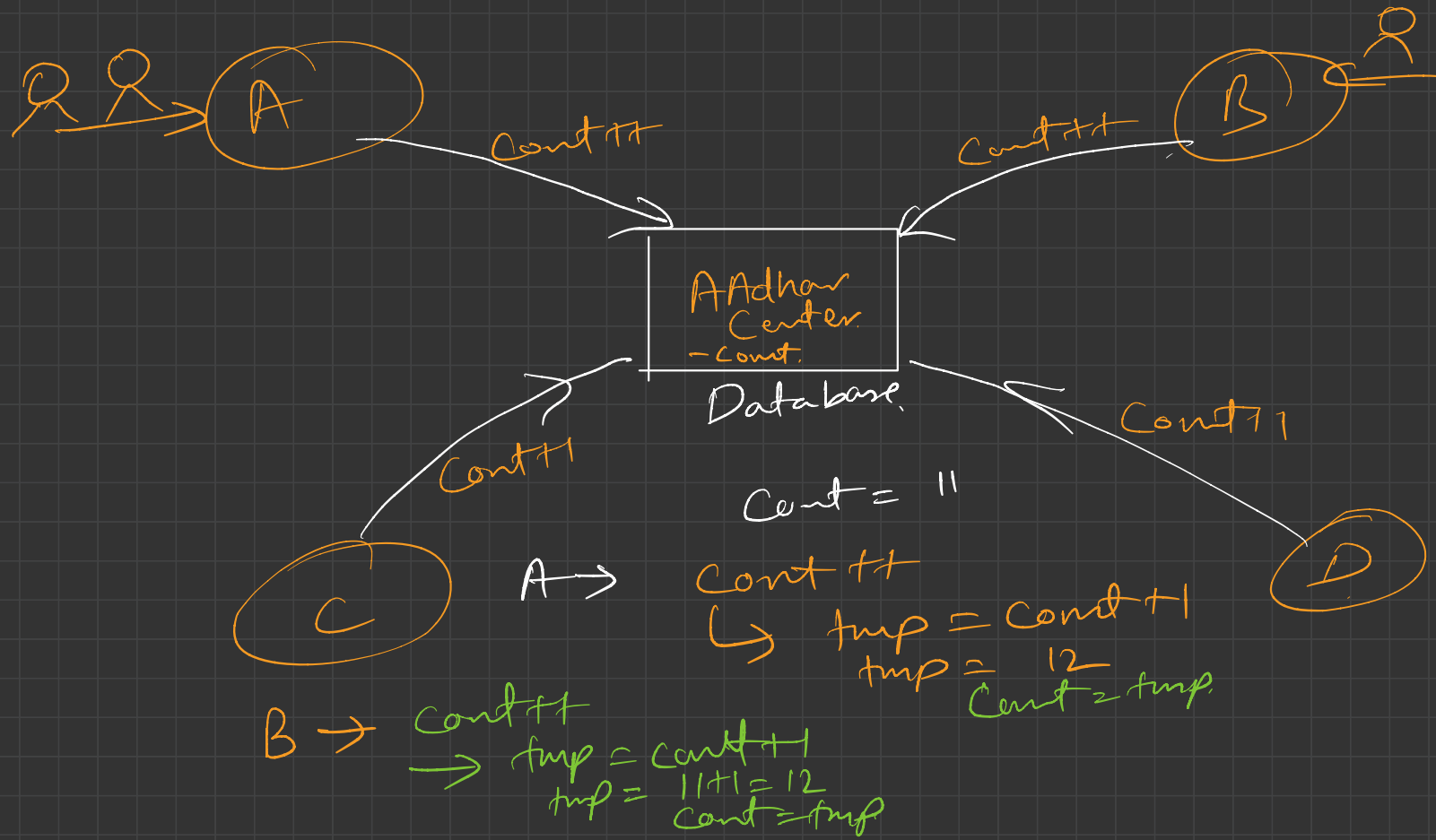

T1
T2
CPU1
CPU2

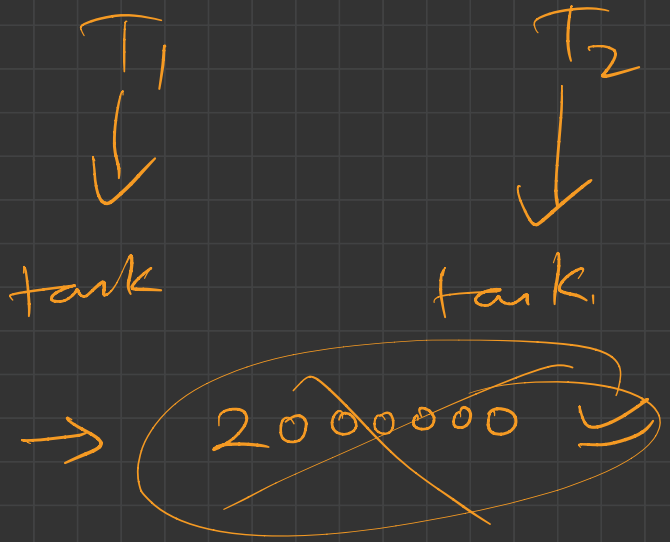
LEC-15: Introduction to Concurrency

1. **Concurrency** is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel.
2. **Thread**:
    - Single sequence stream within a process.
    - An independent path of execution in a process.
    - Light-weight process.
    - Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
    - E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell checking, formatting of text and saving the text are done concurrently by multiple threads.)
3. **Thread Scheduling:** Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.
4. **Threads context switching**
    - OS saves current state of thread & switches to another thread of same process.
    - Doesn't includes switching of memory address space. (But Program counter, registers & stack are included.)
    - Fast switching as compared to process switching
    - CPU's cache state is preserved.
5. **How each thread get access to the CPU?**
    - Each thread has its own program counter.
    - Depending upon the thread scheduling algorithm, OS schedule these threads.
    - OS will fetch instructions corresponding to PC of that thread and execute instruction.
6. **I/O or TQ, based context switching is done here as well**
    - We have TCB (Thread control block) like PCB for state storage management while performing context switching.
7. **Will single CPU system would gain by multi-threading technique?**
    - Never.
    - As two threads have to context switch for that single CPU.
    - This won't give any gain.
8. **Benefits of Multi-threading.**
    - Responsiveness
    - Resource sharing: Efficient resource sharing.
    - Economy: It is more economical to create and context switch threads.
        1. Also, allocating memory and resources for process creation is costly, so better to divide tasks into threads of same process.
    - Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Lec-16




A
B
C
D
Count++
Count++
Count++
Count++
AAdhar Center
- Count
Database
Count = 11
A → Count++
↳ tmp = Count+1
tmp = 12
Count = tmp
B → Count++
→ tmp = Count+1
tmp = 11+1 = 12
Count = tmp


T1
T2
tank
tank
o/p →
2000000
o/p → inconsistent

Critical section problem

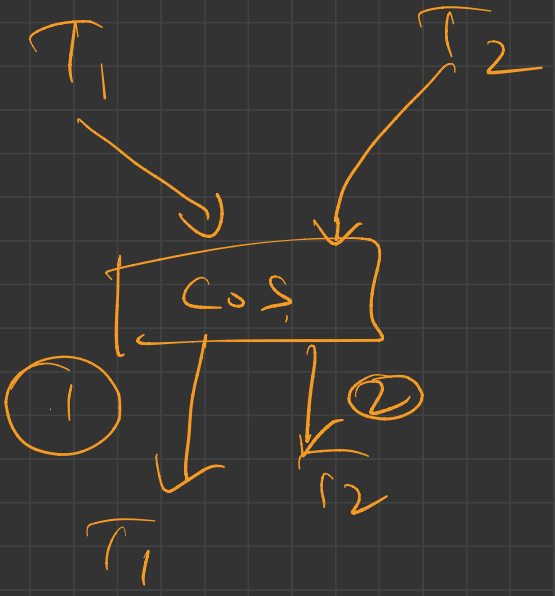
⇒ Solution of Race cond.

(1) Count++ ⇒ tmp = count + 1 ✓
count = tmp ✓

Atomic ⇒ ✓

C++ ⇒ atomic <int> ⇒

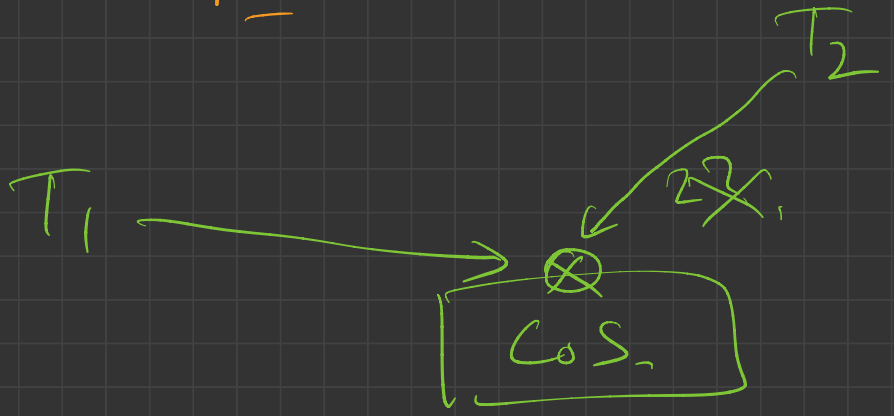
(2) Mutual Exclusion.



$T_1 \rightarrow T_2$
OR
$T_2 \rightarrow T_1$

Locks → Mutex

③ Semaphores



$T_1 \rightarrow T_2$

Locks ⇒ C.S ⇒ solve

Q → Can we use single flag ??

Q → Soln of C.S should have 3 conditions?

① Mutual exclusion.

② Progress. T_1 T_2

$T_1 \rightarrow T_2$ ✗

$T_2 \rightarrow T_1$ ✗

③ Bonded waiting :- – indefinite waiting ✗.

→ Limited waiting time.

Q single flag ⇒ turn = flag = 0/1

T_1

```
while (1)
{
    while (turn != 0);
        C.S
    turn = 1
        R.S
}
```

T_2

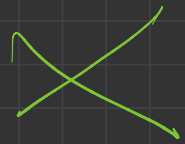
```
while (1)
{
    while (turn != 1);
        C.S
    Turn = 0
        RS
}
```

① ⇒ turn = 0 ⇒ T_1 ⇒ T_2

turn = 1 ⇒ T_2 ⇒ T_1

② Progress — fixed order.

=



→ single flag X

↓

improvement

→ Petersons solution

* Peterson's solution — flag[2] →
— turn

flag[2] → indicate if a thread is ready to enter the C.S, flag[i] = true implies that P_i is ready

turn → indicates whose turn is to enter the C.S.

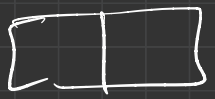
↳ 0/1

T_1

```
while(1)
{
    flag[0] = T
    turn = 1
    while (turn == 1 && flag[1] == T);
    C.S
    flag[0] = F
}
```

T_2

```
while (1)
{
    flag[1] = T
    turn = 0
    while (turn == 0 && flag[0] == T);
    C.S
    flag[1] = F
}
```

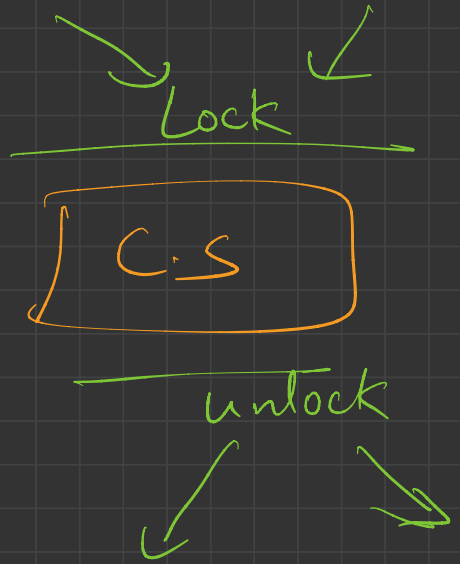
flag 

turn :

Peterson's Algo.

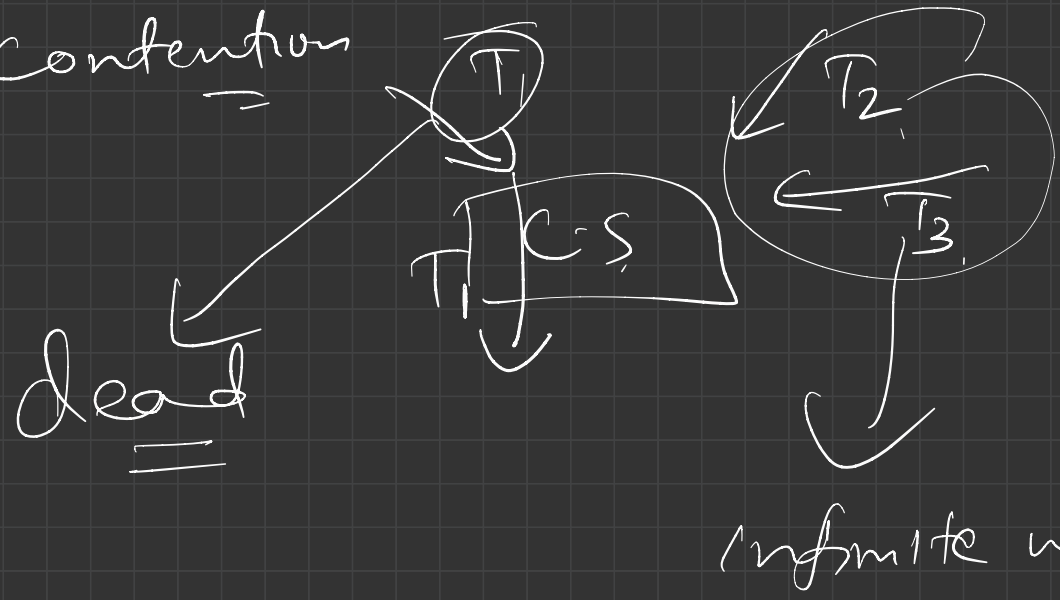
→ Peterson's sol^{n} → 2 threads safe.

→ Locks →

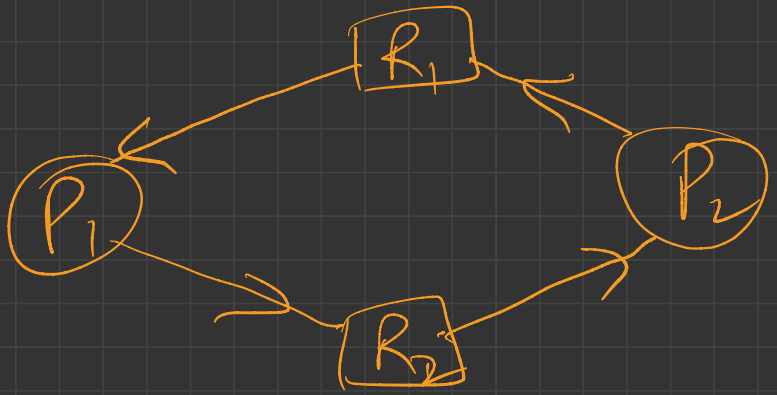


Locks disadvantages

① contention

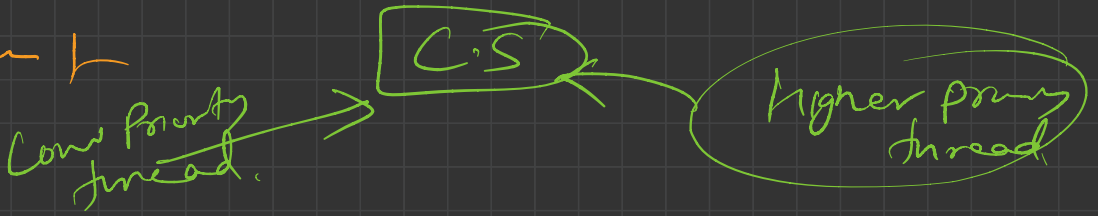


② Deadlock.



③ Debugging issue

④ Starvation



LEC-16: Critical Section Problem and How to address it

1. Process synchronization techniques play a key role in maintaining the consistency of shared data
2. **Critical Section (C.S)**
    a. The critical section refers to the segment of code where processes/threads access shared resources, such as common variables and files, and perform write operations on them. Since processes/threads execute concurrently, any process can be interrupted mid-execution.
3. **Major Thread scheduling issue**
    a. Race Condition
        i. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e., both threads are "racing" to access/change the data.
4. **Solution to Race Condition**
    a. Atomic operations: Make Critical code section an atomic operation, i.e., Executed in one CPU cycle.
    b. Mutual Exclusion using locks.
    c. Semaphores
5. Can we use a simple flag variable to solve the problem of race condition?
    a. No.
6. **Peterson's solution** can be used to avoid race condition but holds good for only 2 process/threads.
7. **Mutex/Locks**
    a. Locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread/process to access critical section.
    b. Disadvantages:
        i. Contention: one thread has acquired the lock, other threads will be busy waiting, what if thread that had acquired the lock dies, then all other threads will be in infinite waiting.
        ii. Deadlocks
        iii. Debugging
        iv. Starvation of high priority threads.

Lec-17

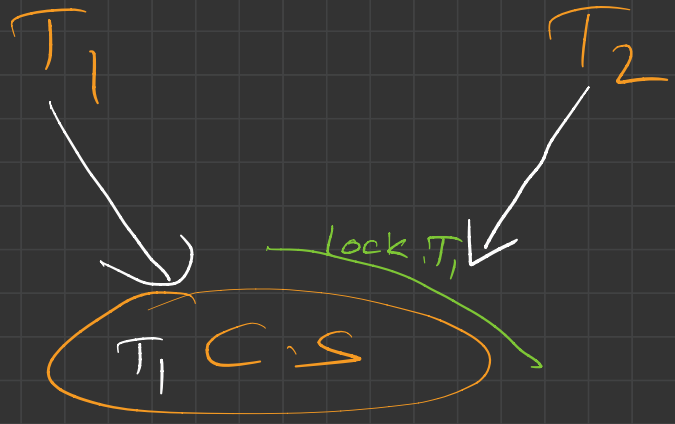
① — single flag ✗

② — Peterson's soln 2 threads

③ — Locks / Mutex

↓

Problems ⟹ Busy waiting



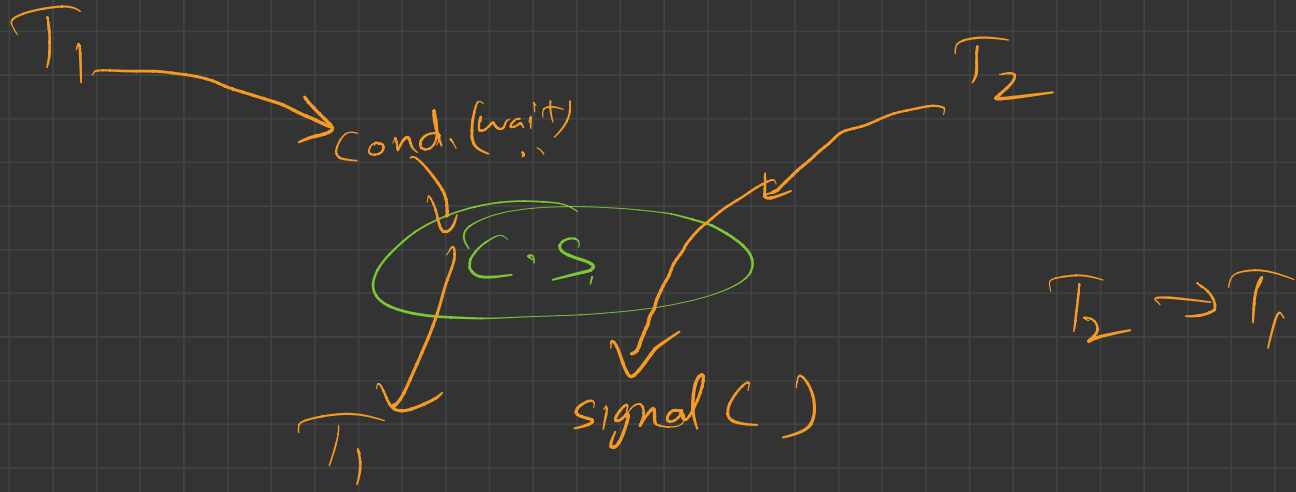
while (flag);

* Conditional Variables :-

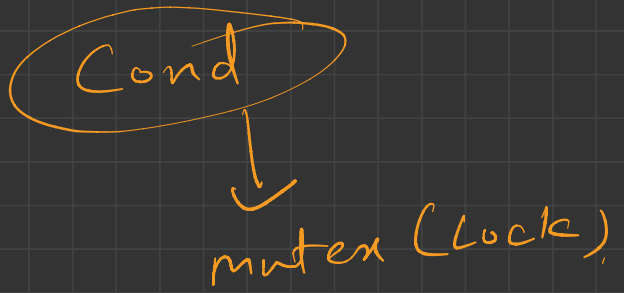
var => cond (lock)

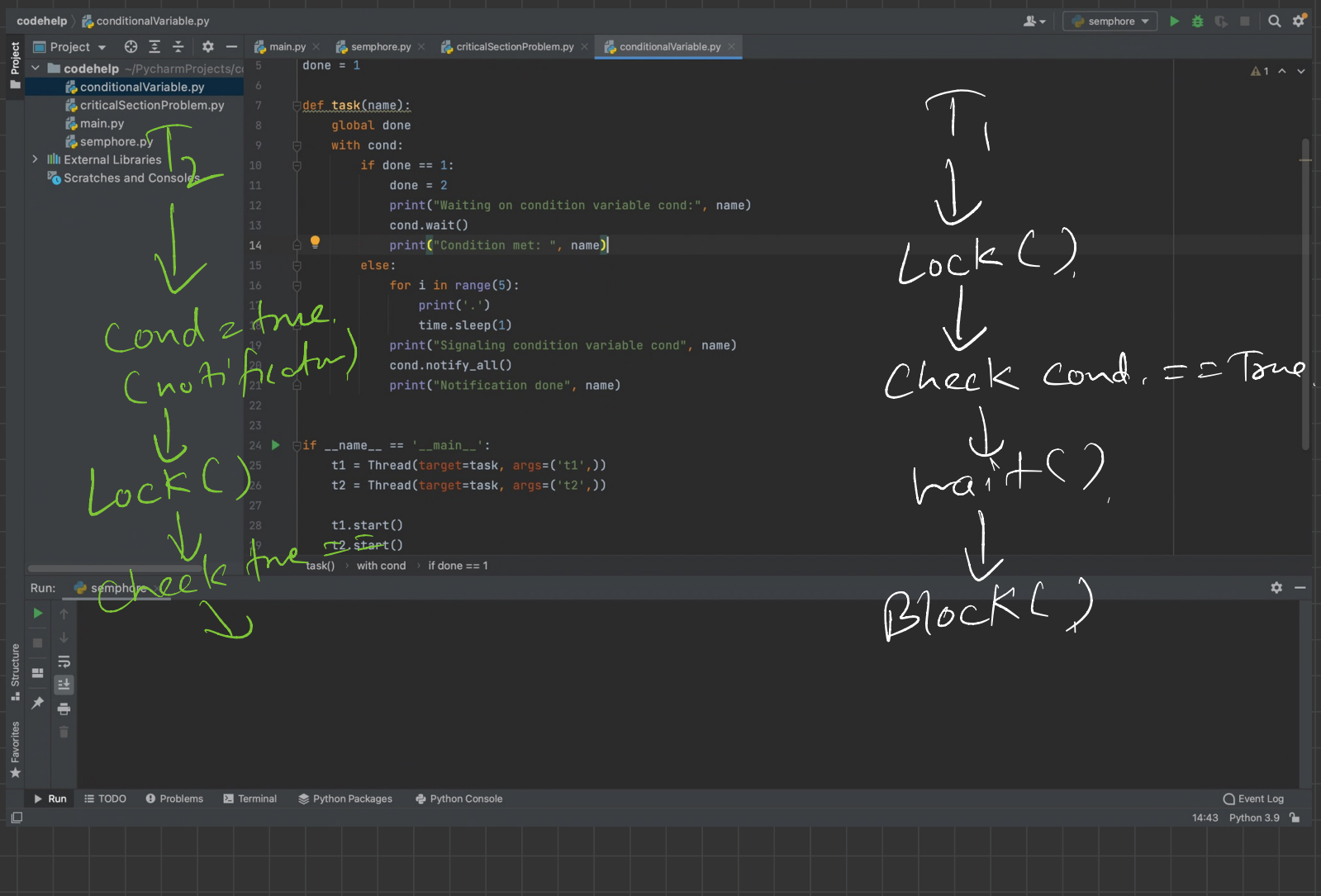
wait

signal



$T_2 \rightarrow T_1$

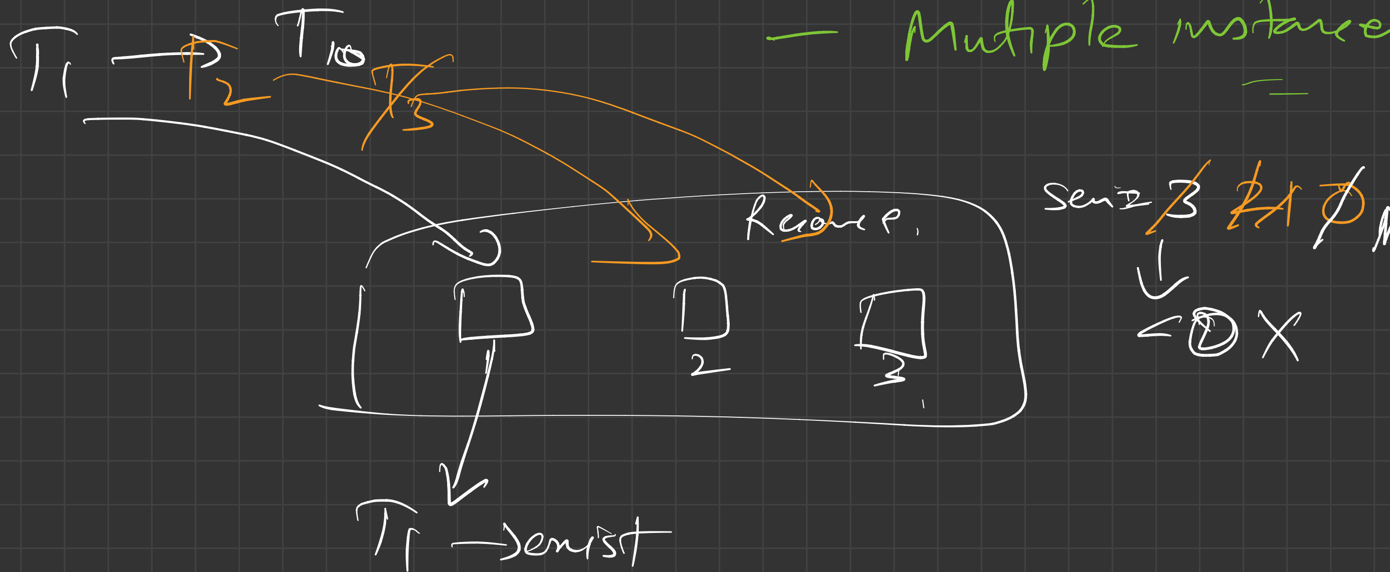

Cond
mutex (lock)


codehelp
conditionalVariable.py
semphore
Project
codehelp ~/PycharmProjects/c
conditionalVariable.py
criticalSectionProblem.py
main.py
semphore.py
External Libraries
Scratches and Consoles
main.py
semphore.py
criticalSectionProblem.py
conditionalVariable.py
done = 1
def task(name):
global done
with cond:
if done == 1:
done = 2
print("Waiting on condition variable cond:", name)
cond.wait()
print("Condition met: ", name)
else:
for i in range(5):
print('.')
time.sleep(1)
print("Signaling condition variable cond", name)
cond.notify_all()
print("Notification done", name)
if __name__ == '__main__':
t1 = Thread(target=task, args=('t1',))
t2 = Thread(target=task, args=('t2',))
t1.start()
t2.start()
task() › with cond › if done == 1
Run:
semphore
Run
TODO
Problems
Terminal
Python Packages
Python Console
Event Log
14:43
Python 3.9
T2
Cond = true
(notification)
Lock()
Check
T1
Lock()
Check Cond. == True
wait()
Block()

