

## 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.

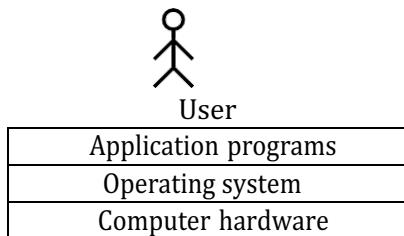
An **operating system** is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as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



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

## LEC-2: Types of OS

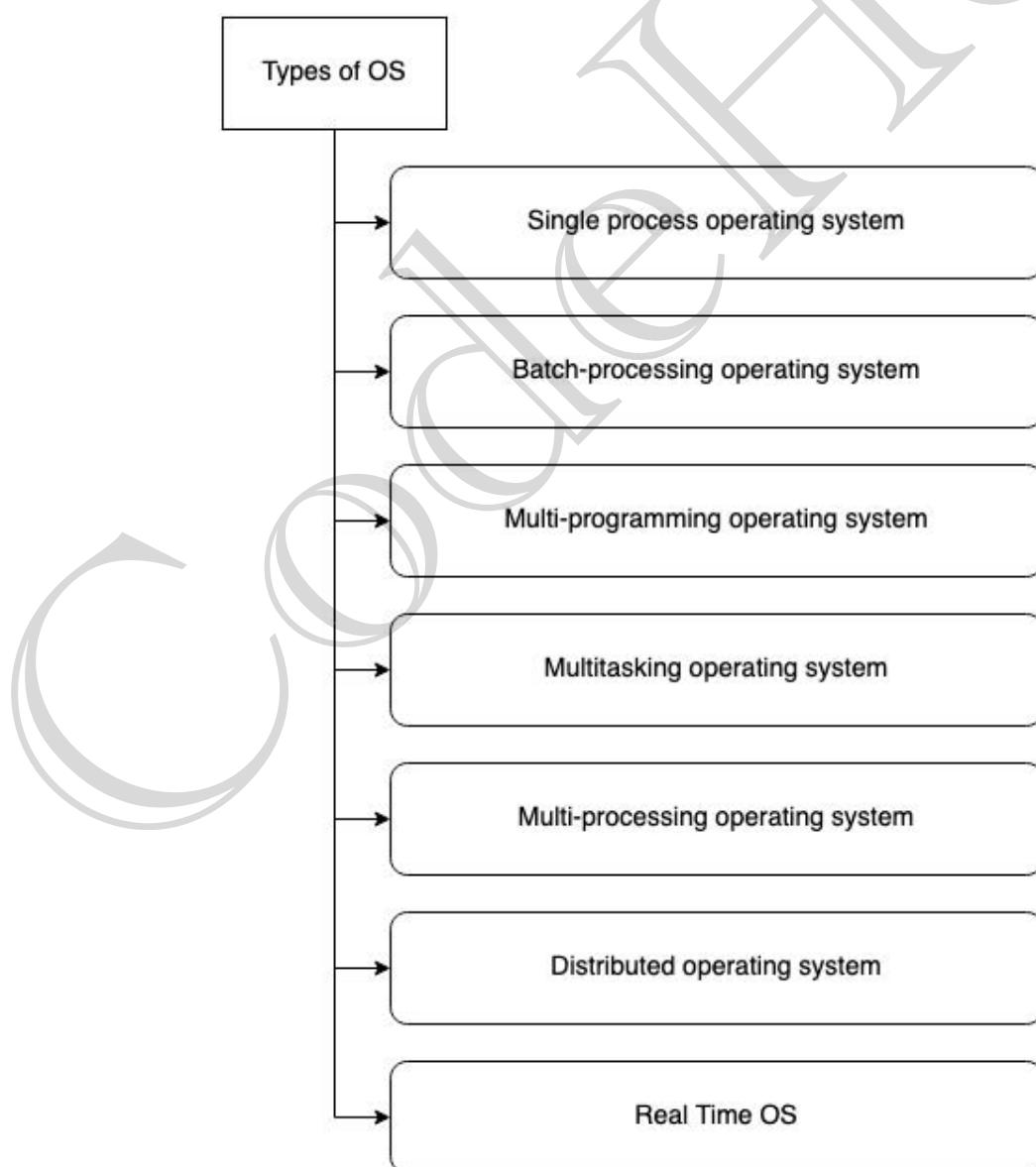


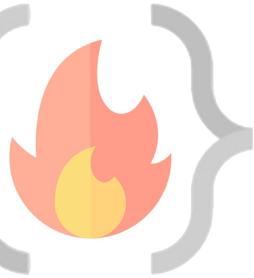
OS goals -

- 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]  |

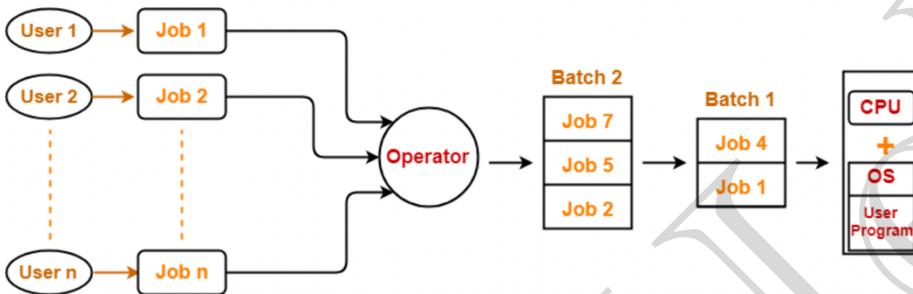




**Single process OS**, only 1 process executes at a time from the ready queue. [Oldest]

### 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.

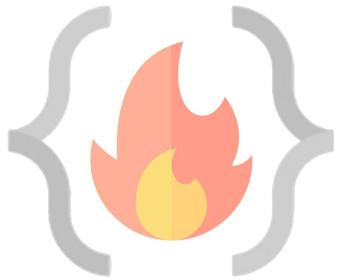
- Single CPU
- Context switching for processes.
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

**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,  $\geq 1$  CPUs,  $\geq 1$  memory,  $\geq 1$  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.

CodeHelp



### 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.)

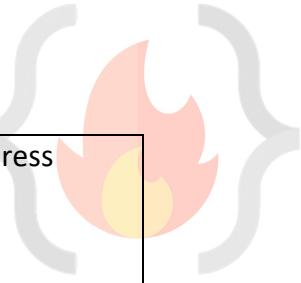
Multi-Tasking	Multi-Threading
The execution of more than one task simultaneously is called as multitasking.	A process is divided into several different sub-tasks called as threads, which has its own path of execution. This concept is called as multithreading.