Semaphores → integer

Resource → Single X

— Multiple instances



wait()
C.S
signd.

```
wait(S)
{
    S->value--;
    if(S->value<0)
    {
        add to S->blockList
        Block();
    }
}
```

Semaphore S(2);

```
Signal(S)
{
    S->value++
    if(S->value<=0)
    {
        remove P from S->BlakList
        wakeup(P)
    }
}
```

T_1 → wait() → S->val=1 → After C.S → signd

T_2 → wait() → S->val=0 →

T_3 → wait() → S->val → -1 → Block()

CPU.

Semaphore → n → 1 → Binary semaphore.

Semaphore → >1 → Counting semaphore

LEC-17: Conditional Variable and Semaphores for Threads synchronization

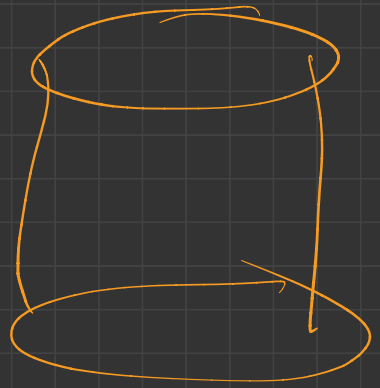
1. **Conditional variable**
    a. The condition variable is a synchronization primitive that lets the thread wait until a certain condition occurs.
    b. Works with a lock
    c. Thread can enter a wait state only when it has acquired a lock. When a thread enters the wait state, it will release the lock and wait until another thread notifies that the event has occurred. Once the waiting thread enters the running state, it again acquires the lock immediately and starts executing.
    d. Why to use conditional variable?
        i. To avoid busy waiting.
    e. **Contention** is not here.
2. **Semaphores**
    a. Synchronization method.
    b. An integer that is equal to number of resources
    c. Multiple threads can go and execute C.S concurrently.
    d. Allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource one at a time.
    e. Binary semaphore: value can be 0 or 1.
        i. Aka, mutex locks
    f. Counting semaphore
        i. Can range over an unrestricted domain.
        ii. Can be used to control access to a given resource consisting of a finite number of instances.
    g. To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process car block itself. The block- operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the Waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
    h. A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal () operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

Lec-19

Reader writer problem

(1) Reader thread → Read

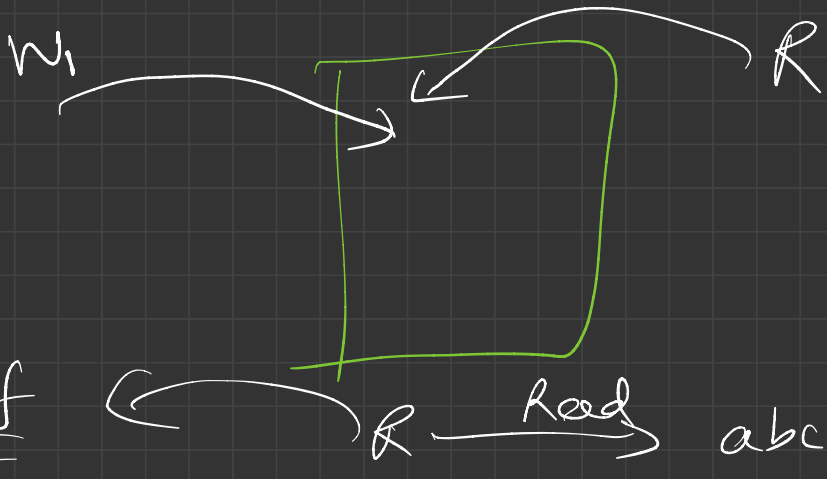
(2) writer threads → write update



(1) if >1 Readers are reading → No issue ✗

② >1 writers OR 1 writer & some other thread (R/W), parallely,

→ Race condition & data inconsistent



Semaphors soln to R-W problem

(1) mutex → Binary semaphore
- to ensure mutual exclusion, when read count (rc) is updated.
- No. two threads modify R.C at same time.

(2) wrt → Binary semaphore,
- common for both reader, writer.

(3) read count (rc) → integer [0],
tracks how many readers are reading in C.S

Writer Soln

```
do {
    wait (wrt);
    // do write operation.
    signal (wrt);
} while ( true );
```

Reader soln

```
do{
    wait (mutex) ; // to mutex readcount variable
     rc++;
     if (rc==1)
        wait (wrt) ; // ensures no writer can enter if there is
                     // even one reader

      signal(mutex) ;

// C.S : Reader is reading

     wait (mutex)
     rc--; // a reader leaves

     if( rc==0) // no reader is left in CS
        signal(wrt) ; // writer can enter
      signal(mutex) ; // reader leaves
} while (1)
```

Soln:

→ implement ✓

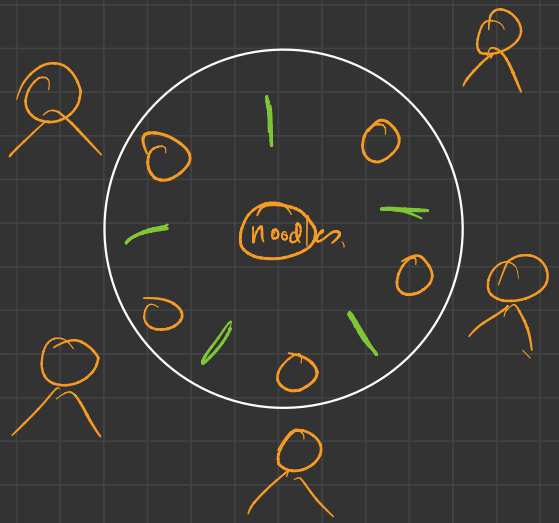
Lec-20

Dining philosophers

(A) 5 - ph.

(B) noodles

(C) 5 forks



Semaphores.

① each fork — semaphore (Binary sema)

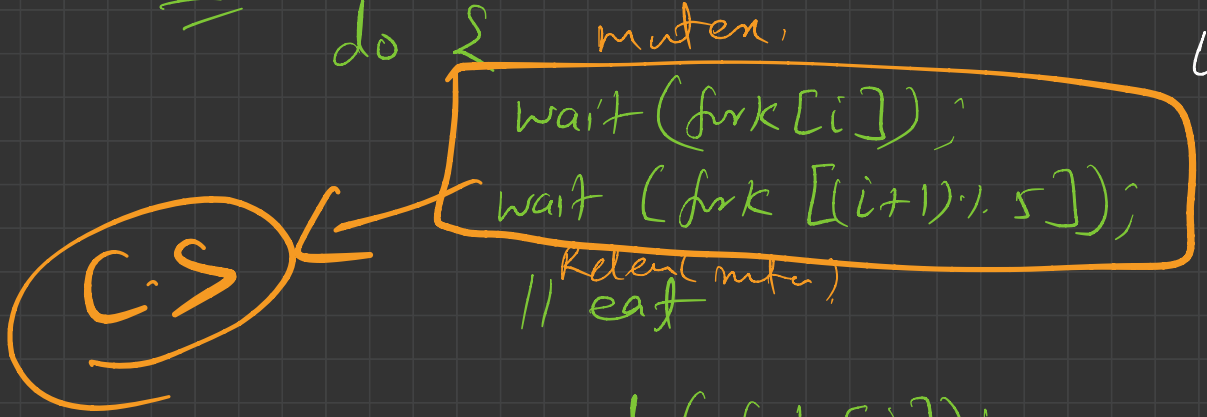
Semaphore fork[5] {1}

② wait() → fork[i] ⇒ ph[i] → acquire.

③ Release() → fork[i] → fork → Release.

Soln

Ph [i]

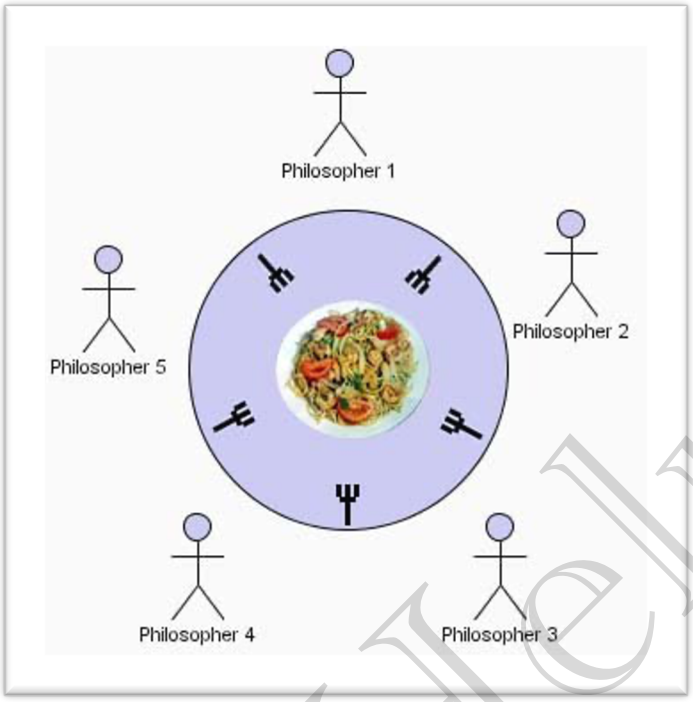


i = 1

(1+1) % 5

→ 2

Lec-20: The Dining Philosophers problem



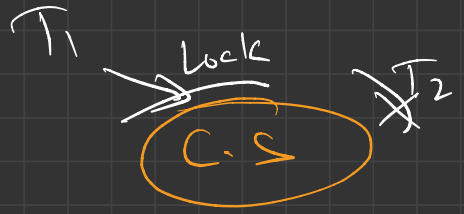
1. We have **5 philosophers.**
2. They spend their life just being in **two states:**
    a. Thinking
    b. Eating
3. They sit on a circular table surrounded by 5 chairs (1 each), in the center of table is a bowl of noodles, and the table is laid with 5 single forks.
4. **Thinking state**: When a ph. Thinks, he doesn't interact with others.
5. **Eating state:** When a ph. Gets hungry, he tries to pick up the 2 forks adjacent to him (Left and Right). He can pick one fork at a time.
6. One can't pick up a fork if it is already taken.
7. When ph. Has both forks at the same time, he eats without releasing forks.
8. Solution can be given using semaphores.
    a. Each fork is a binary semaphore.
    b. A ph. Calls wait() operation to acquire a fork.
    c. Release fork by calling signal().
    d. **Semaphore fork[5]{1};**
9. Although the semaphore solution makes sure that no two neighbors are eating simultaneously but it could still create **Deadlock**.
10. Suppose that all 5 ph. Become hungry at the same time and each picks up their left fork, then All fork semaphores would be 0.
11. When each ph. Tries to grab his right fork, he will be waiting for ever (Deadlock)
12. We must use **some methods to avoid Deadlock and make the solution work**
    a. Allow at most 4 ph. To be sitting simultaneously.
    b. Allow a ph. To pick up his fork only if both forks are available and to do this, he must pick them up in a critical section (atomically).

c. **Odd-even rule.**
an odd ph. Picks up first his left fork and then his right fork, whereas an even ph. Picks up his right fork then his left fork.

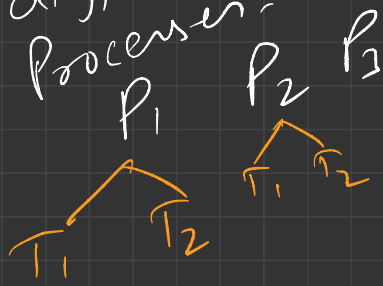
13. Hence, only semaphores are not enough to solve this problem.
We must add some enhancement rules to make deadlock free solution.

Lec-21

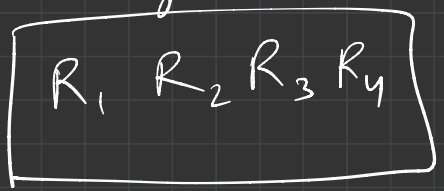
Deadlock



different Processes.



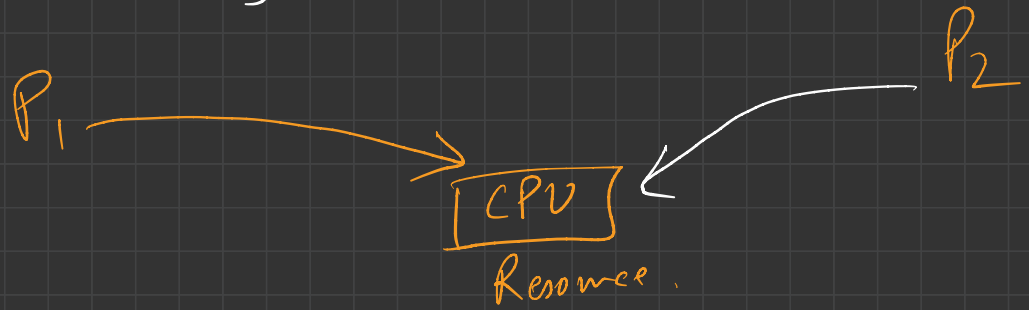
system.



⇒ Finite no. of Resource.

⇒ Multiple Process

→ Memory space, CPU, files, locks, I/O devices etc.



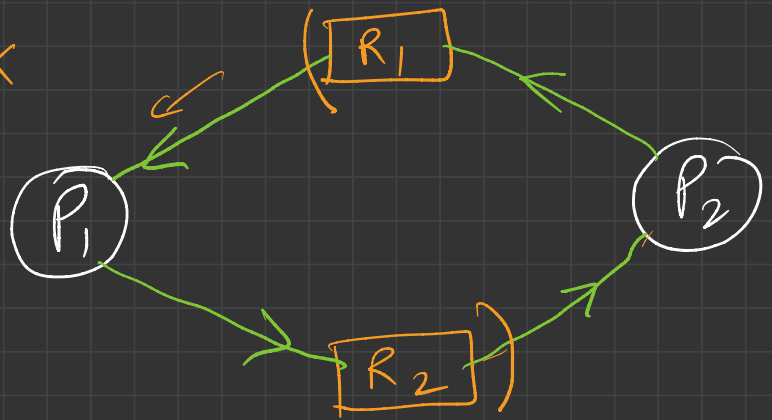
System.

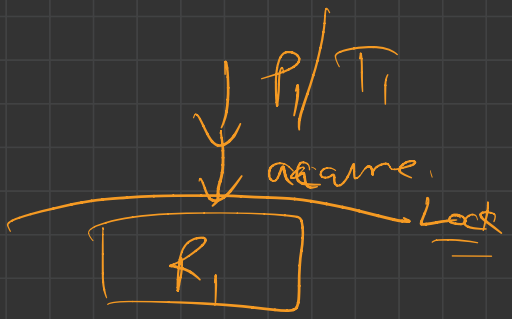
↓

Deadlock

P_1 → Program X

P_2 → Program X.



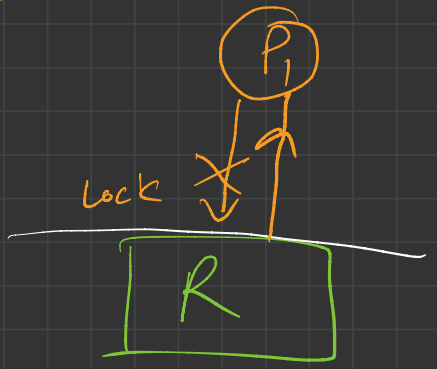


$P_1 \rightarrow R_1 \& R_2 \Rightarrow$

$P_2 \Rightarrow R_1 \& R_2 \Rightarrow$

How a Process/Thread utilize a R ?

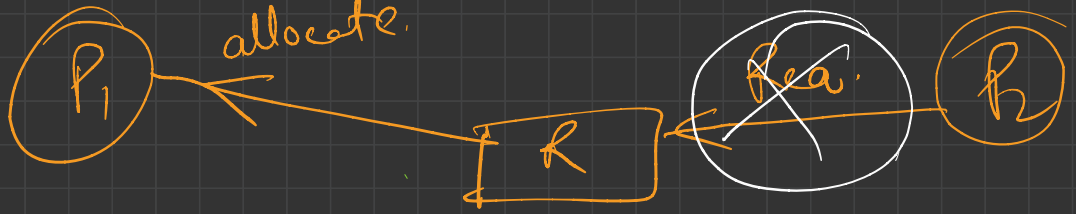
1. Request
2. Use
3. Release.



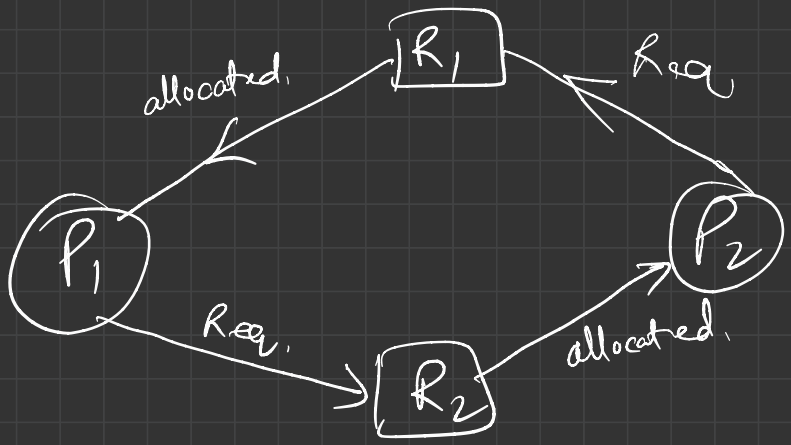
Necessary conditions for DL (4)

(1) Mutual exclusion :-

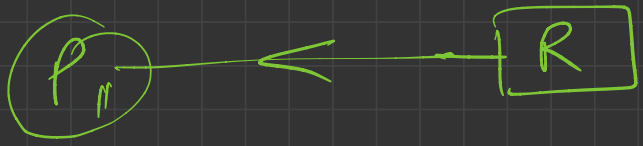
1 R → 1 Process / thread.



② Hold & wait

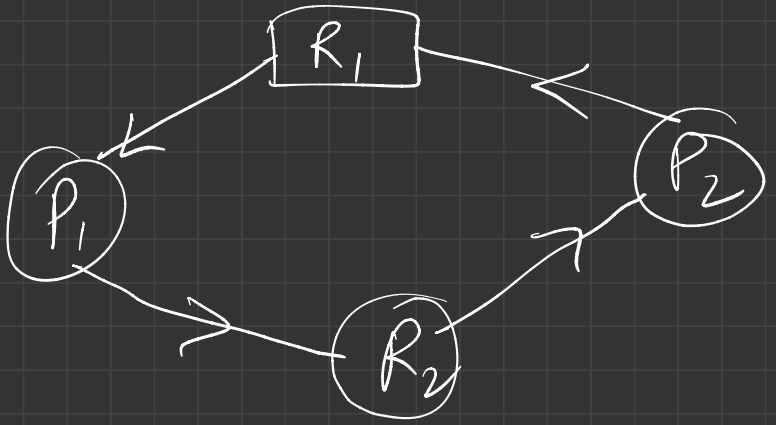


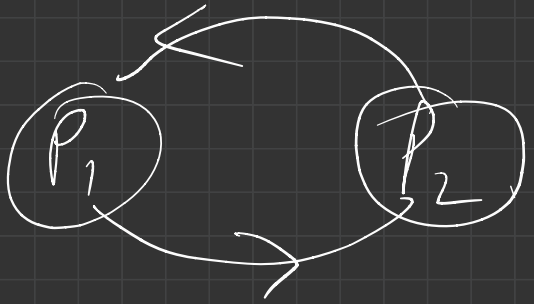
(3) No preemption :-



P_I → execution complete.

(4) Circular wait.





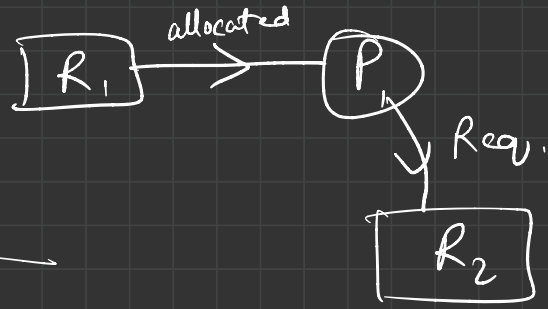
* Resource allocation graph (RAG)

① Vertex → ① Process vertex (P)

② Resource vertex (R)

② Edges → ① Assign

② Request



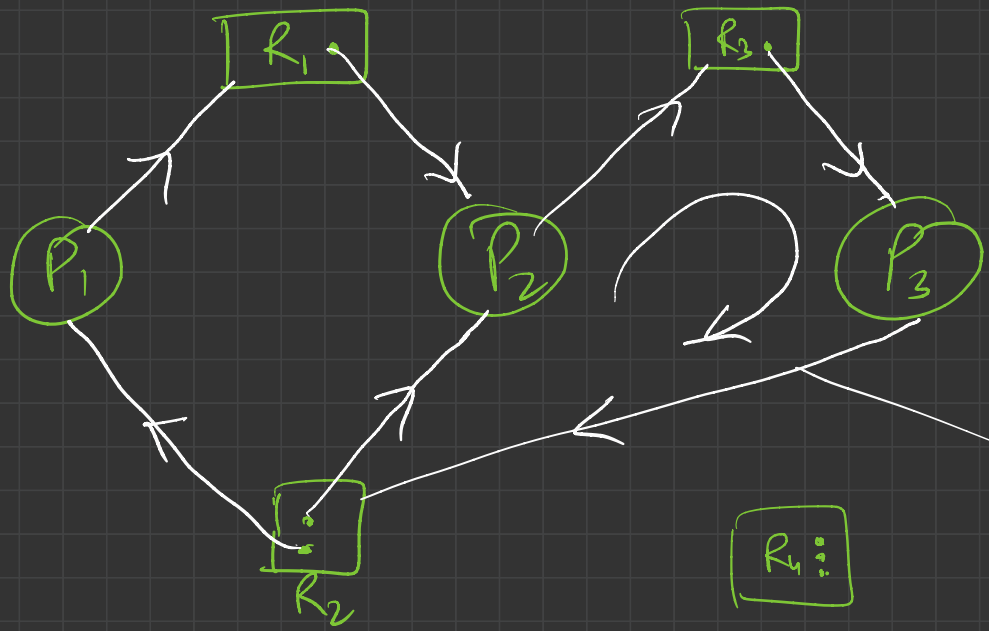
Multiple instance → R (...)

CPU → 4 (CPU)

R

→ System Representation

(1)



By definition of RAG

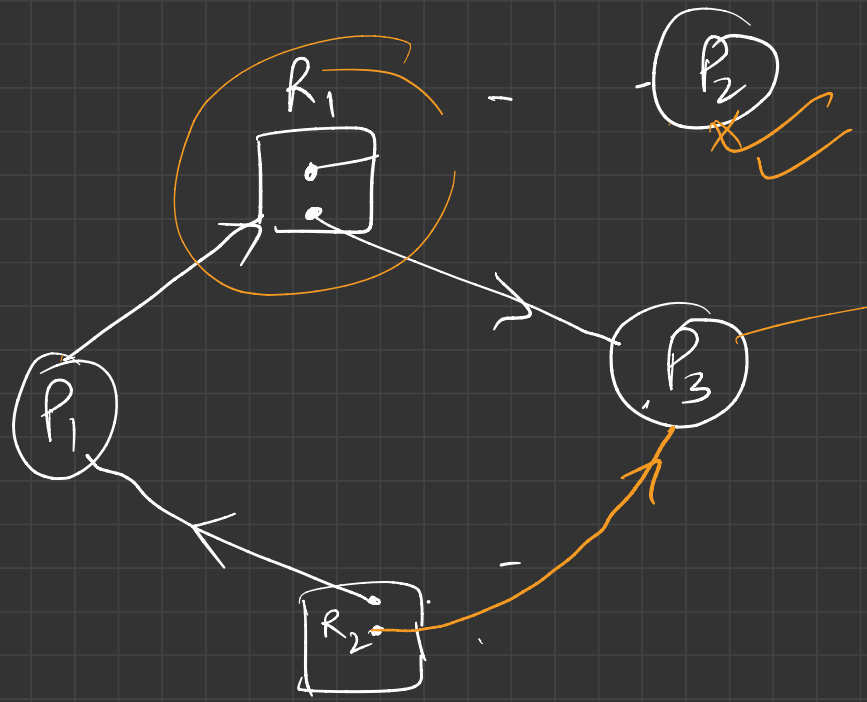
→

① RAG No cycle → No DL

② RAG cycle → may be DL.

DL is present.

(2)



execution succes

t2 0

P4 → R2 → Release

* Methods to handle DL

1. Prevent or avoid DL
2. allow system to go in DL
   - Detect → recover.
3. Ostrich algorithm :- (DL ignorance)
   App. programmers.

* DL Prevention :-

① Mutual exclusion :-

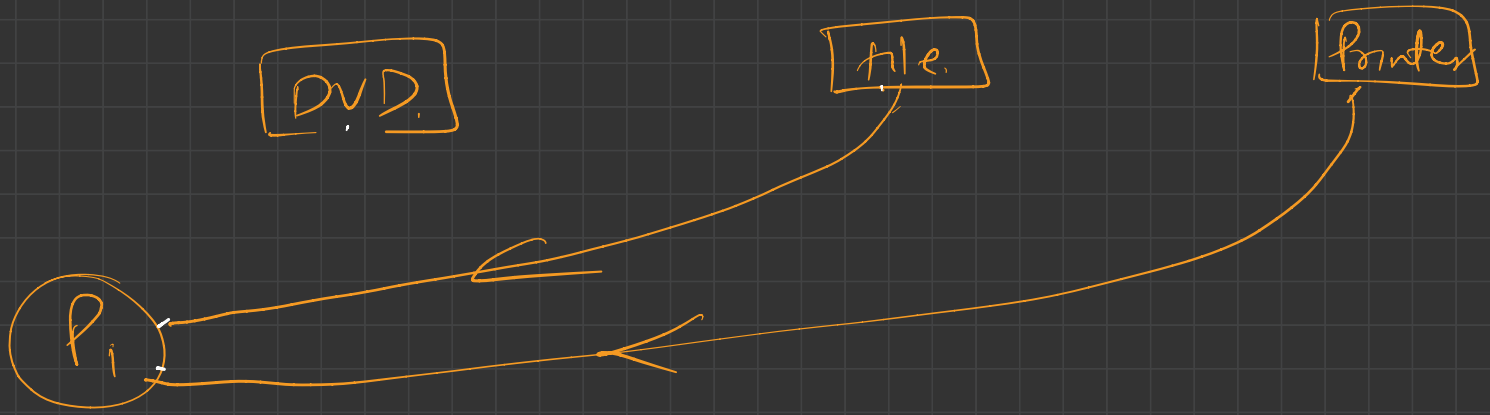
→ non-sharable resources. → mutex.

CS → Memory space

② Read-only file → sharable resource

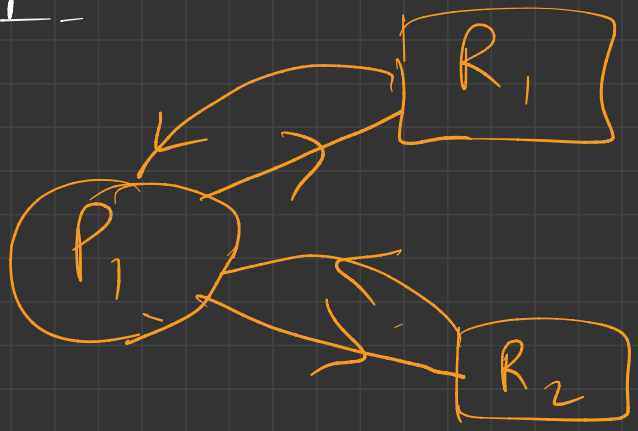
Lock X

(2) Hold & wait :-



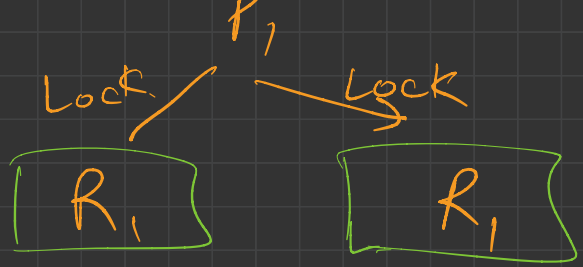
(1) Copy
(2) Sort
(3) Print ⟶ Printer Req.

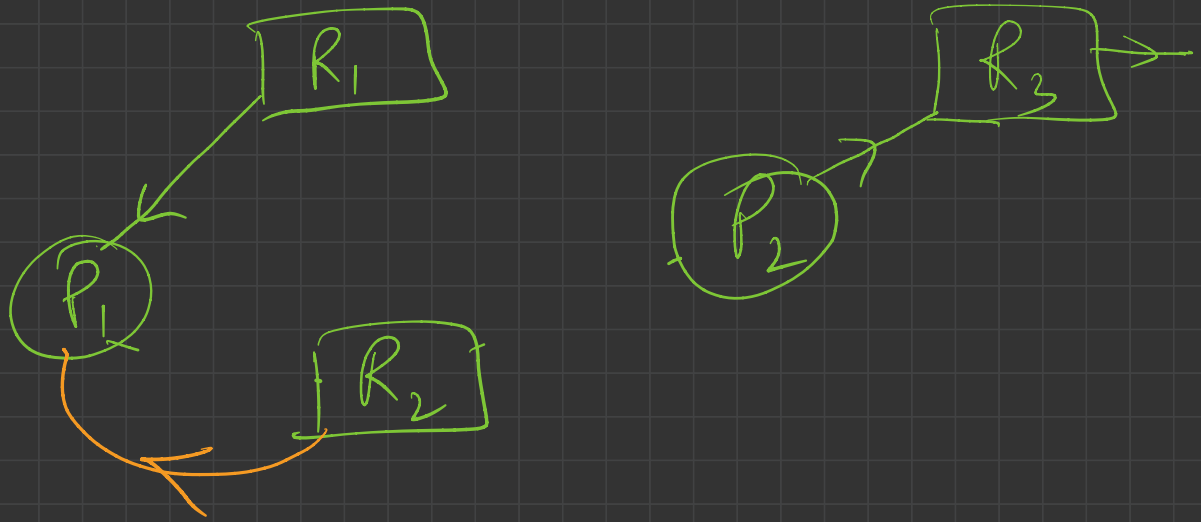
③ No - preemption :-



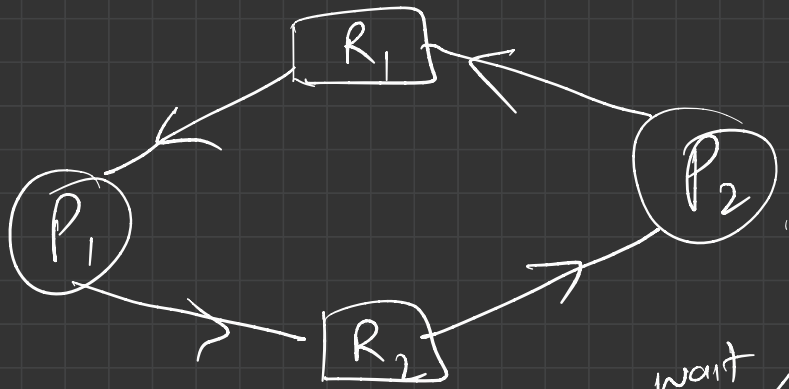
$P_1 \leftarrow R_1$

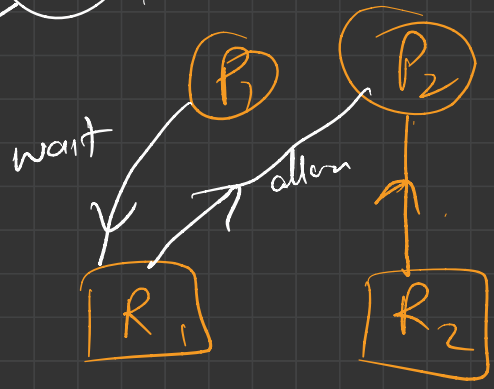
Give Lock




R1
P1
R2
R3
P2

④ Circular wait





Soln ⇒ $P_1 \rightarrow R_1 \& R_2$

$P_2 \rightarrow R_1 \& R$

Order ⇒ $R_1 \xrightarrow{R_1 \text{ already}} R_2$

fixed order

Lec -22

* Deadlock avoidance :-

① Current state → ① no. of processes
② max need of R. of each proc.
③ Currently allocated amount of R. to each Proc.
④ Max. amount of each resource.

schedule R(s) R(s)

↳ schedule.
Process & Resources.

→ System safe state

safe state

P_1 P_2 P_3 P_4 →

$P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2$

P	allocated A	allocated B	allocated C	max. need A	max. need B	max. need C	available A	available B	available C	Remaining need A	Remaining need B	Remaining need C	
P_1	0	1	0	7	5	3	3	3	2	7	4	3	P_1
P_2	2	0	0	3	2	2	5	3	2	1	2	2	P_2
P_3	3	0	2	9	0	2	7	4	3	6	0	0	P_3
P_4	2	1	1	4	2	2	7	4	5	2	1	1	P_4
P_5	0	0	2	5	3	3	7	5	5	5	3	1	P_5
Total already:	7	2	5				10	5	7				

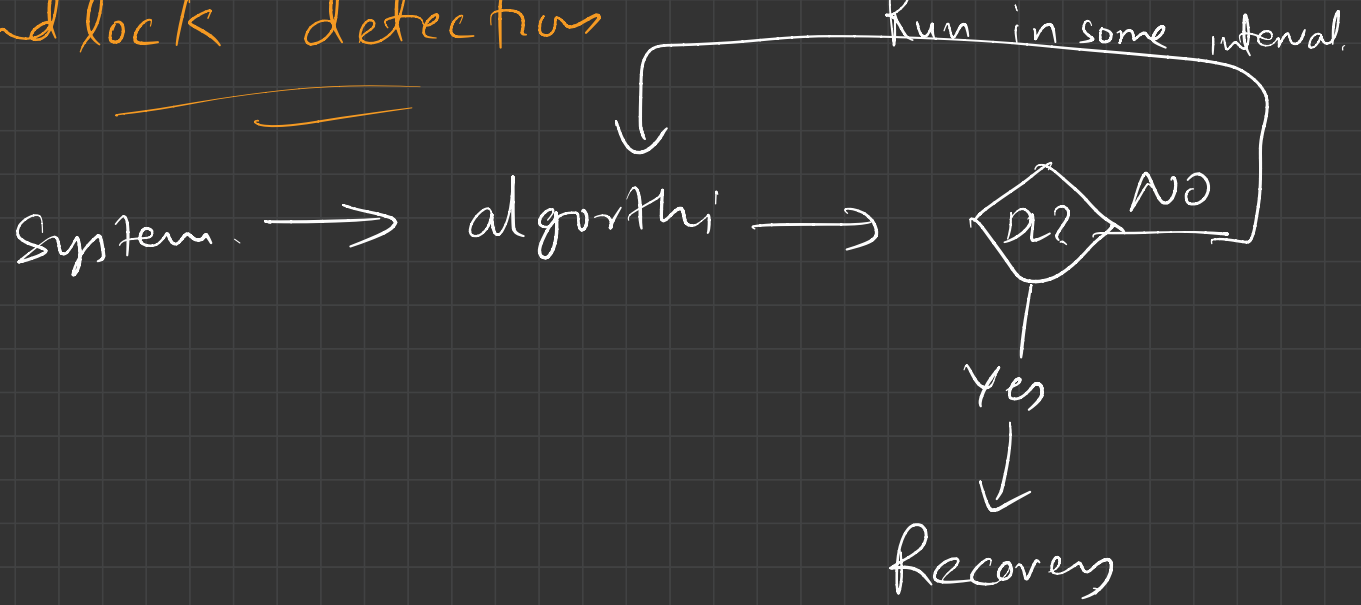
Total ⇒ A = 10, B = 5, C = 7

$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

↳ safe state

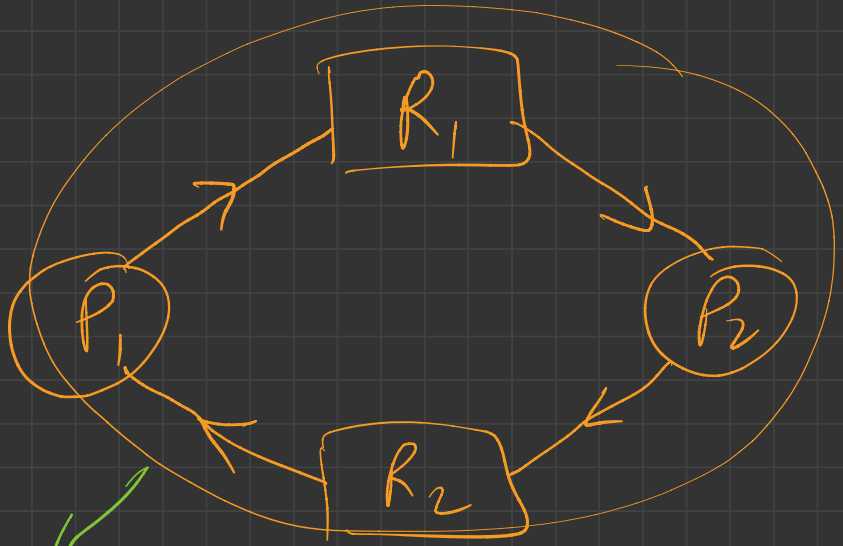
unsafe state

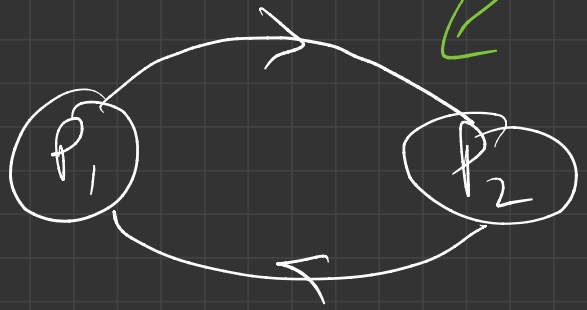
* Deadlock detection



① Single Instance of each resource

* wait-for graph





Cycle ⇒ DL ✓

No cycle ⇒ DL X.

② Multiple Instances.

— Banker's algorithm.

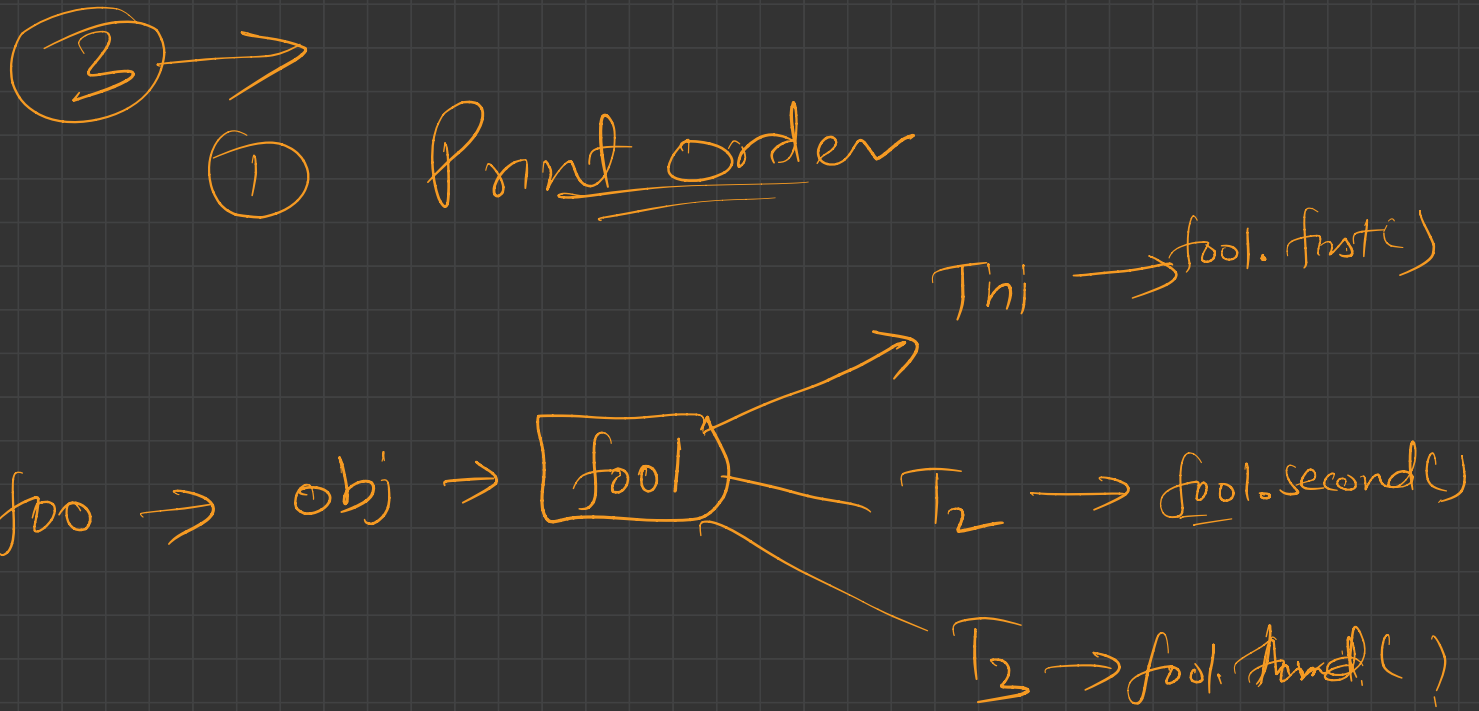
→ Safe sequence available → No DL.

→ No safe seq. available → DL detected.

LEC-22: Deadlock Part-2

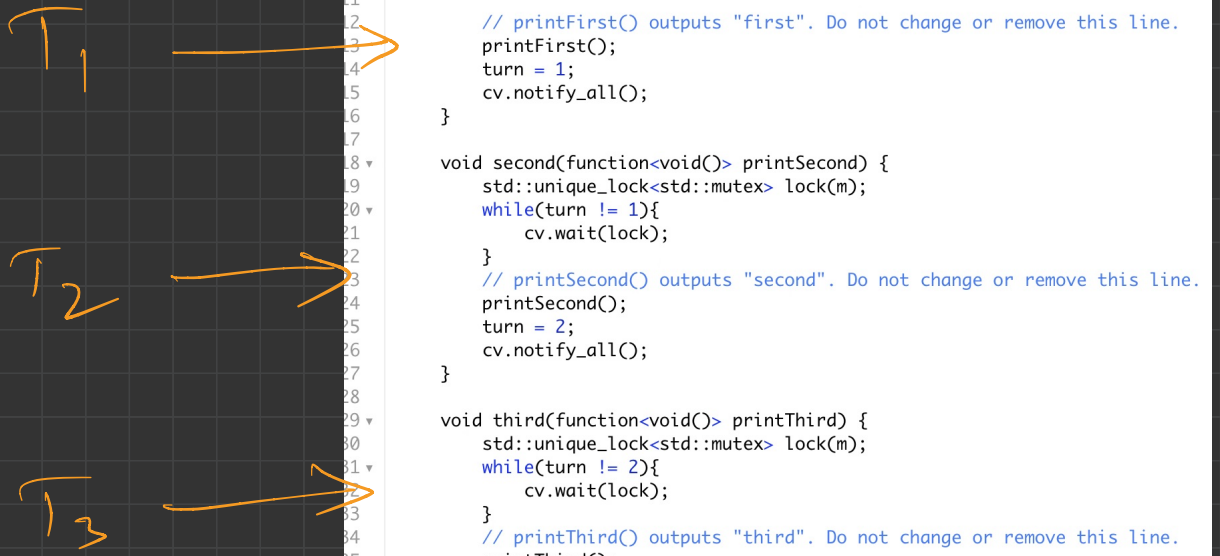
1. **Deadlock Avoidance**: Idea is, the kernel be given in advance info concerning which resources will use in its lifetime.
   By this, system can decide for each request whether the process should wait.
   To decide whether the current request can be satisfied or delayed, the system must consider the resources currently available, resources currently allocated to each process in the system and the future requests and releases of each process.
   a. Schedule process and its resources allocation in such a way that the DL never occur.
   b. Safe state: A state is safe if the system can allocate resources to each process (up to its max.) in some order and still avoid DL.
      A system is in safe state only if there exists a safe sequence.
   c. In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.
   d. The main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the resulting state is a safe state.
   e. In a case, if the system is unable to fulfill the request of all processes, then the state of the system is called unsafe.
   f. Scheduling algorithm using which DL can be avoided by finding safe state. (Banker Algorithm)
2. **Banker Algorithm**
   a. When a process requests a set of resources, the system must determine whether allocating these resources will leave the system in a safe state. If yes, then the resources may be allocated to the process. If not, then the process must wait till other processes release enough resources.
3. **Deadlock Detection:** Systems haven't implemented deadlock-prevention or a deadlock avoidance technique, then they may employ DL detection then, recovery technique.
   a. Single Instance of Each resource type **(wait-for graph method)**
      i. A deadlock exists in the system if and only if there is a cycle in the wait-for graph. In order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the wait-for graph.
   b. Multiple instances for each resource type
      i. Banker Algorithm
4. **Recovery from Deadlock**
   a. Process termination
      i. Abort all DL processes
      ii. Abort one process at a time until DL cycle is eliminated.
   b. Resource preemption
      i. To eliminate DL, we successively preempt some resources from processes and give these resources to other processes until DL cycle is broken.

Lec-23


3
1
Print order
foo
obj
fool
Thi
fool.first()
T2
fool.second()
T3
fool.third()

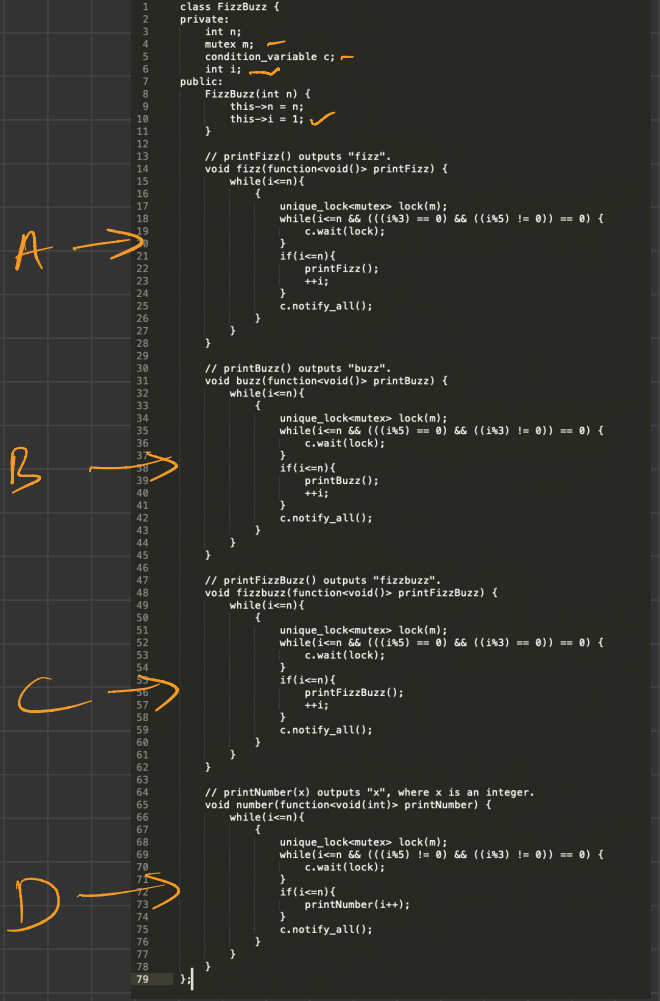
locks & conditional variable

1. first → printfirst() → T_1 execute.
2. when first is printed → T_2 execute.




```cpp
class Foo {
    std::mutex m;
    std::condition_variable cv;
    int turn;
public:
    Foo() {
        turn = 0;
    }

    void first(function<void()> printFirst) {

        // printFirst() outputs "first". Do not change or remove this line.
        printFirst();
        turn = 1;
        cv.notify_all();
    }

    void second(function<void()> printSecond) {
        std::unique_lock<std::mutex> lock(m);
        while(turn != 1){
            cv.wait(lock);
        }
        // printSecond() outputs "second". Do not change or remove this line.
        printSecond();
        turn = 2;
        cv.notify_all();
    }

    void third(function<void()> printThird) {
        std::unique_lock<std::mutex> lock(m);
        while(turn != 2){
            cv.wait(lock);
        }
        // printThird() outputs "third". Do not change or remove this line.
        printThird();
    }
};
```

(2) → Fizz buzz problem



```
class FizzBuzz {
private:
    int n;
    mutex m;
    condition_variable c;
    int i;
public:
    FizzBuzz(int n) {
        this->n = n;
        this->i = 1;
    }

    // printFizz() outputs "fizz".
    void fizz(function<void()> printFizz) {
        while(i<=n){
            {
                unique_lock<mutex> lock(m);
                while(i<=n && (((i%3) == 0) && ((i%5) != 0)) == 0) {
                    c.wait(lock);
                }
                if(i<=n){
                    printFizz();
                    ++i;
                }
                c.notify_all();
            }
        }
    }

    // printBuzz() outputs "buzz".
    void buzz(function<void()> printBuzz) {
        while(i<=n){
            {
                unique_lock<mutex> lock(m);
                while(i<=n && (((i%5) == 0) && ((i%3) != 0)) == 0) {
                    c.wait(lock);
                }
                if(i<=n){
                    printBuzz();
                    ++i;
                }
                c.notify_all();
            }
        }
    }

    // printFizzBuzz() outputs "fizzbuzz".
    void fizzbuzz(function<void()> printFizzBuzz) {
        while(i<=n){
            {
                unique_lock<mutex> lock(m);
                while(i<=n && (((i%5) == 0) && ((i%3) == 0)) == 0) {
                    c.wait(lock);
                }
                if(i<=n){
                    printFizzBuzz();
                    ++i;
                }
                c.notify_all();
            }
        }
    }

    // printNumber(x) outputs "x", where x is an integer.
    void number(function<void(int)> printNumber) {
        while(i<=n){
            {
                unique_lock<mutex> lock(m);
                while(i<=n && (((i%5) != 0) && ((i%3) != 0)) == 0) {
                    c.wait(lock);
                }
                if(i<=n){
                    printNumber(i++);
                }
                c.notify_all();
            }
        }
    }
};
```

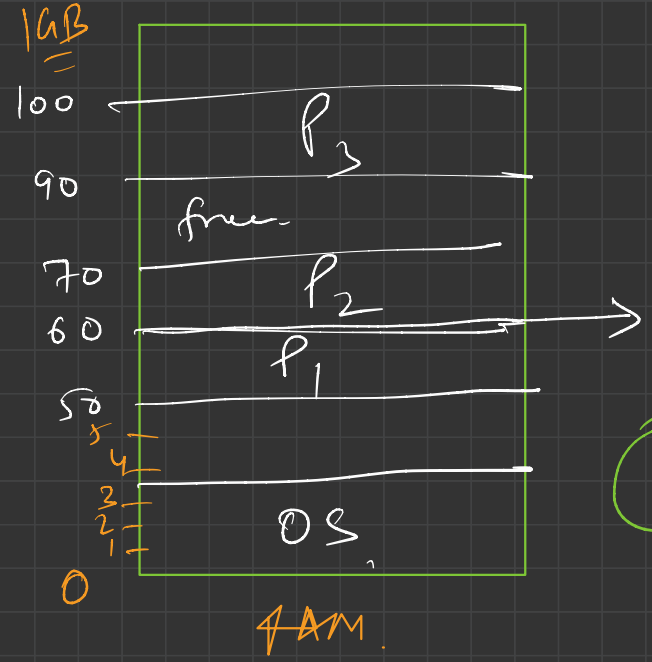
③ Dining philosopher's problem

video no. 20

```cpp
class Semaphore
{
public:
    Semaphore() {}
    Semaphore(int c) : count(c){};

    void setCount(int a)
    {
        count = a;
    }

    inline void signal()
    {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        if(count <= 0)
            cv.notify_one();
    }

    inline void wait()
    {
        std::unique_lock<std::mutex> lock(mtx);
        count--;
        while (count < 0)
        {
            cv.wait(lock);
        }
    }

private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};

class DiningPhilosophers {
    Semaphore fork[5];
    std::mutex m, l;
public:
    DiningPhilosophers() {
        for (int i = 0; i < 5; ++i) {
            fork[i].setCount(1);
        }
    }

    void wantsToEat(int philosopher,
                    function<void()> pickLeftFork,
                    function<void()> pickRightFork,
                    function<void()> eat,
                    function<void()> putLeftFork,
                    function<void()> putRightFork) {
        {
            std::lock_guard<std::mutex>lock(m);
            fork[(philosopher + 1) % 5].wait();
            fork[philosopher].wait();
        }
        pickLeftFork();
        pickRightFork();

        eat();

        putLeftFork();
        fork[(philosopher + 1) % 5].signal();
        putRightFork();
        fork[philosopher].signal();
    }
};
```

Lec-24

⇒ Multi-programming → many procen → RAM



P1

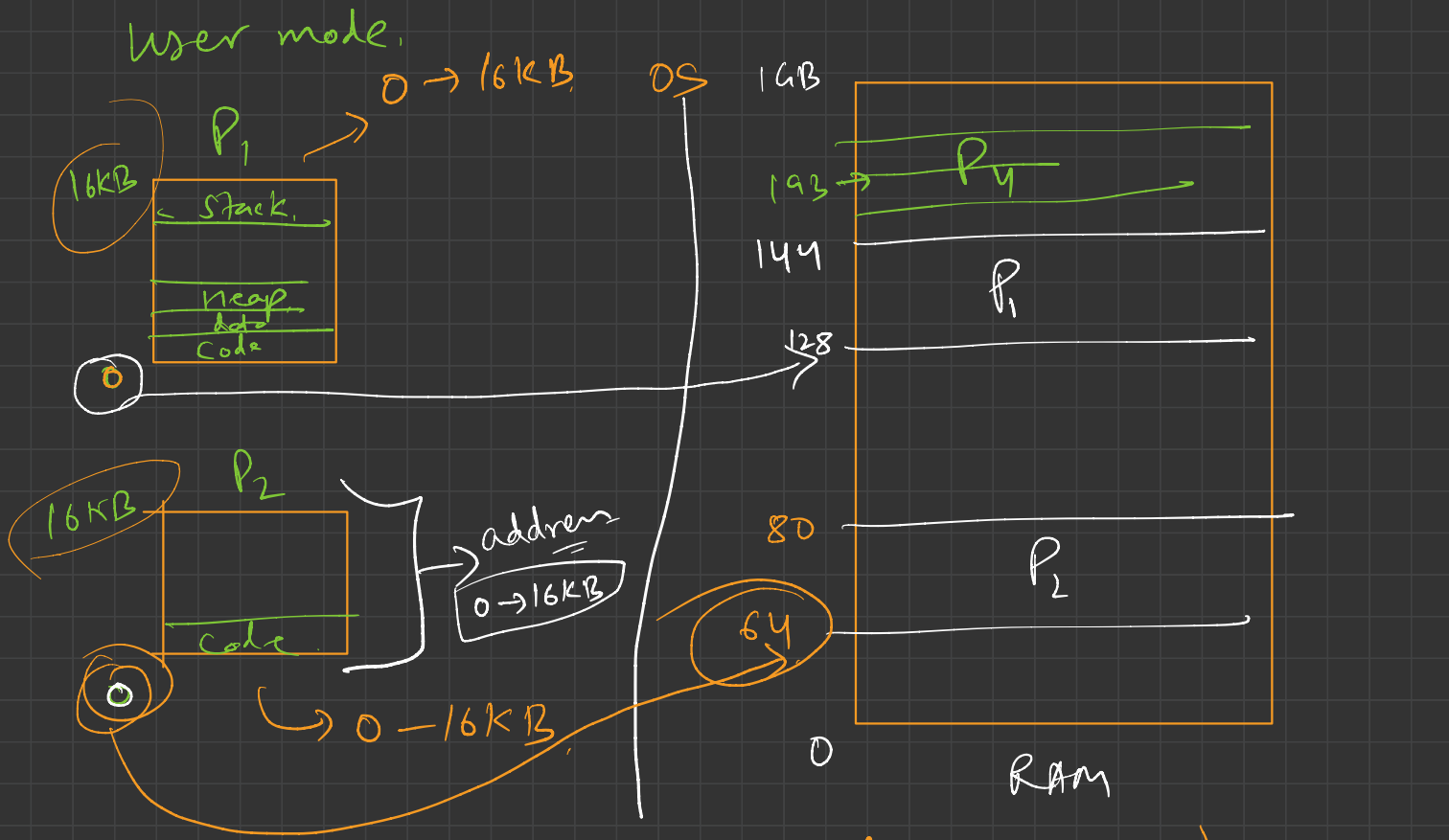
P2 P3

P1 → 50 → 60 in loc.

P1 → Program ⇒ add + 5 ⇒ data

60 + 5 ⇒ 65

access



① P_1 → Base at RAM (Physical address space)

B → 128

offset → 16

P_2 → B → 64 → start.

offset → 16

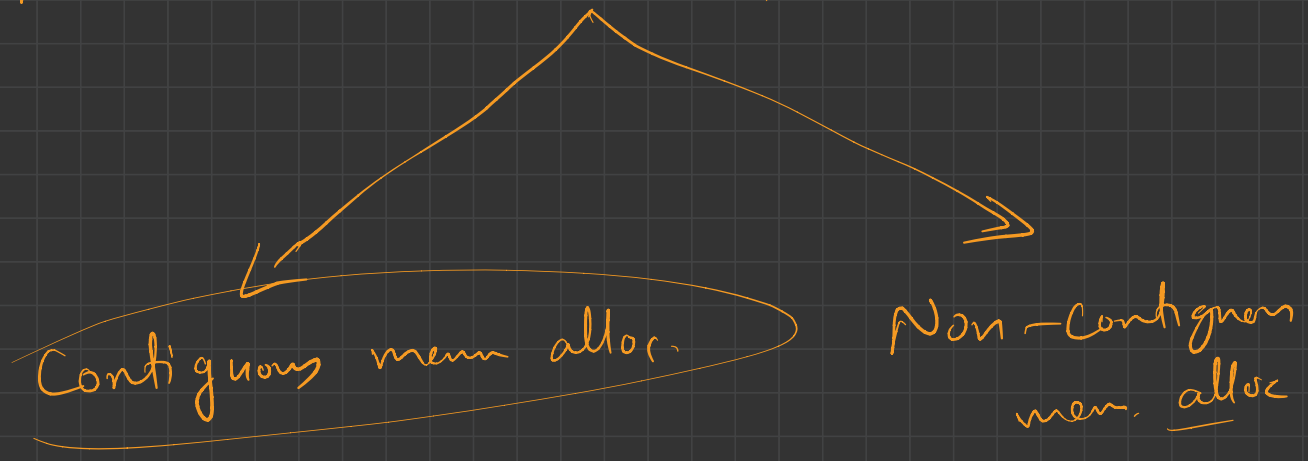
$P_2 \Rightarrow$ add $\Rightarrow$ 0 + Random value

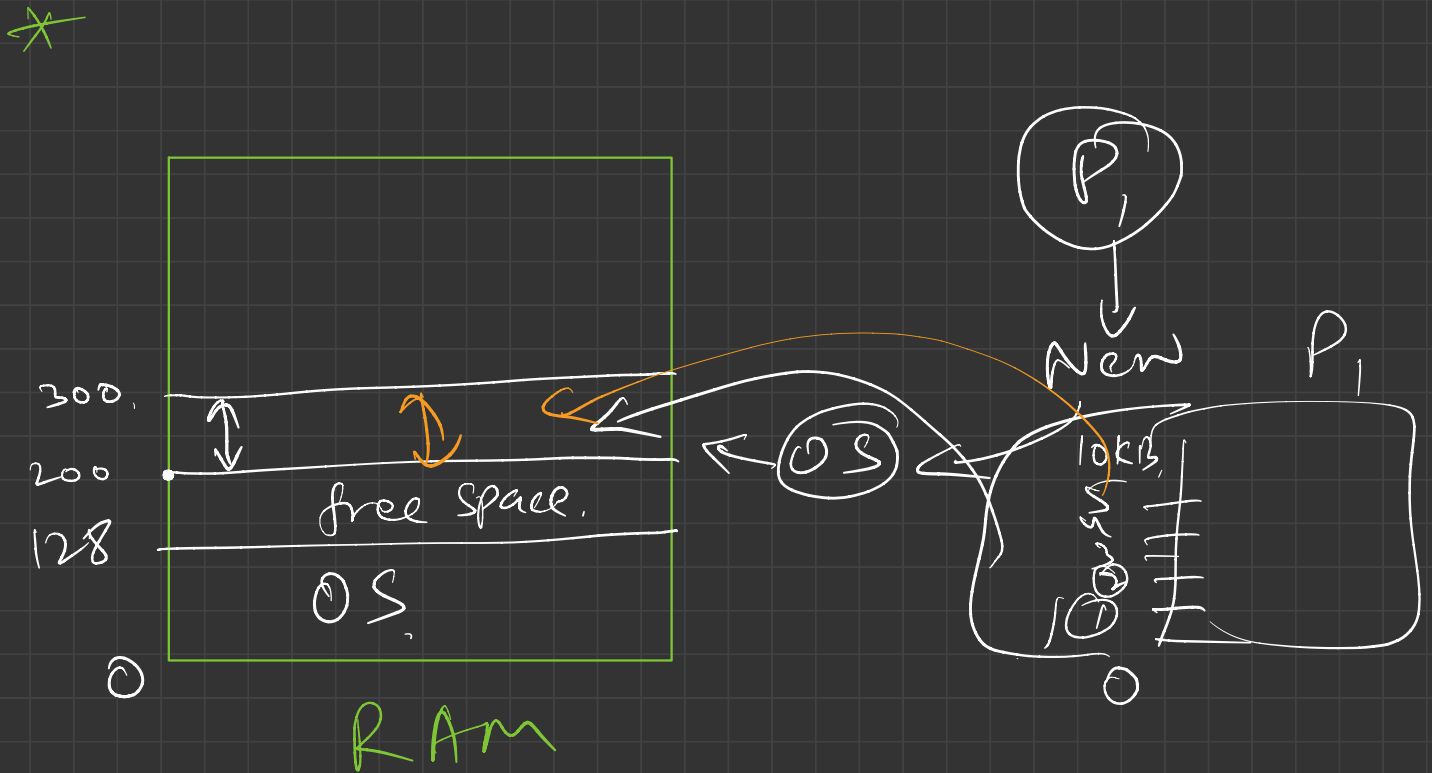
0 + 129

add $\Rightarrow$ 129.

actual Physical add: 64 + 129 = 193

→ Allocation methods in physical memory

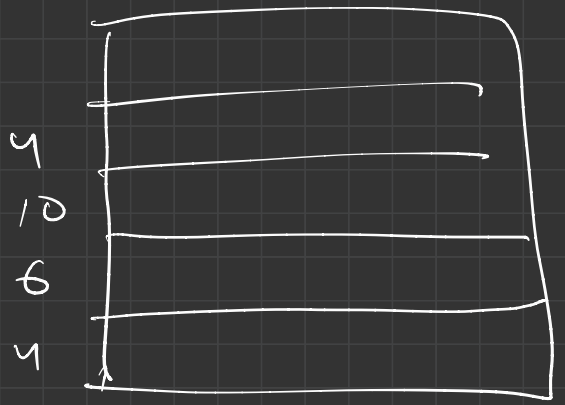



P1
New
P1
300
200
128
0
free space
OS
OS
10KB
swap
100
0
RAM

① Contiguous → each process is contained in a single contiguous block.

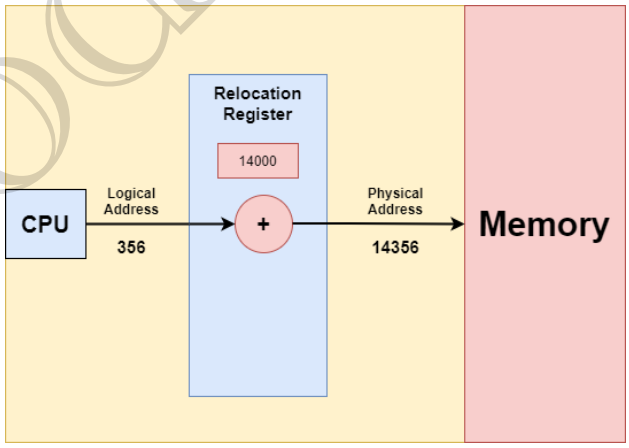
①

fixed partitioning



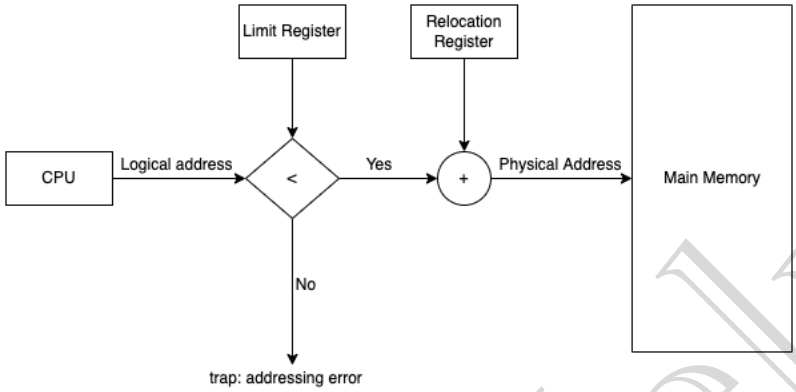
LEC-24: Memory Management Techniques | Contiguous Memory Allocation

1. In Multi-programming environment, we have multiple processes in the main memory (Ready Queue) to keep the CPU utilization high and to make computer responsive to the users.
2. To realize this increase in performance, however, we must keep several processes in the memory; that is, we must **share** the main memory. As a result, we must **manage** main memory for all the different processes.
3. **Logical versus Physical Address Space**
    - **a.** Logical Address
        - **i.** An address generated by the **CPU**.
        - **ii.** The logical address is basically the address of an instruction or data used by a process.
        - **iii.** User can access logical address of the process.
        - **iv.** User has indirect access to the physical address through logical address.
        - **v.** Logical address does not exist physically. Hence, aka, **Virtual address.**
        - vi. The set of all logical addresses that are generated by any program is referred to as Logical Address Space.
        - **vii. Range: 0 to *max*.**
    - b. Physical Address
        - i. An address loaded into the memory-address register of the physical memory.
        - ii. User can never access the physical address of the Program.
        - iii. The physical address is in the memory unit. It's a location in the main memory physically.
        - iv. A physical address can be accessed by a user indirectly but not directly.
        - v. The set of all physical addresses corresponding to the Logical addresses is commonly known as Physical Address Space.
        - vi. It is computed by the **Memory Management Unit (MMU).**
        - **vii. Range: (R + 0) to (R + max), for a base value R.**
    - **c. The runtime mapping from virtual to physical address is done by a hardware device called the memory-management unit (MMU).**
    - d. The user's program mainly generates the logical address, and the user thinks that the program is running in this logical address, but the program mainly needs physical memory in order to complete its execution.
    - e.

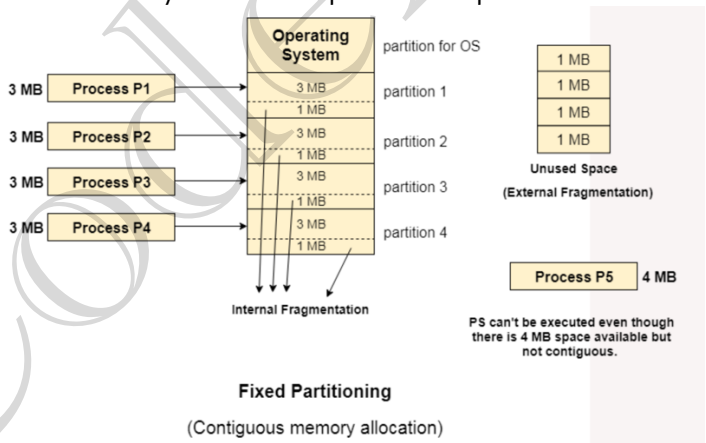


4. How OS manages the isolation and protect? **(Memory Mapping and Protection)**
    - a. OS provides this Virtual Address Space (VAS) concept.
    - b. To separate memory space, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
    - c. The relocation register contains value of smallest physical address (Base address [R]); the limit register contains the range of logical addresses (e.g., relocation = 100040 & limit = 74600).
    - d. Each logical address must be less than the limit register.

e. MMU maps the logical address dynamically by adding the value in the relocation register.
f. When CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Since every address generated by the CPU (Logical address) is checked against these registers, we can protect both OS and other users' programs and data from being modified by running process.
g. Any attempt by a program executing in user mode to access the OS memory or other uses' memory results in a trap in the OS, which treat the attempt as a fatal error.
h. **Address Translation**



5. **Allocation Method on Physical Memory**
    a. Contiguous Allocation
    b. Non-contiguous Allocation
6. **Contiguous Memory Allocation**
    a. In this scheme, each process is contained in a single contiguous block of memory.
    b. Fixed Partitioning
        i. The main memory is divided into partitions of equal or different sizes.

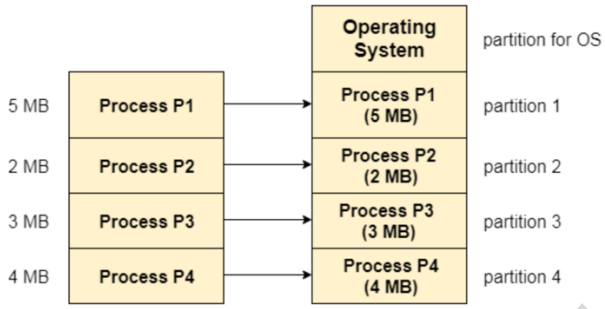


        ii.
        iii. Limitations:
            1. **Internal Fragmentation:** if the size of the process is lesser then the total size of the partition then some size of the partition gets wasted and remain unused. This is wastage of the memory and called internal fragmentation.
            2. **External Fragmentation:** The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.
            3. Limitation on process size: If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.

4. Low degree of multi-programming: In fixed partitioning, the degree of multiprogramming is fixed and very less because the size of the partition cannot be varied according to the size of processes.

c. Dynamic Partitioning

i. In this technique, the partition size is not declared initially. It is declared at the time of process loading.



Dynamic Partitioning

(Process Size = Partition Size)

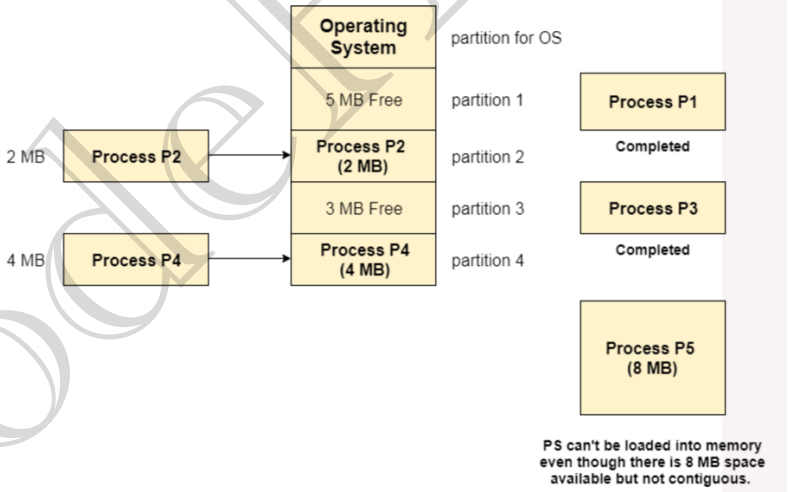
ii.

iii. Advantages over fixed partitioning

1. No internal fragmentation
2. No limit on size of process
3. Better degree of multi-programming

iv. Limitation

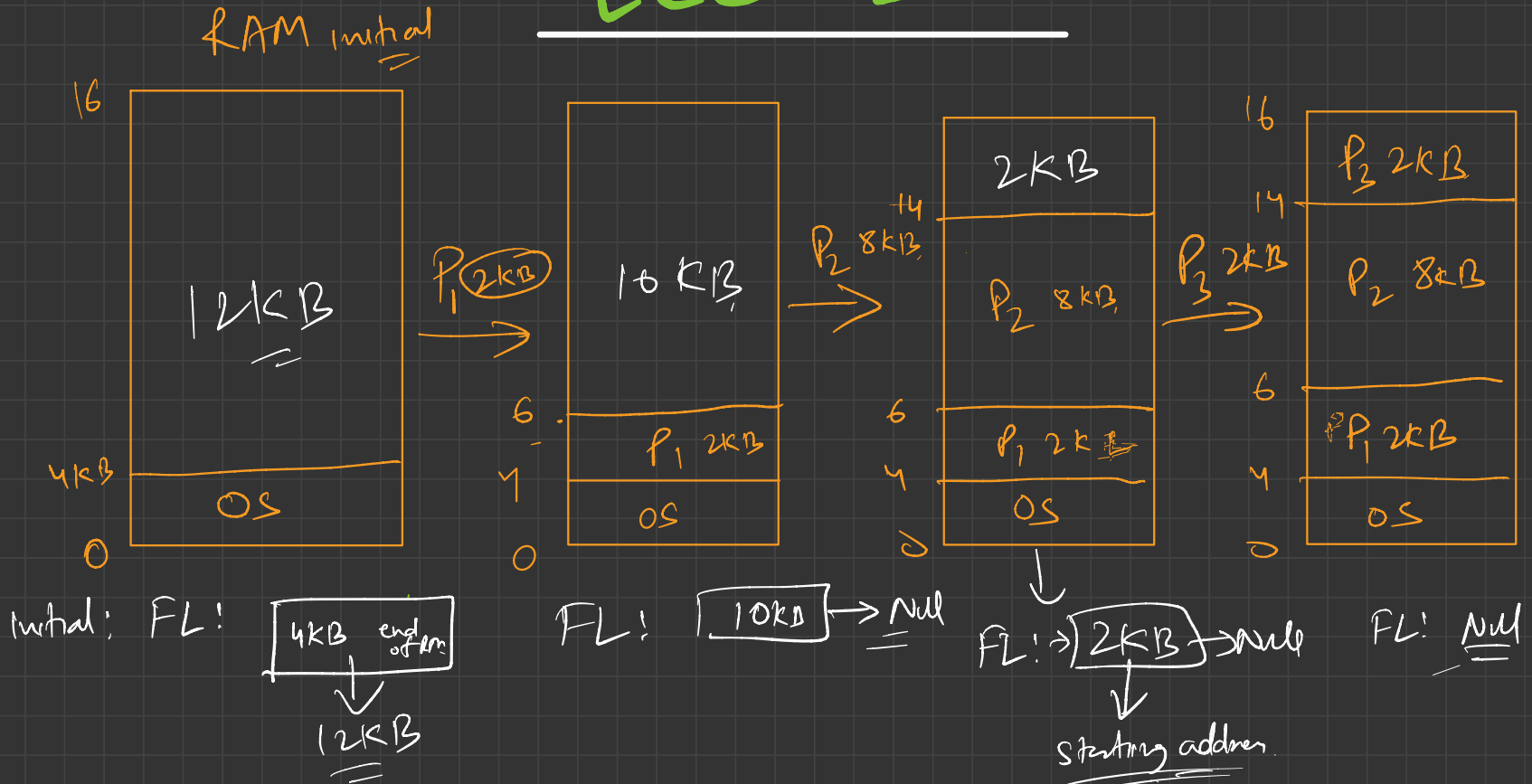
1. External fragmentation



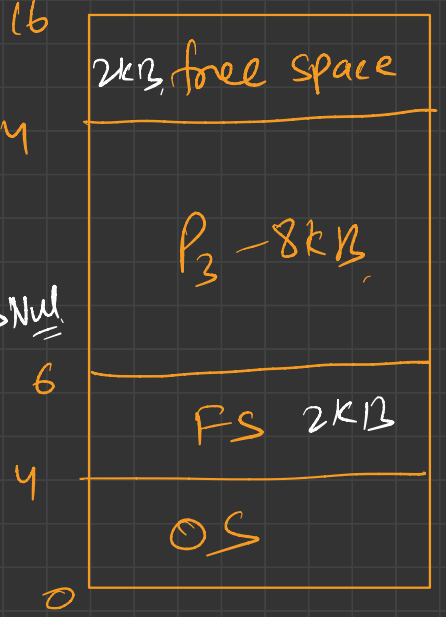
External Fragmentation in Dynamic Partitioning

X

Lec-25


RAM initial
16
12KB
4KB
OS
0
P1 2KB
10KB
6
P1 2KB
4
OS
0
P2 8KB
2KB
14
P2 8KB
6
P1 2KB
4
OS
0
P3 2KB
16
P3 2KB
14
P2 8KB
6
P1 2KB
4
OS
0
Initial: FL: 4KB end of RAM
12KB
FL: 10KB → Null
FL: → 2KB → Null
starting address
FL: Null

After P_1 & P_2 exit



FL: 2KB → 2KB → Null

OS

F.S

free List

↓

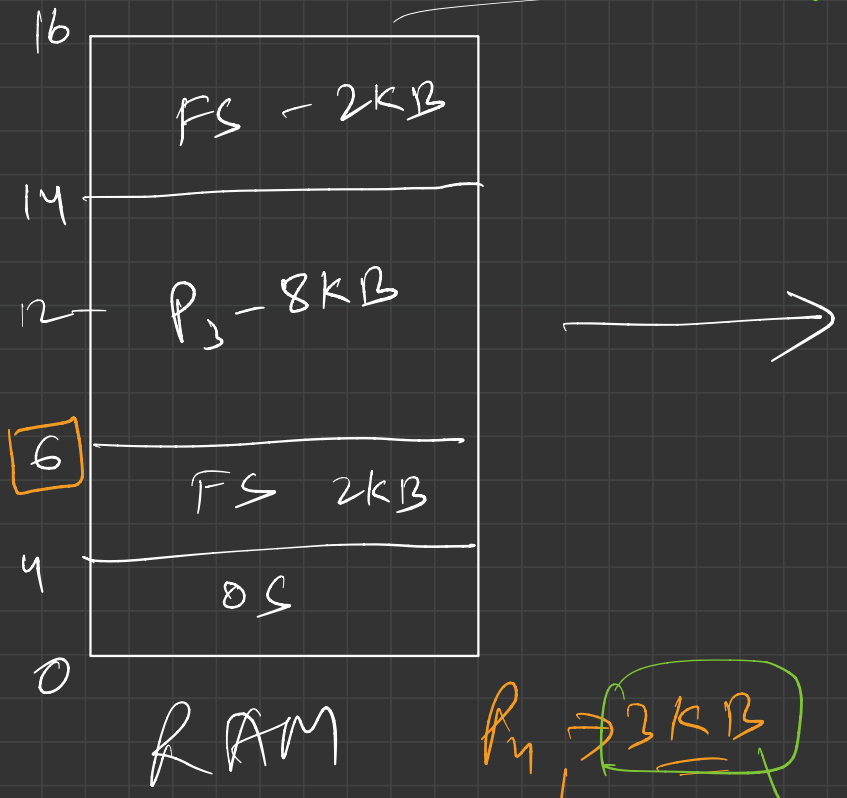
Linked list

Problem → P_4 → 3KB

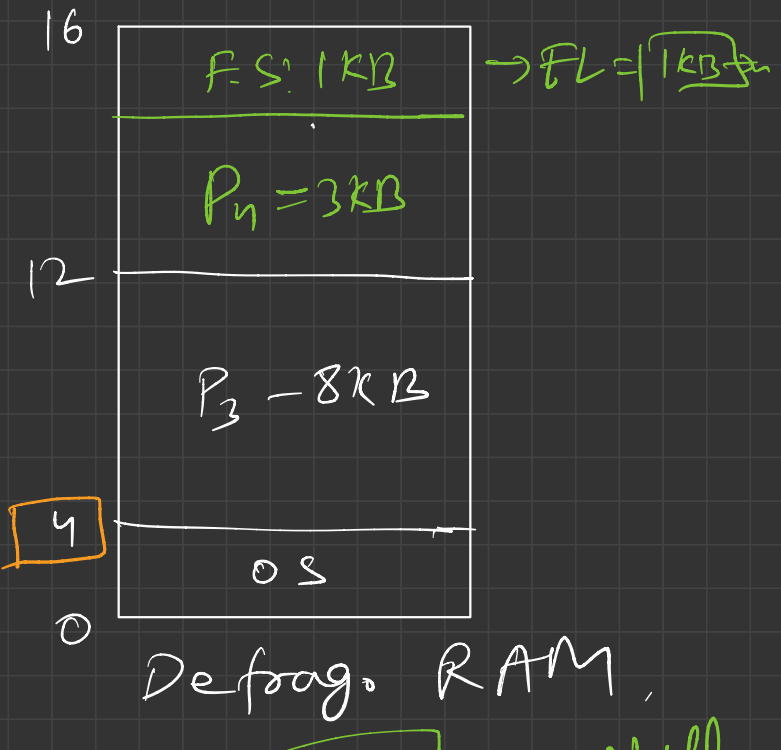
↳ Total free mem RAM ⇒ 4KB

Req → 3KB →

* Defragmentation / Compaction



RAM



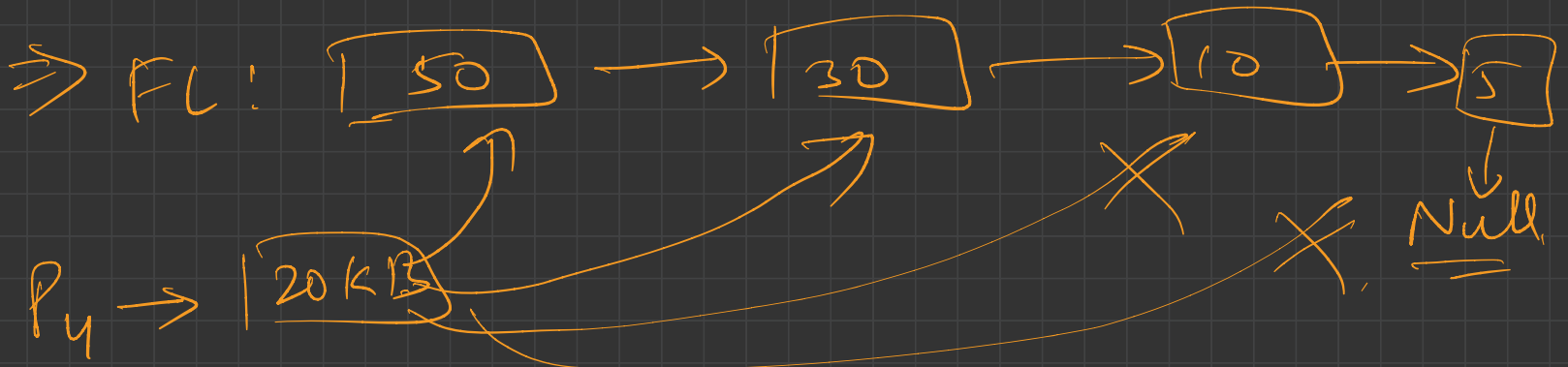
Defrag. RAM.

P4 → 3KB

OS? →

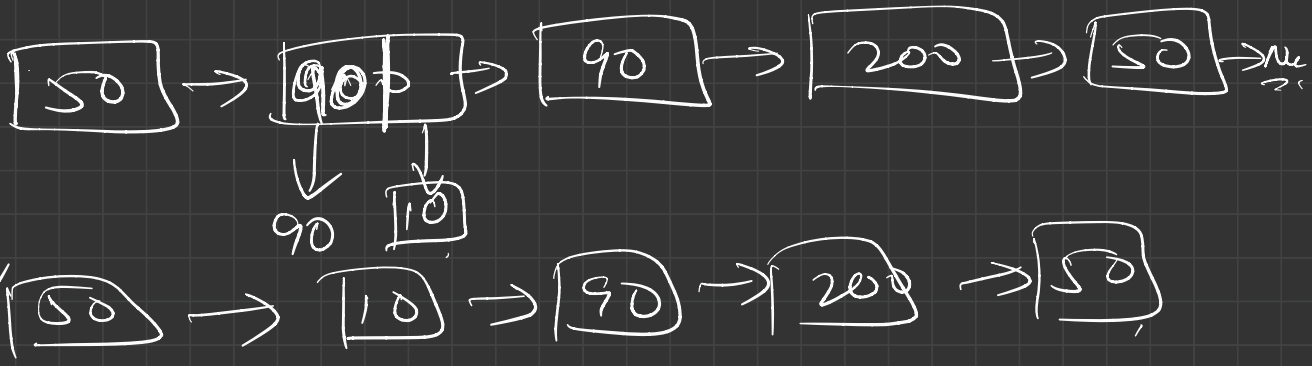
FL: 4K → Null.

Yes →


FL:
50
30
10
5
Null
P4
20KB

① First fit :- Req: 90 KB → P1

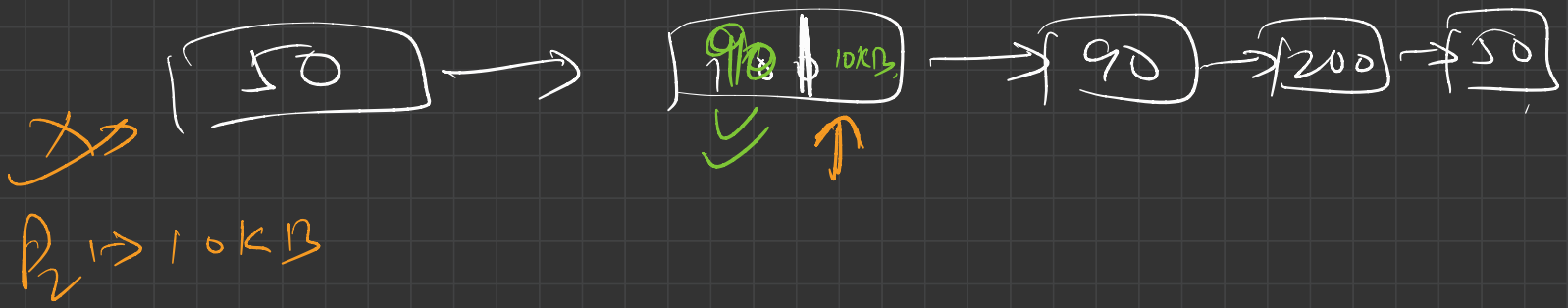
FL:



⇒ fast !

⇒ Simple / easy to implement

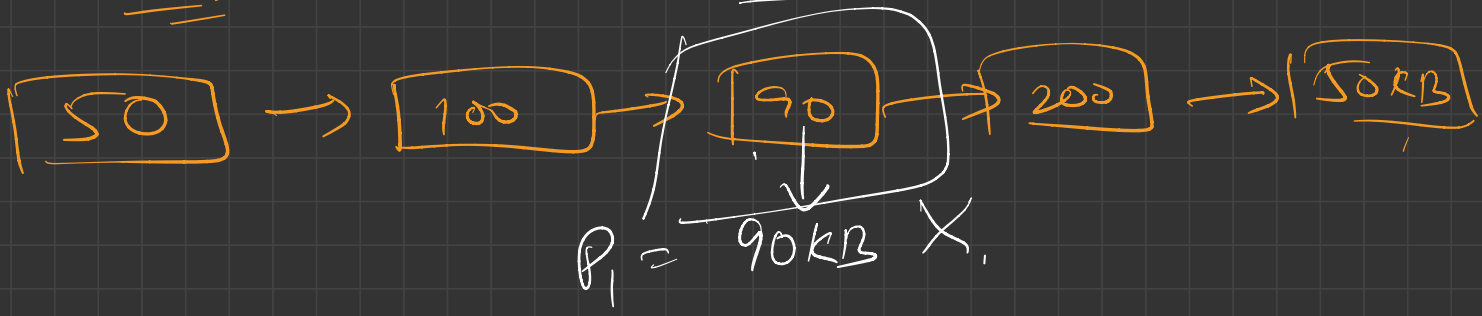
② Next fit 90KB



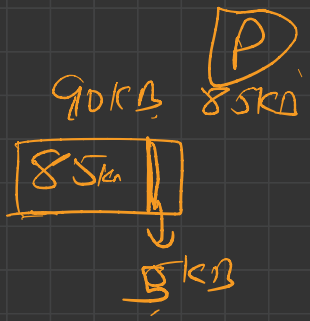
P_2 => 10KB

② enhancement/tweak Firstfit

③ Best fit $P_1 \rightarrow$ 90 KB

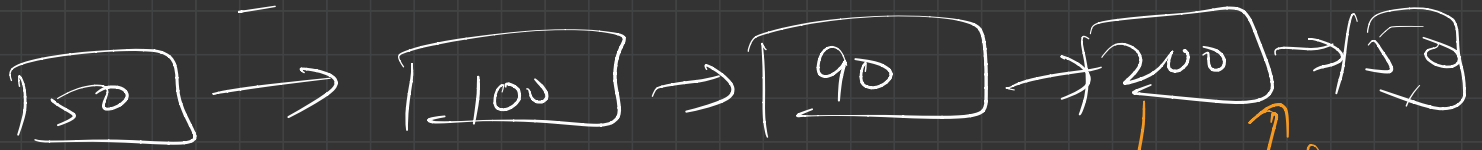


⇒ Lesser internal frag.
⇒ slow
⇒ major external frag.

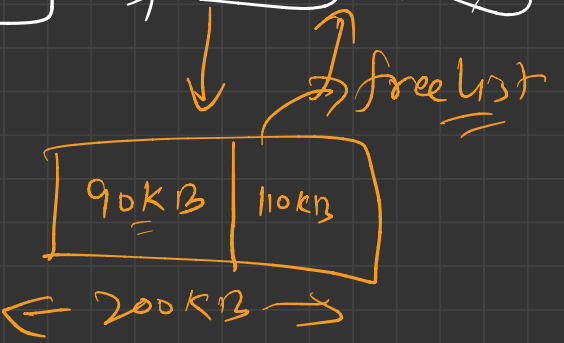


P_1; 2KB → 3KB
→ 5KB → 1KB

(4) Worst fit → allocate largest hole.



Req → 90KB



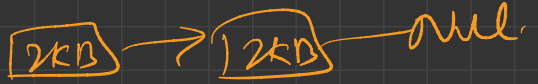
⇒ Less external fragmentation

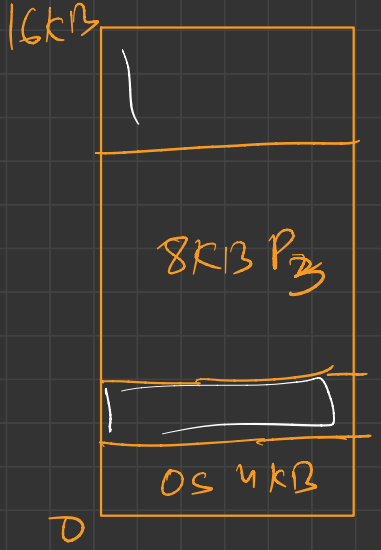
⇒ Slow.

LEC-25: Free Space Management

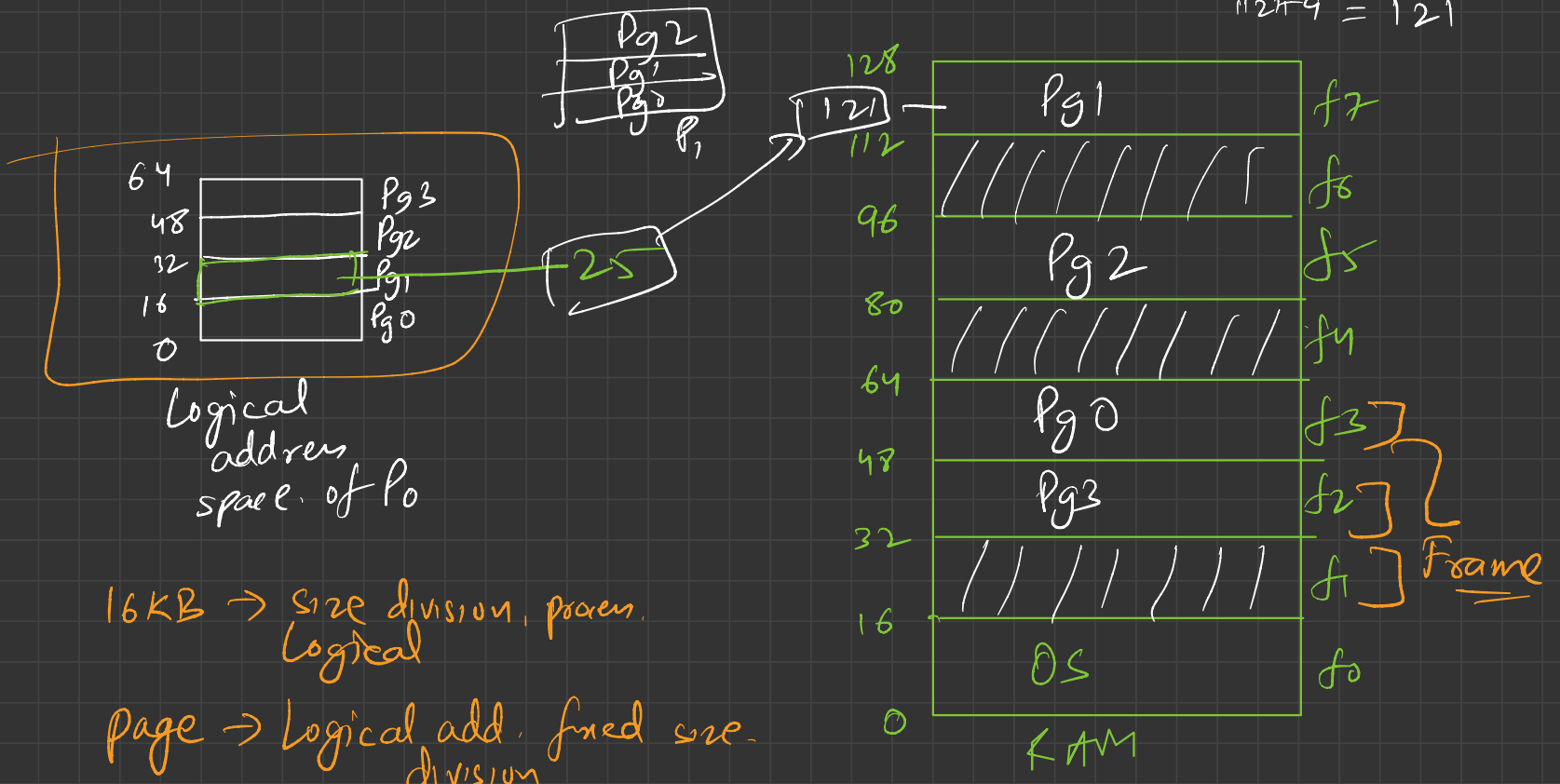
1. **Defragmentation/Compaction**
    a. Dynamic partitioning suffers from external fragmentation.
    b. Compaction to minimize the probability of external fragmentation.
    c. All the free partitions are made contiguous, and all the loaded partitions are brought together.
    d. By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called **defragmentation**.
    e. The efficiency of the system is decreased in the case of compaction since all the free spaces will be transferred from several places to a single place.
2. **How free space is stored/represented in OS?**
    a. Free holes in the memory are represented by a free list (Linked-List data structure).
3. **How to satisfy a request of a of n size from a list of free holes?**
    a. Various algorithms which are implemented by the Operating System in order to find out the holes in the linked list and allocate them to the processes.
    b. First Fit
        i. Allocate the first hole that is big enough.
        ii. Simple and easy to implement.
        iii. Fast/Less time complexity
    c. Next Fit
        i. Enhancement on First fit but starts search always from last allocated hole.
        ii. Same advantages of First Fit.
    d. Best Fit
        i. Allocate smallest hole that is big enough.
        ii. Lesser internal fragmentation.
        iii. May create many small holes and cause major external fragmentation.
        iv. Slow, as required to iterate whole free holes list.
    e. Worst Fit
        i. Allocate the largest hole that is big enough.
        ii. Slow, as required to iterate whole free holes list.
        iii. Leaves larger holes that may accommodate other processes.

Lec-26





P4 => 3KB



16KB → size division, process Logical

Page → Logical add. fixed size division

Page size = frame size

* Page table

Logical page no.	physical frame no.
00 Pg 0 → 1	frame 3 011
01 1	7 → 111
10 2	5 101
11 3	2 010

⇒ Logical address → Pg → 64 bytes.

$2^6 = 64$ p d. 01

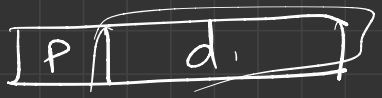
6 bits ⇒ 25 = 01 001

01 → Page no. 1

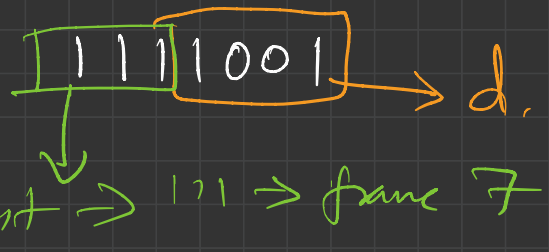
001 → d. → offset → base address. 9

Pg no 1 → Base → 16 + 9 ⇒ 25

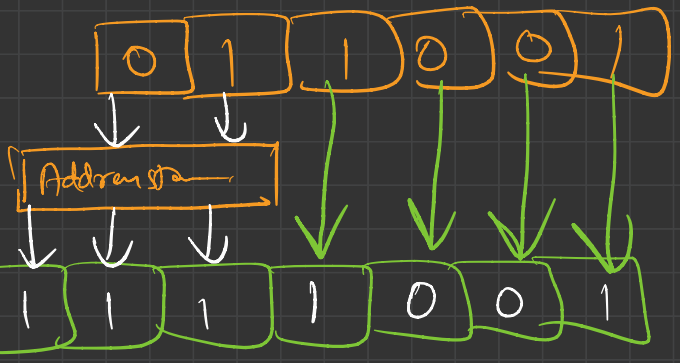
Logical addrn space.



Physical addrn

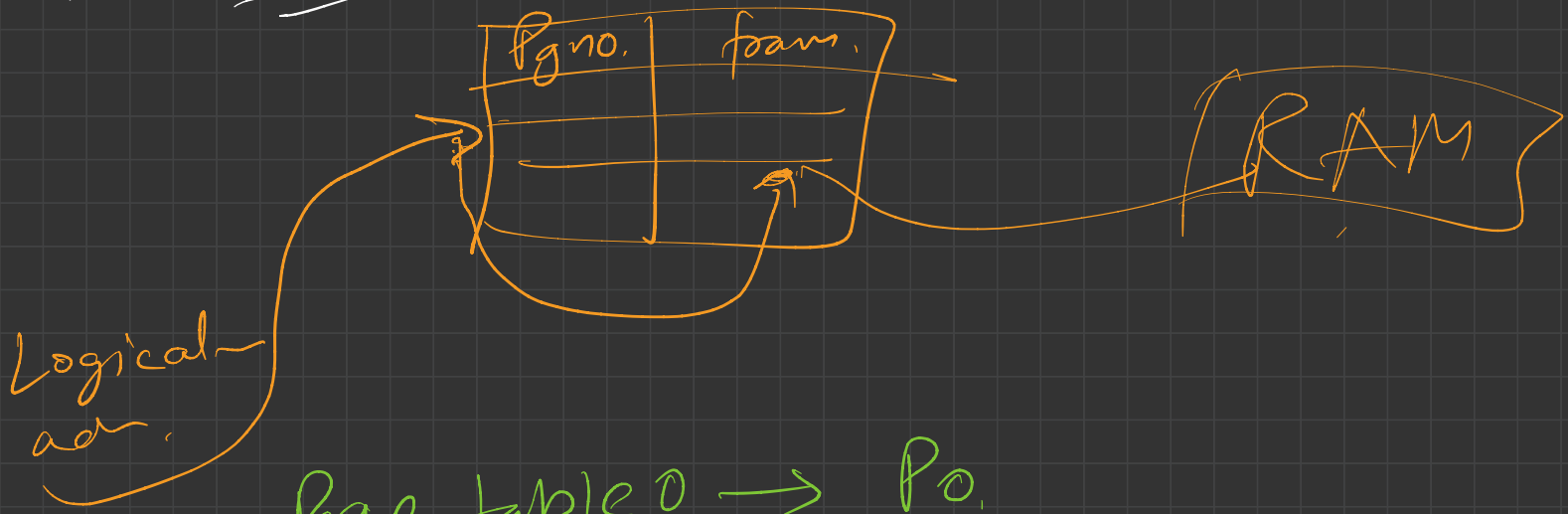
121 ⇒ 

Logical addrn



Physical addrn ⇒

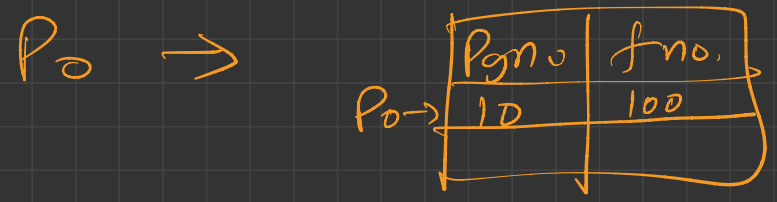
TLB



Page table 0 → P0

PT 1 → P1

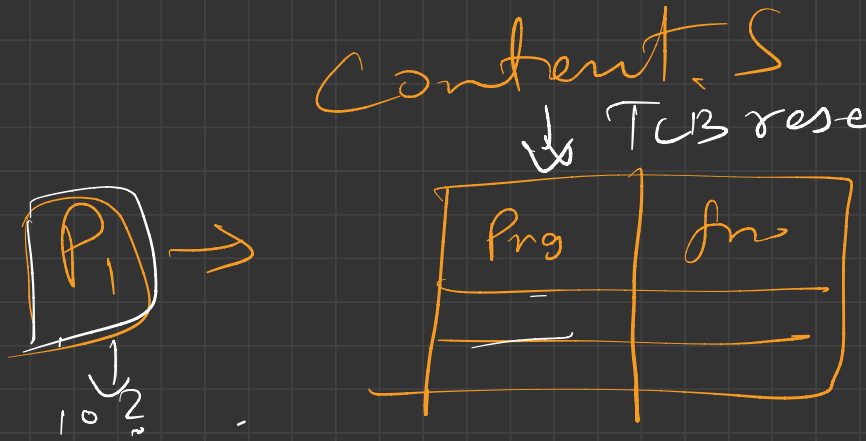
PT 2 → P2



TLB

Context S

TLB reset



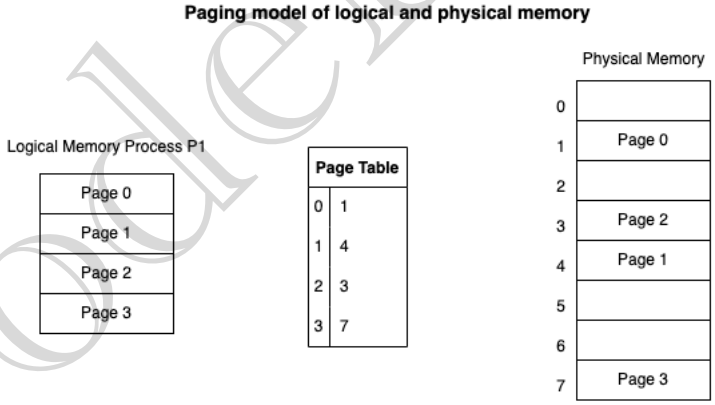
ASID	Pgn.	fno.
0	10	100
1	10	45

① flush the TLB when Context switches occurs

② unique identifier that will identify unique proces.

LEC-26: Paging | Non-Contiguous Memory Allocation

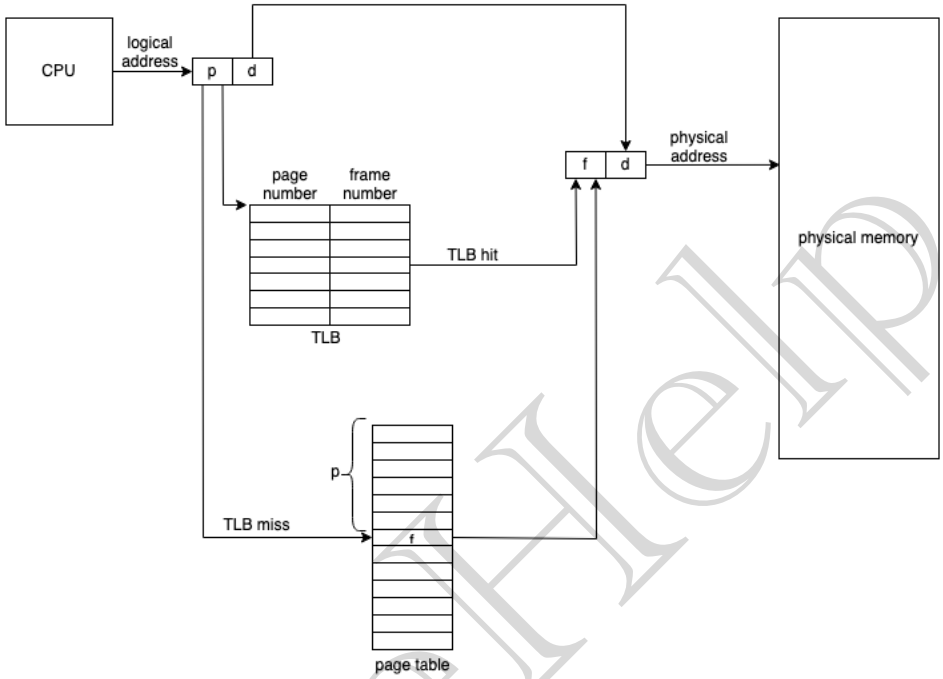
1. The main **disadvantage of Dynamic partitioning is External Fragmentation.**
    a. Can be removed by Compaction, but with overhead.
    b. We need more dynamic/flexible/optimal mechanism, to load processes in the partitions.
2. **Idea behind Paging**
    a. If we have only two small non-contiguous free holes in the memory, say 1KB each.
    b. If OS wants to allocate RAM to a process of 2KB, in contiguous allocation, it is not possible, as we must have contiguous memory space available of 2KB. (External Fragmentation)
    c. What if we divide the process into 1KB-1KB blocks?
3. Paging
    a. Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
    b. It avoids external fragmentation and the need of compaction.
    c. Idea is to divide the physical memory into fixed-sized blocks called **Frames**, along with divide logical memory into blocks of same size called **Pages**. (# Page size = Frame size)
    d. **Page size** is usually determined by the processor architecture. Traditionally, pages in a system had uniform size, such as 4,096 bytes. However, processor designs often allow two or more, sometimes simultaneous, page sizes due to its benefits.
    e. Page Table
        i. A Data structure stores which page is mapped to which frame.
        ii. The page table contains the base address of each page in the physical memory.
    f. Every address generated by CPU (logical address) is divided into two parts: a page number (p) and a page offset (d). The p is used as an index into a page table to get base address the corresponding frame in physical memory.



    g. Page table is stored in main memory at the time of process creation and its base address is stored in process control block (**PCB**).
    h. A page table base register (**PTBR**) is present in the system that points to the current page table. Changing page tables requires only this one register, at the time of context-switching.
4. **How Paging avoids external fragmentation?**
    a. Non-contiguous allocation of the pages of the process is allowed in the random free frames of the physical memory.
5. **Why paging is slow and how do we make it fast?**
    a. There are too many memory references to access the desired location in physical memory.
6. **Translation Look-aside buffer (TLB)**
    a. A Hardware support to speed-up paging process.
    b. It's a hardware cache, high speed memory.
    c. TBL has key and value.

d. Page table is stores in main memory & because of this when the memory references is made the translation is slow.
e. When we are retrieving physical address using page table, after getting frame address corresponding to the page number, we put an entry of the into the TLB. So that next time, we can get the values from TLB directly without referencing actual page table. Hence, make paging process faster.

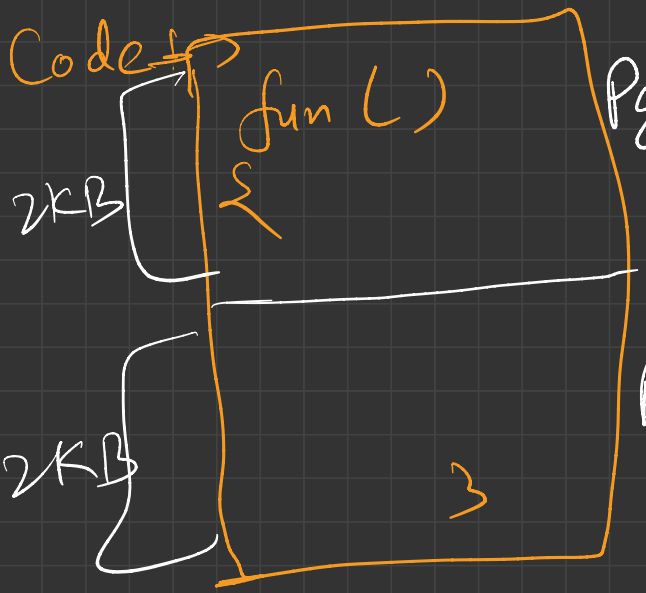
Paging hardware with TLB

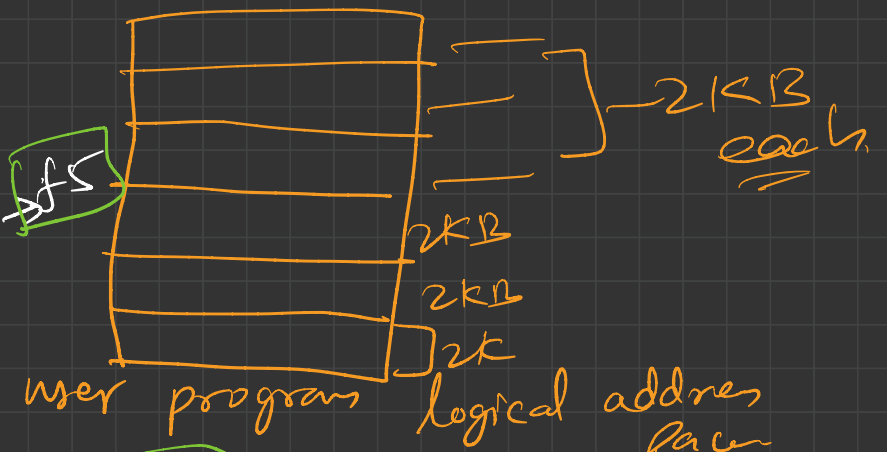


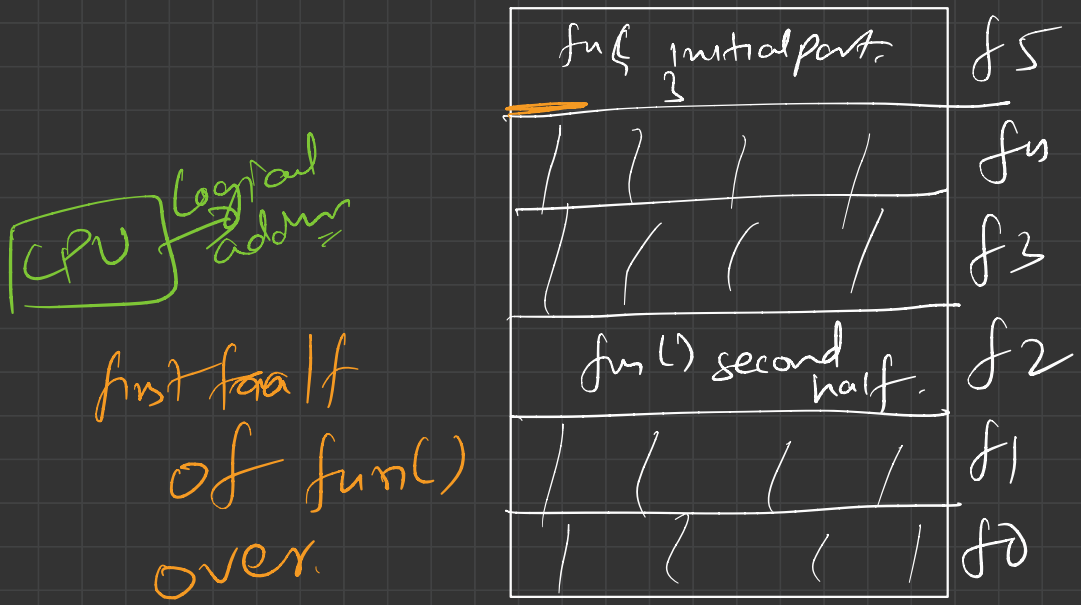
f. TLB hit, TLB contains the mapping for the requested logical address.
g. **Address space identifier (ASIDs)** is stored in each entry of TLB. ASID uniquely identifies each process and is used to **provide address space protection and allow to TLB to contain entries for several different processes.** When TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently executing process matches the ASID associated with virtual page. If it doesn't match, the attempt is treated as TLB miss.

Lec-27

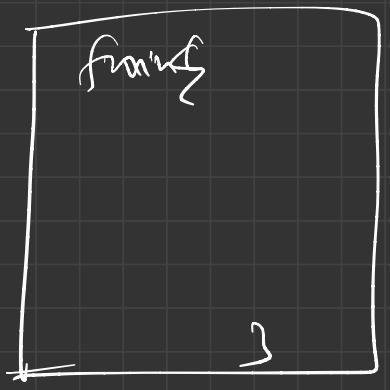
Paging :> Problems





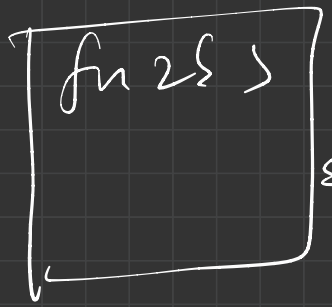

fun() initial part.
3
f5
f4
f3
fun() second half.
f2
f1
f0
CPU
Logical address
first half of fun() over.

* Segmentation

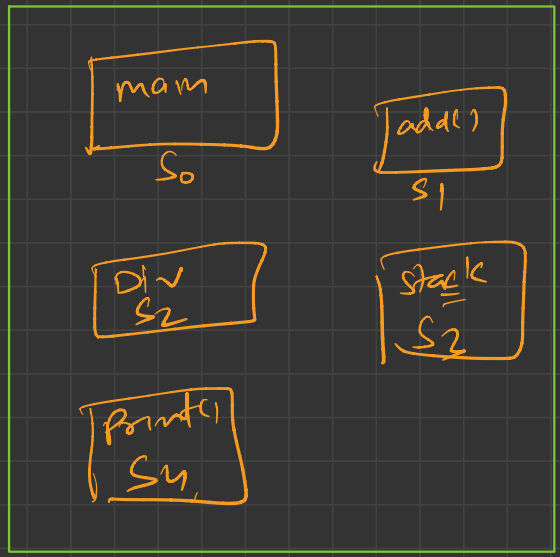


→ Variable partitioning of Logical address space.

→ diff. segments
→ varying sizes.
→ user view



segment

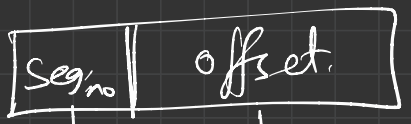


Segment size diff

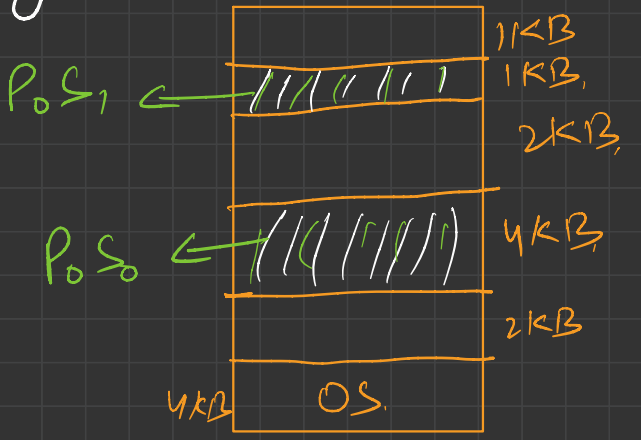
* MMU => Translation ??

LA → PA ?

=> LA =>



=> Ext frag. => in Segmentation



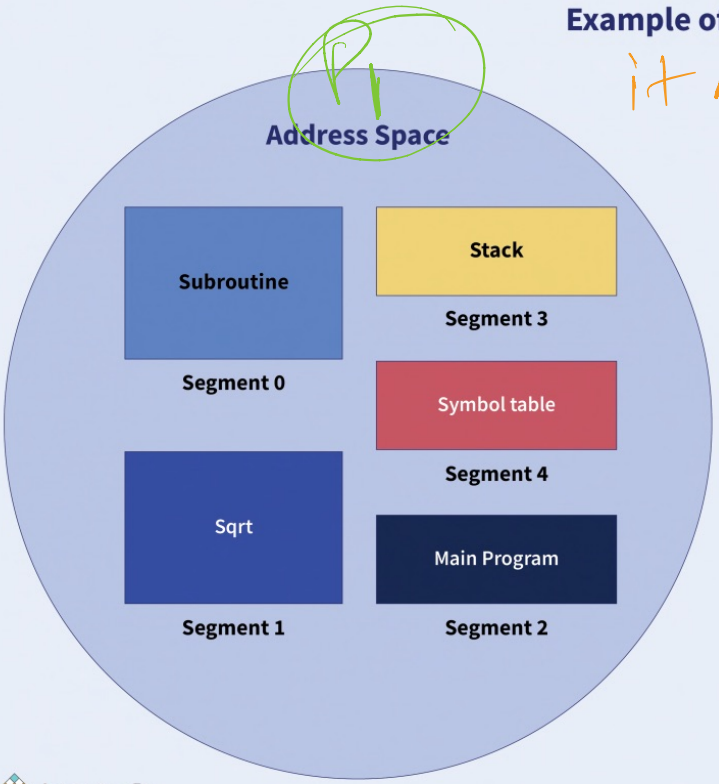
$P_0 S_1$

$P_0 S_0$

$P_1 \rightarrow S_0 \rightarrow 2KB$

$S_1 \rightarrow 3KB$

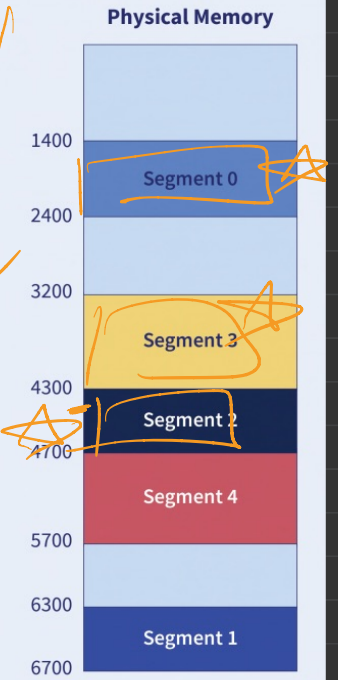
Example of Segmentation



Segment Table

Limit	Base
1000	1400
400	6300
400	4300
1100	3200
1000	4700

Ex: Segment 2 = 4300 +53 = 4353



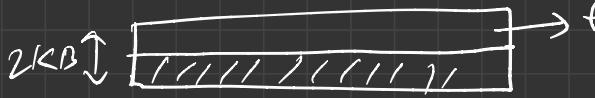
InterviewBit

Internal frag. in paging

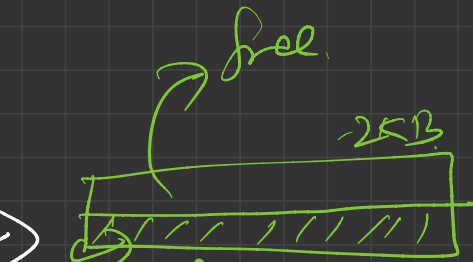
Program → 15 KB

Page size → 2KB

16KB → ✓

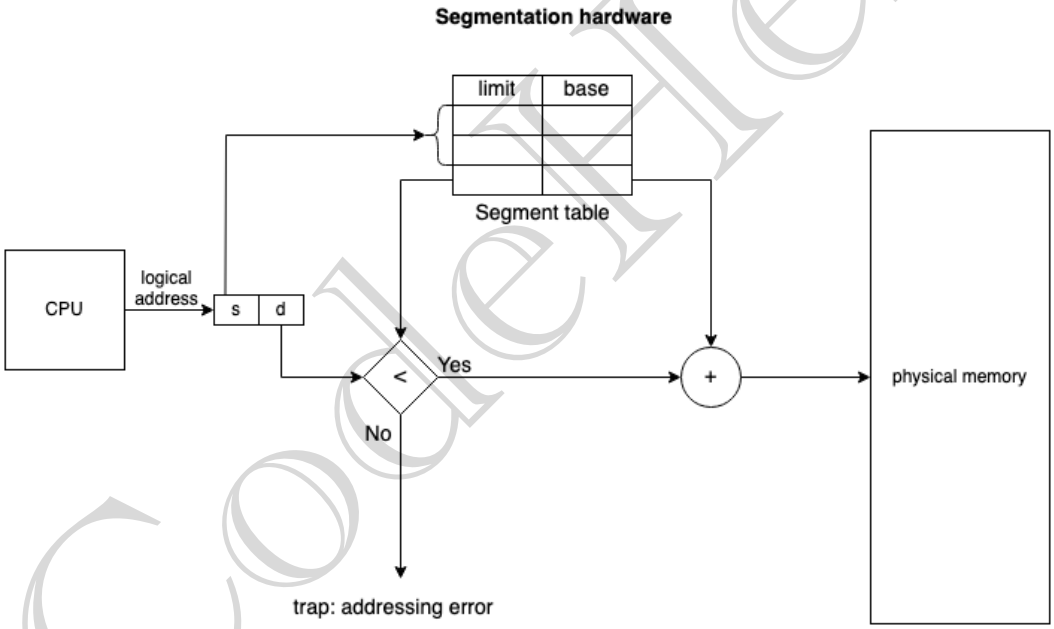


Page.



Return

LEC-27: Segmentation | Non-Contiguous Memory Allocation

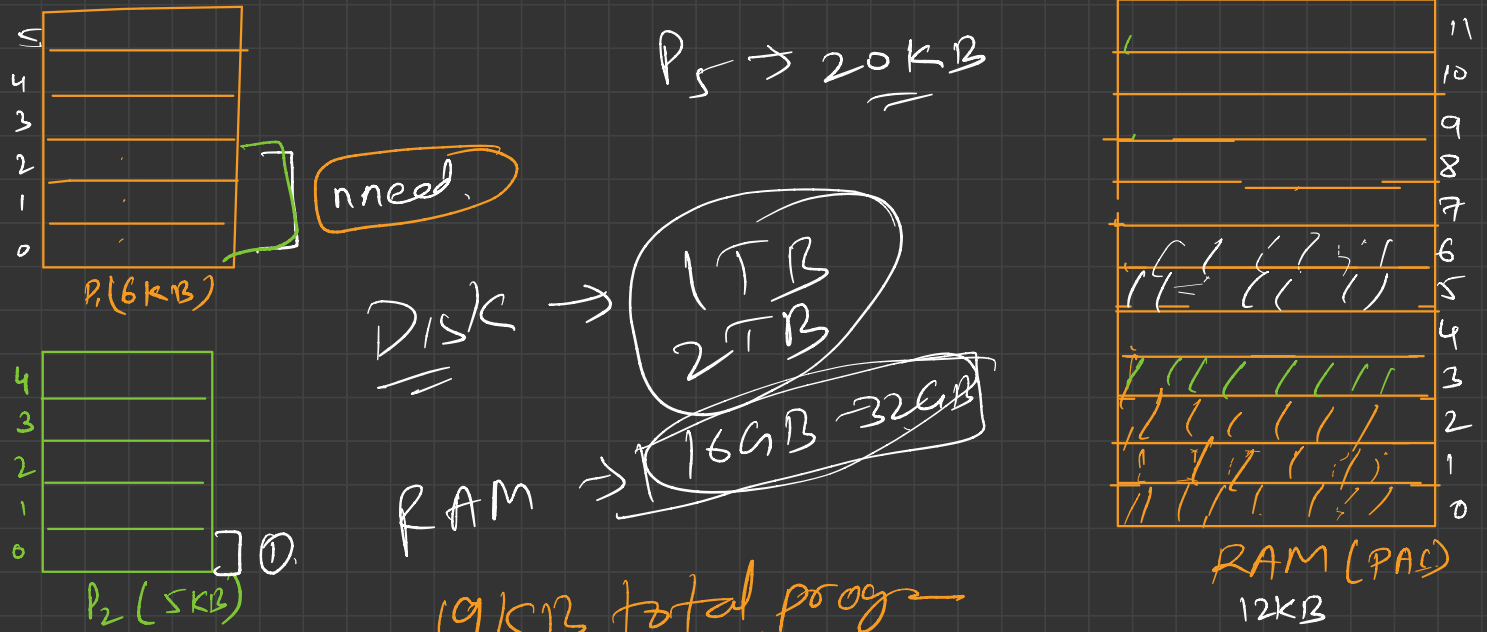
1. An important aspect of memory management that become unavoidable with paging is separation of user's view of memory from the actual physical memory.
2. Segmentation is memory management technique that supports the **user view of memory.**
3. A logical address space is a collection of segments, these segments are based on **user view** of logical memory.
4. Each segment has **segment number and offset,** defining a segment.
   <segment-number, offset> {s,d}
5. Process is divided into **variable segments based on user view.**
6. **Paging** is closer to the Operating system rather than the **User**. It divides all the processes into the form of pages although a process can have some relative parts of functions which need to be loaded in the same page.
7. Operating system doesn't care about the **User's view** of the process. It may **divide the same function into different page**s and those **pages may or may not be loaded at the same time into the memory.** It decreases the efficiency of the system.
8. It is better to have segmentation which divides the process into the segments. Each segment contains the same type of functions such as the main function can be included in one segment and the library functions can be included in the other segment.
9. 

10. **Advantages**:
    a. No internal fragmentation.
    b. One segment has a contiguous allocation, hence efficient working within segment.
    c. The size of segment table is generally less than the size of page table.
    d. It results in a more efficient system because the compiler keeps the same type of functions in one segment.
11. **Disadvantages**:
    a. External fragmentation.
    b. The different size of segment is not good that the time of swapping.
12. Modern System architecture provides both segmentation and paging implemented in some hybrid approach.

Lec-28

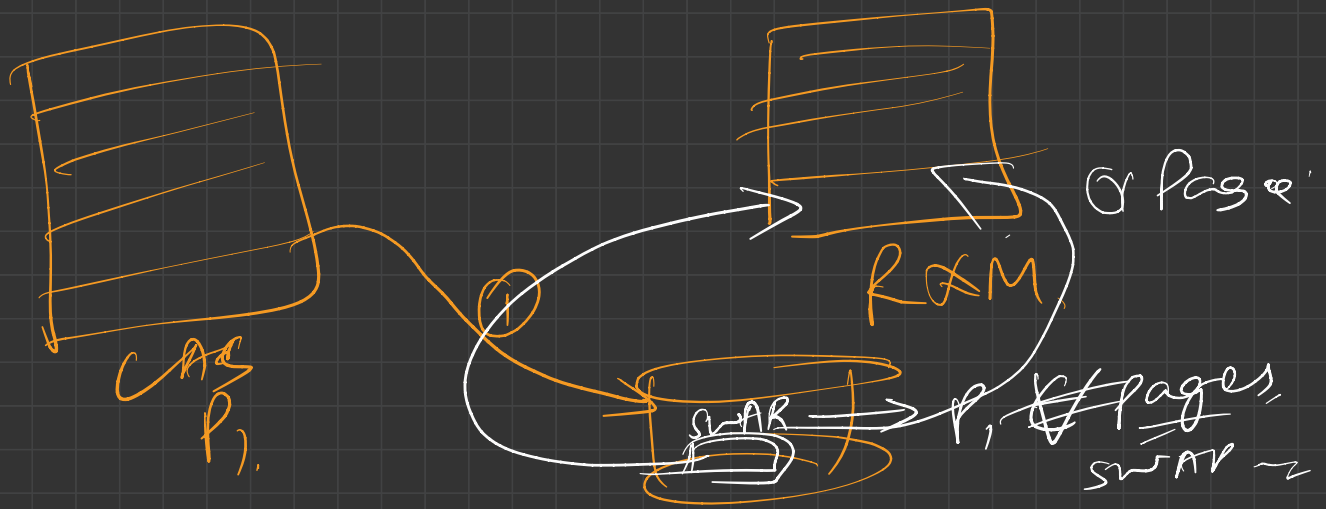
* Virtual memory management

→ Physical memory (PAS)
→ Logical memory (LAS)


5
4
3
2
1
0
P1 (6KB)
nneed.
P5 → 20KB
Disk →
1TB
2TB
RAM →
16GB 32GB
4
3
2
1
0
P2 (5KB)
11
10
9
8
7
6
5
4
3
2
1
0
RAM (PAS)
12KB
19KB total prog
12KB RAM
4
3
2
1
0
needed.
P3 (5KB)
P4. → 3 pages.
DDD DDDD
DDD
1 GB
swap space
Disk

RAM + SWAP AREA → Virtual memory

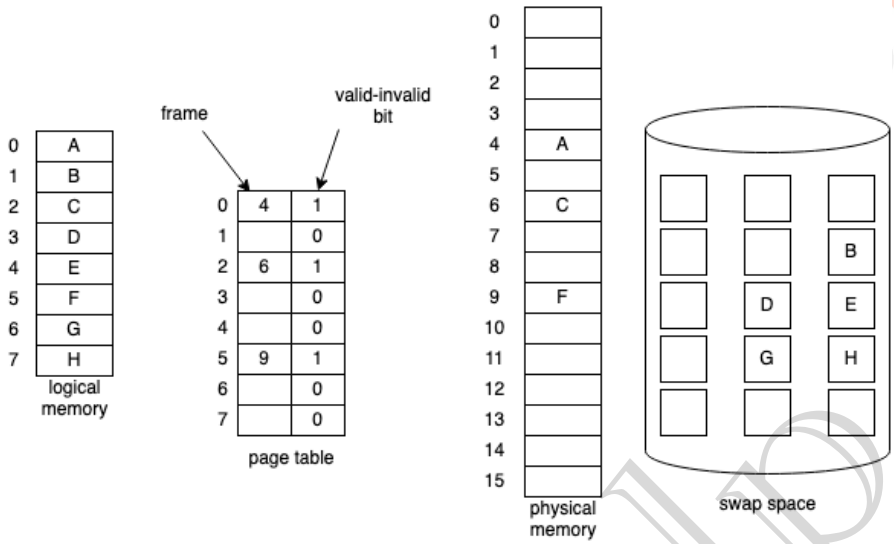
* Demand Paging



LEC-28: What is Virtual Memory? || Demand Paging || Page Faults

1. **Virtual memory** is a technique that allows the execution of processes that are not completely in the memory. It provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory. (Swap-space)
2. **Advantage** of this is, programs can be larger than physical memory.
3. It is required that instructions must be in physical memory to be executed. But it limits the size of a program to the size of physical memory. In fact, in many cases, the entire program is not needed at the same time. So, we want an ability to execute a program that is only partially in memory would give many benefits:
    a. A program would no longer be constrained by the amount of physical memory that is available.
    b. Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding **increase in CPU utilization and throughput.**
    c. Running a program that is not entirely in memory would benefit **both the system and the user.**
4. Programmer is provided very large virtual memory when only a smaller physical memory is available.
5. **Demand Paging** is a popular method of **virtual memory management.**
6. In demand paging, the pages of a process which are least used, get stored in the secondary memory.
7. A page is copied to the main memory when its demand is made, or **page fault** occurs. There are various **page replacement algorithms** which are used to determine the pages which will be replaced.
8. Rather than swapping the entire process into memory, we use **Lazy Swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.
9. We are viewing a process as a sequence of pages, rather than one large contiguous address space, using the term **Swapper is technically incorrect**. A swapper manipulates entire processes, whereas a **Pager** is concerned with individual pages of a process.
10. **How Demand Paging works?**
    a. When a process is to be swapped-in, the pager guesses which pages will be used.
    b. Instead of swapping in a whole process, the pager brings only those pages into memory. This, it avoids reading **into memory pages that will not be used anyway.**
    c. Above way**, OS decreases the swap time and the amount of physical memory needed.**
    d. The **valid-invalid bit scheme in the page table** is used to distinguish between pages that are in memory and that are on the disk.
        i. Valid-invalid bit **1** means, the associated page is both legal and in memory.
        ii. Valid-invalid bit **0** means, the page either is not valid (not in the LAS of the process) or is valid but is currently on the disk.

Page table when some pages are not in memory



e.

f. If a process never attempts **to access some invalid bit page, the process will be executed successfully without even the need pages present in the swap space.**

g. What happens if the process tries to access a page that was not brought into memory, access to a page marked invalid causes **page fault.** Paging hardware noticing invalid bit for a demanded page will cause **a trap to the OS.**

h. The procedure to handle the page fault:

i. Check an internal table (in PCB of the process) to determine whether the reference was valid or an invalid memory access.

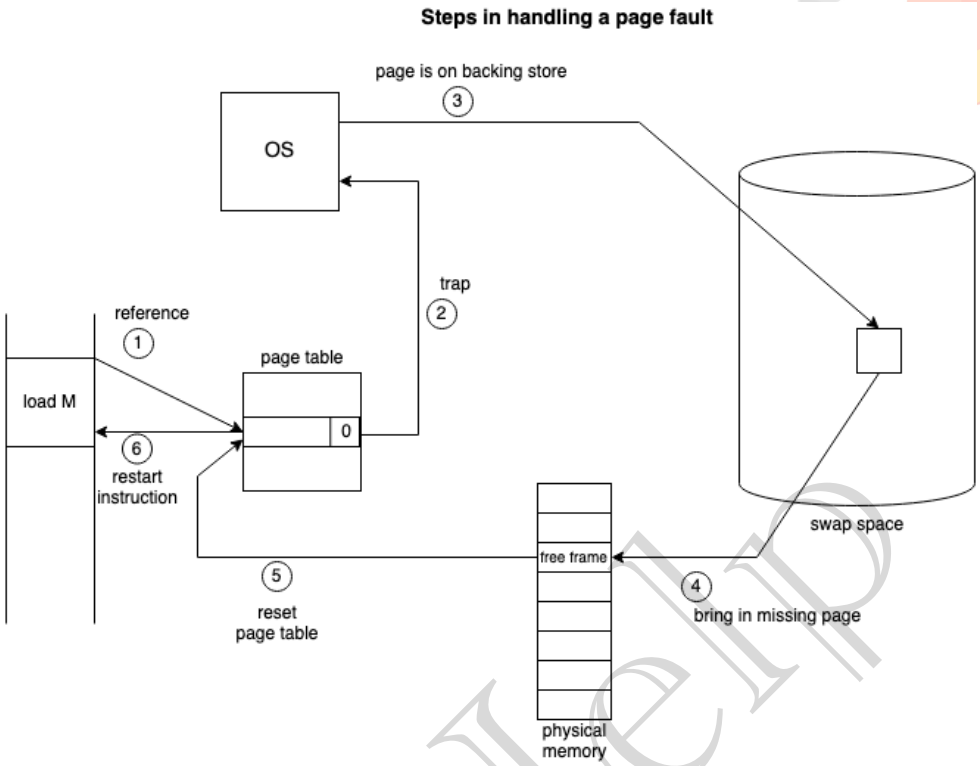
ii. If ref. was invalid process throws exception.
If ref. is valid, pager will swap-in the page.

iii. We find a free frame (from free-frame list)

iv. Schedule a disk operation to read the desired page into the newly allocated frame.

v. When disk read is complete, we modify the page table that, the page is now in memory.

vi. Restart the instruction that was interrupted by the trap. The process can now access the page as through it had always been in memory.



i.

j. **Pure Demand Paging**
    i. In extreme case, we can start executing a process with no pages in memory. When OS sets the instruction pointer to the first instruction of the process, which is not in the memory. The process immediately faults for the page and page is brought in the memory.
    ii. Never bring a page into memory until it is required.
k. We use **locality of reference** to bring out reasonable performance from demand paging.

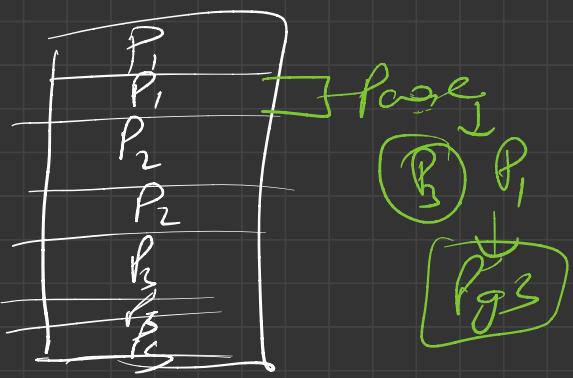
11. **Advantages of Virtual memory**
    a. The degree of multi-programming will be increased.
    b. User can run large apps with less real physical memory.
12. **Disadvantages of Virtual Memory**
    a. The system can become slower as swapping takes time.
    b. **Thrashing** may occur.

Lec-29

Page replacement algos :-

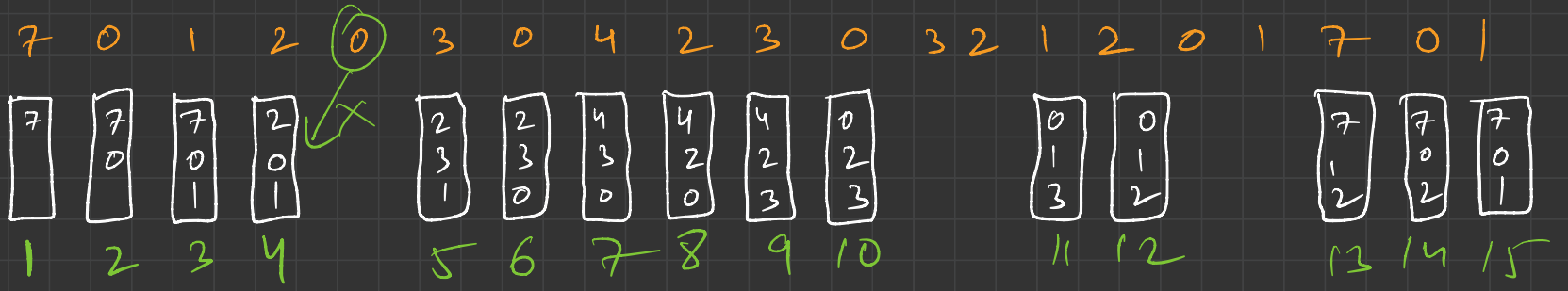
⇒ Page faults ??

Page fault service time



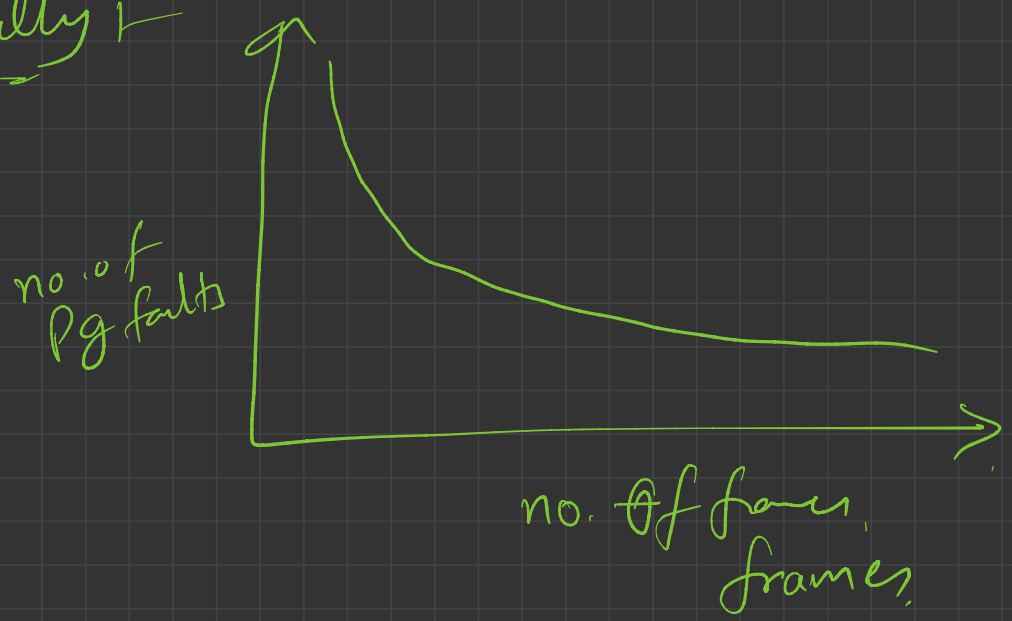
① FIFO :-

→ Ref string :-



FIFO → 15 page faults

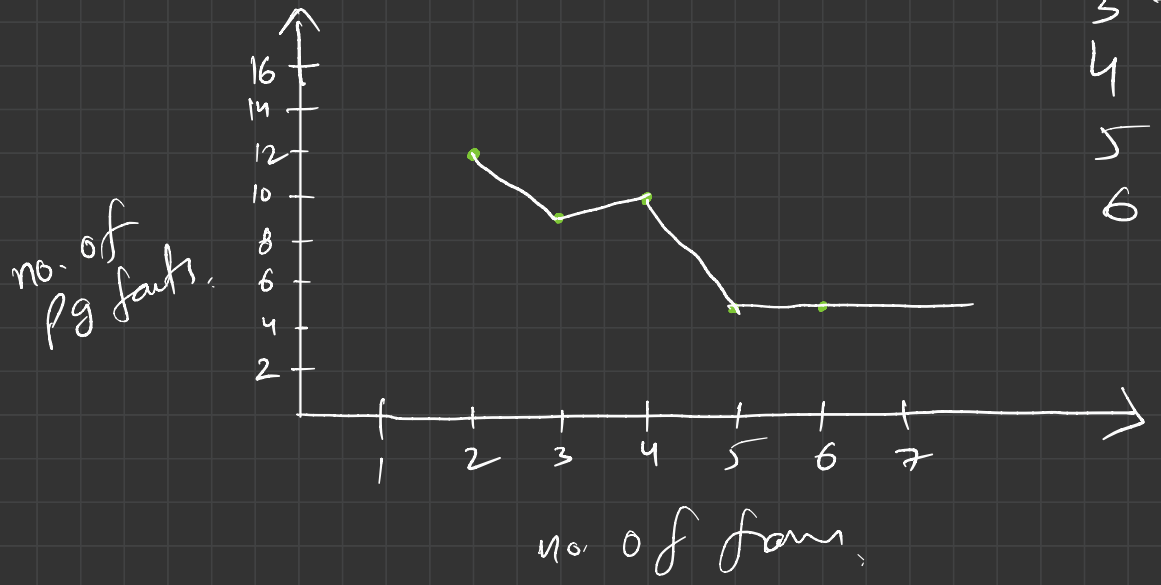
Normally ↓



Belady's anomaly in FIFO

Ref: 1 2 3 4 1 2 5 1 2 3 4 5

2 → 12
3 → 9
4 → 10
5 → 5
6 → 5



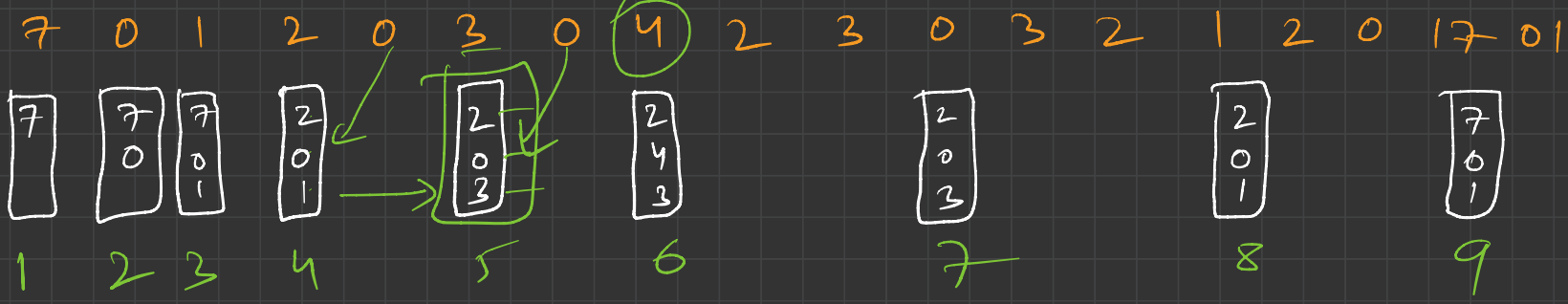
② Optimal Page Replacement

→ Best

→ pg. faults minimum

→ almost impossible to implement.

eg.



OPR → 9 page faults

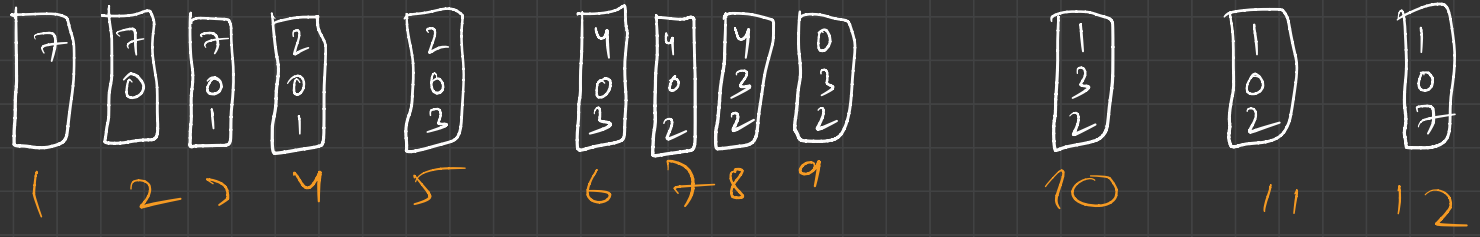
③ LRU Least Recently used

→ approximation past based on.

→

Q

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

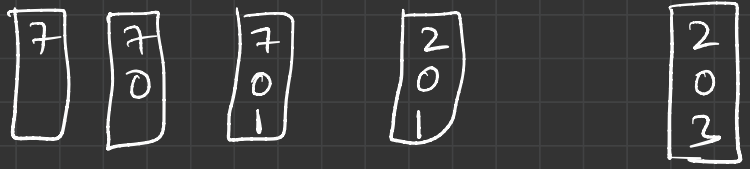


LRU → 12 Pg. faults

LRU implement

(1) Counter.

7 0 1 2 0 3 0 4 2 3



Count → universal

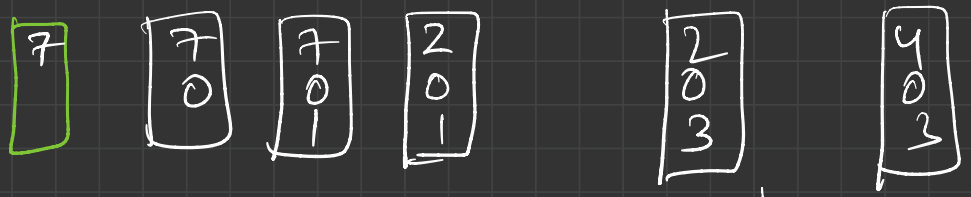
0	→ 5
1	→
2	→ 4
3	→ 6
4	→
7	→

Global shared count variable

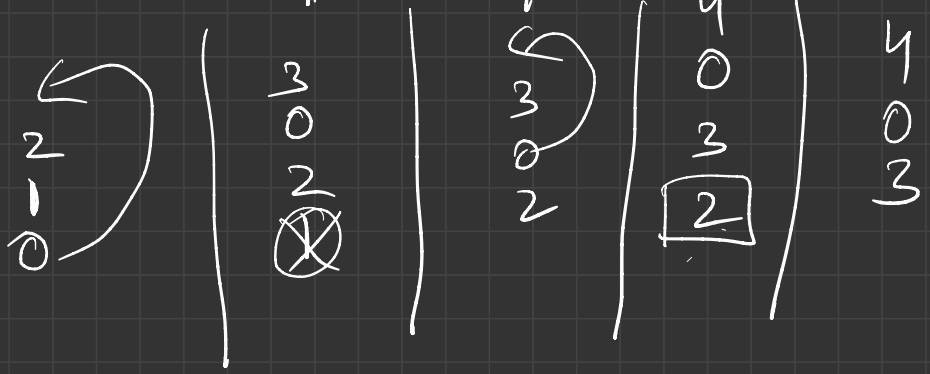
LRU implement

② stack based.

7 0 1 2 0 3 0 4 2 3 ——— _ _



Stack →



LEC-29: Page Replacement Algorithms

1. Whenever **Page Fault** occurs, that is, a process tries to access a page which is not currently present in a frame and OS must bring the page from swap-space to a frame.
2. OS must do page replacement to accommodate new page into a free frame, but there might be a possibility the system is working in high utilization and all the frames are busy, in that case OS must replace one of the pages allocated into some frame with the new page.
3. The **page replacement algorithm** decides which memory page is to be replaced. Some allocated page is swapped out from the frame and new page is swapped into the freed frame.
4. **Types** of Page Replacement Algorithm: (**AIM** is to have minimum page faults)
    a. **FIFO**
        i. Allocate frame to the page as it comes into the memory by **replacing the oldest page.**
        ii. Easy to implement.
        iii. Performance is **not** always good
            1. The page replaced may be an initialization module that was used long time ago **(Good replacement candidate)**
            2. The page may contain a heavily used variable that was initialized early and is in content use. (**Will again cause page fault)**
        iv. **Belady's** anomaly is present.
            1. **In the case of LRU and optimal page replacement algorithms, it is seen that the number of page faults will be reduced if we increase the number of frames.** However, Balady found that, In FIFO page replacement algorithm, the number of page faults will get increased with the increment in number of frames.
            2. This is the strange behavior shown by FIFO algorithm **in some of the cases.**
    b. **Optimal** page replacement
        i. Find if a page that is never referenced in future. If such a page exists, replace this page with new page.
        If no such page exists, find a page that is **referenced farthest in future**. Replace this page with new page.
        ii. **Lowest** page fault rate among any algorithm.
        iii. Difficult to implement as **OS requires future knowledge of reference string** which is kind of impossible. (Similar to SJF scheduling)
    c. Least-recently used (**LRU**)
        i. We can use recent past as an approximation of the near future then we can replace the page that has not been used for the longest period.
        ii. Can be implemented by two ways
            1. **Counters**
                a. Associate time field with each page table entry.
                b. Replace the page with smallest time value.
            2. **Stack**
                a. Keep a stack of page number.
                b. Whenever page is referenced, it is removed from the stack & put on the top.
                c. By this, most recently used is always on the top, & least recently used is always on the bottom.
                d. As entries might be removed from the middle of the stack, so Doubly linked list can be used.
    d. **Counting**-based page replacement – Keep a counter of the number of references that have been made to **each** page. (Reference counting)

i. Least frequently used (**LFU**)
    1. Actively used pages should have a large reference count.
    2. Replace page with the smallest count.
ii. Most frequently used (**MFU**)
    1. Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
iii. Neither MFU nor LFU replacement is common.

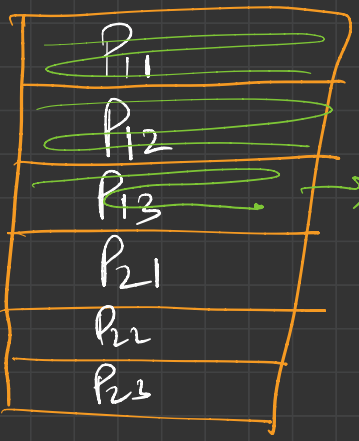
Lec-30

* Thrashing

Situation no. 1

$P_1 \rightarrow$ 3 page

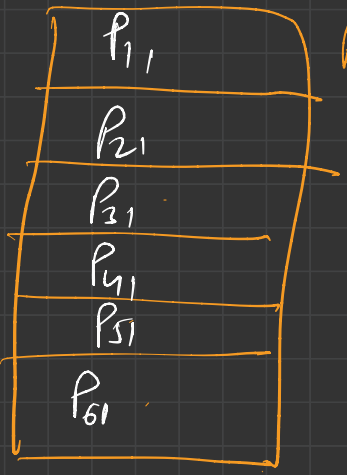
$P_2 \rightarrow$ 3 page



$\rightarrow P_{14} \rightarrow$ Req.

RAM

(2)



$P_1 - P_6 \rightarrow$ each has 1 page

RAM

→ Which has more no. of page faults?

ans → situation 2 → CPU will be more busy servicing page faults.

* Cause of Thrashing

① Initial low CPU utilization .... ↑ing degree of multiprograming

② A Global Pg. replacement algo, replaces pages w/o regard to the process

③ A process may need more frames .... cause faults again

④ Other processes's frames are replaced & they may need those soon.

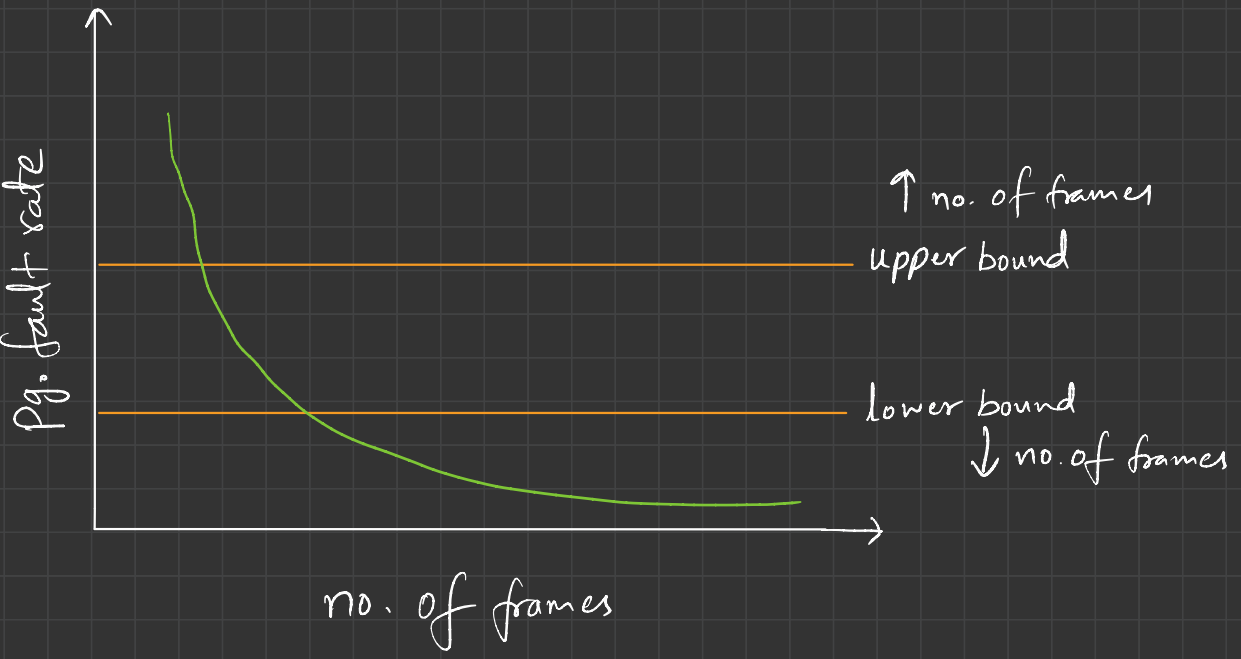
⑤ As a result CPU utilization ↓es.

⑥ CPU scheduler now, may increase CPU utilization by increasing degree of multiprograming

⑦ Ultimately, CPU utilization droops drastically.

* Thrashing → Avoid

* Page fault frequency



* Storage mgmt

⇒ interview ↓

* → ① self - learning

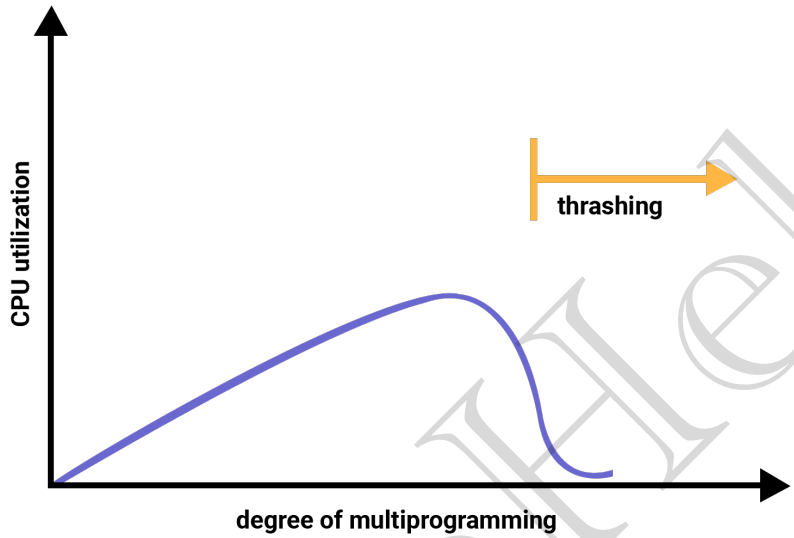
② Files & directories

* → OS course

⇒ Last video

LEC-30: Thrashing

1. **Thrashing**
    a. If the process doesn't have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
    b. This **high paging activity is called Thrashing.**
    c. A system is Thrashing when it **spends more time servicing the page faults than executing processes.**



    d. **Technique to Handle Thrashing**
        i. **Working set model**
            1. This model is based on the concept of the **Locality** Model.
            2. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever **it moves to some new locality.** But if the allocated frames are lesser than the size of the current locality, **the process is bound to thrash.**
        ii. **Page Fault frequency**
            1. **Thrashing** has a high page-fault rate.
            2. We want to **control** the page-fault rate.
            3. When it is too high, the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
            4. We establish upper and lower bounds on the desired page fault rate.
            5. If pf-rate exceeds the upper limit, allocate the process another frame, if pf-rate fails falls below the lower limit, remove a frame from the process.
            6. By controlling pf-rate, thrashing can be prevented.

Lec-31

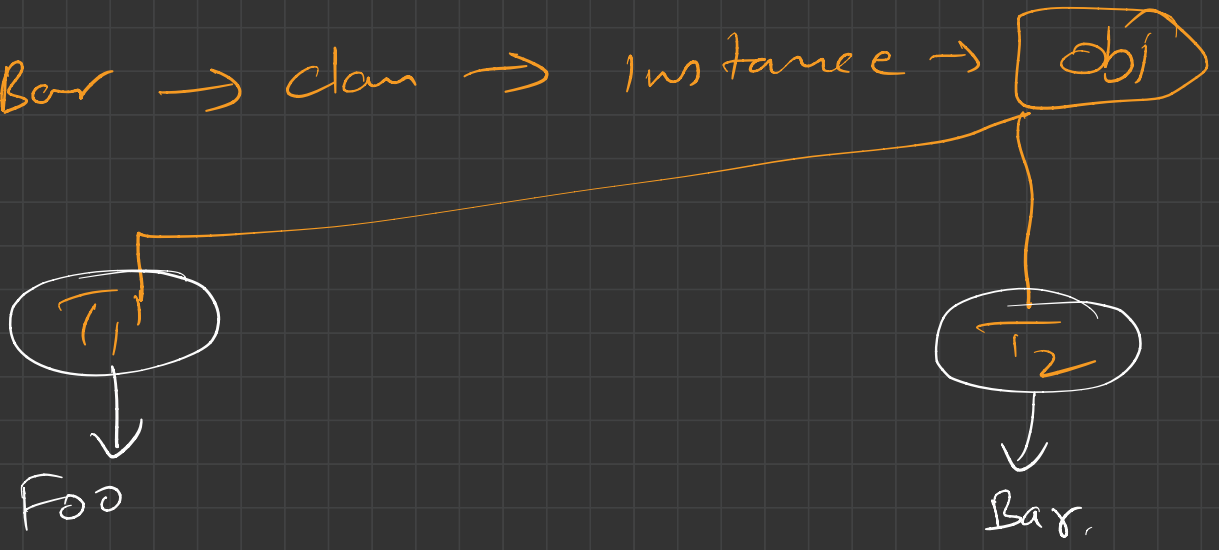
Leetcode concurrency

3

1. Print FooBar
2. Print zero Even odd
3. Building H2O ☆

① Print → FooBar

FooBar → class → instance → obj



Single Process / thread → Foo Bar Foo Bar.

Two threads → $T_1 \rightarrow$ $T_2 \rightarrow$ $T_1 \rightarrow$ $T_2 \rightarrow$

Code =

Watch Lec-23

```cpp
class FooBar {
private:
    int n;
    std:: mutex m;
    std::condition_variable cv;
    bool turn;
public:
    FooBar(int n) {
        this->n = n;
        turn = 0;
    }

    void foo(function<void()> printFoo) {

        for (int i = 0; i < n; i++) {
            std::unique_lock<std::mutex> lock(m);
            while(turn == 1){
                cv.wait(lock);
            }
            printFoo();
            turn = 1;
            cv.notify_all();
        }
    }

    void bar(function<void()> printBar) {

        for (int i = 0; i < n; i++) {
            std::unique_lock<std::mutex> lock(m);
            while(turn == 0){
                cv.wait(lock);
            }
            printBar();
            turn = 0;
            cv.notify_all();
        }
    }
};
```

Initially turn = 0

T_1

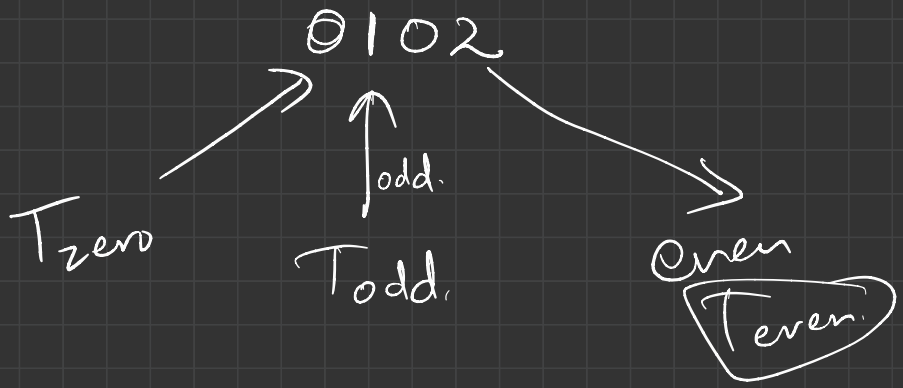
turn = 1

T_2

turn = 0

Problem 2 Print zero Even odd

=> n = 2



Solution 1

n = 3 → 0 1 0 2 0 3

T_zero T_odd T_even

n = 2

```
class ZeroEvenOdd {
private:
    int n;
    std:: mutex m;
    std::condition_variable cv;
    int turn;
    int i;
public:
    ZeroEvenOdd(int n) {
        this->n = n;
        turn = 0;
        i = 1;
    }

    // printNumber(x) outputs "x", where x is an inte
    void zero(function<void(int)> printNumber) {
        while(i<=n){
            std::unique_lock<std::mutex> lock(m);
            while(turn != 0 && i <= n){
                cv.wait(lock);
            }
            if(i>n){
                break;
            }
            printNumber(0);
            turn = (i % 2) == 0 ? 2 : 1;
            cv.notify_all();
        }
    }
```

Turn = 0

```
    void even(function<void(int)> printNumber) {
        while(i<=n){
            std::unique_lock<std::mutex> lock(m);
            while(turn != 2 && i <= n){
                cv.wait(lock);
            }
            if(i>n){
                break;
            }
            printNumber(i++);
            turn = 0;
            cv.notify_all();
        }
    }

    void odd(function<void(int)> printNumber) {
        while(i<=n){
            std::unique_lock<std::mutex> lock(m);
            while(turn != 1 && i <= n){
                cv.wait(lock);
            }
            if(i>n){
                break;
            }
            printNumber(i++);
            turn = 0;
            cv.notify_all();
        }
    }
};
```

i = 3 turn = 2

turn = 1

Case 1: n = 2

① i = 1 → turn = 1

② i = 2 → turn = 2

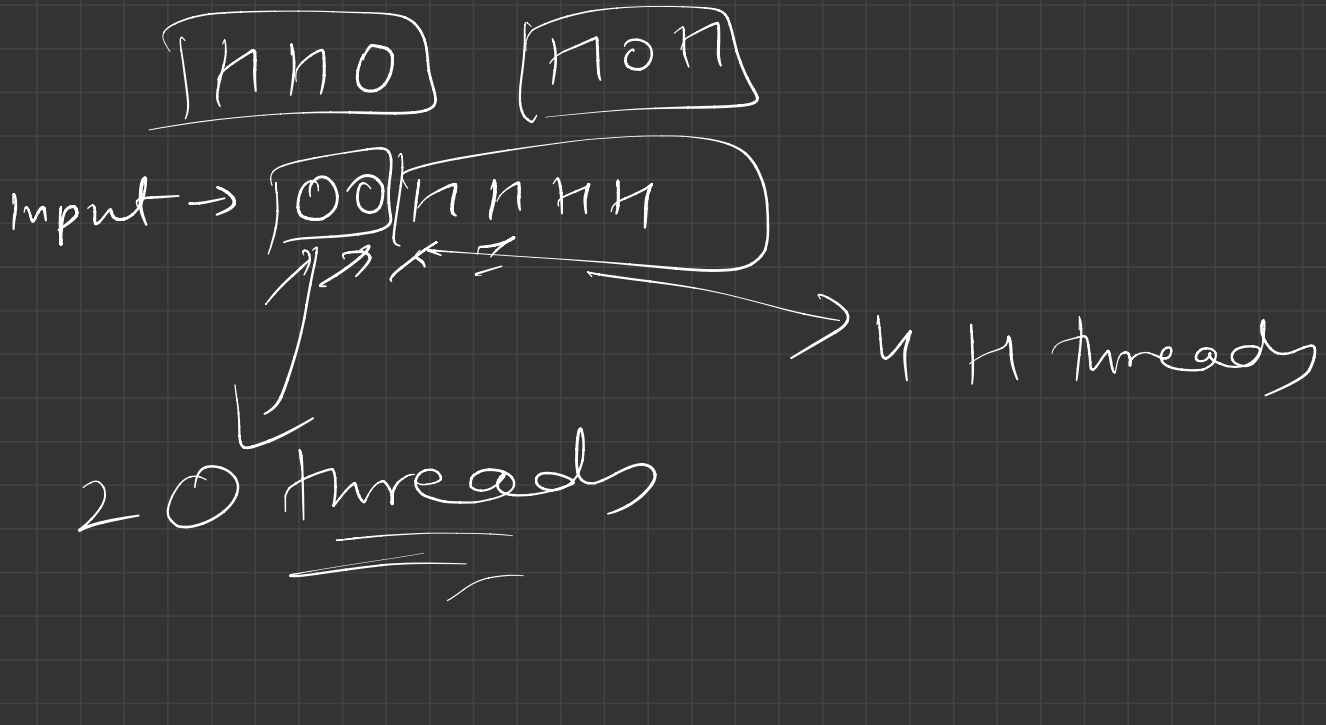
③ i = 3

output: 0102

(3) H2O

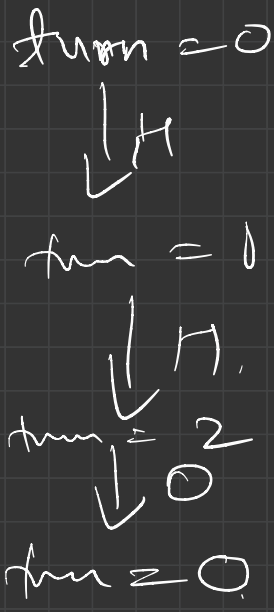
=> H → Threads multiple

O → Threads → multiple



solution:-

```
class H2O {
    std:: mutex m;
    std::condition_variable cv;
    int turn;
public:
    H2O() {
       turn = 0;
    }

    void hydrogen(function<void()> releaseHydrogen) {
        std::unique_lock<std::mutex> lock(m);
        while(turn == 2){
            cv.wait(lock);
        }
        releaseHydrogen();
        ++turn;
        cv.notify_all();
    }

    void oxygen(function<void()> releaseOxygen) {
        std::unique_lock<std::mutex> lock(m);
        while(turn < 2){
            cv.wait(lock);
        }
        releaseOxygen();
        turn = 0;
        cv.notify_all();
    }
};
```



Conclusion

turn → mutex conditions