Concept of more than 1 processes being context switched.	Concept of more than 1 thread. Threads are context switched.
No. of CPU 1.	No. of CPU $\geq 1$ . (Better to have more than 1)
<b>Isolation and memory protection</b> exists. OS must allocate separate memory and resources to each program that CPU is executing.	<b>No isolation and memory protection</b> , 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.

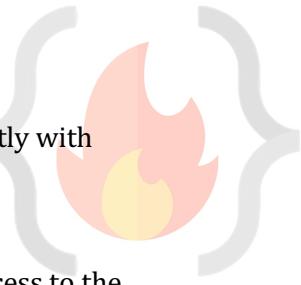
#### **Difference between Thread Context Switching and Process Context Switching:**

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 include 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.

CodeHelp



1. **Kernel:** A **kernel** is that part of the operating system which interacts directly with the hardware and performs 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

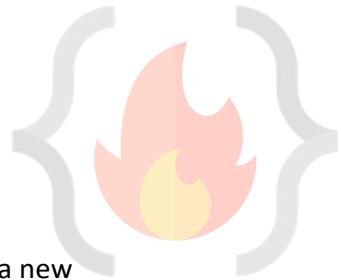
## 4. Nano/Exo kernels...

**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-5: System Calls



**How do apps interact with Kernel?** -> using system calls.

Eg. Mkdir laks

- Mkdir indirectly calls kernel and asked the file mgmt. module to create a new directory.
- Mkdir is just a wrapper of actual system calls.
- Mkdir interacts with kernel using system calls.

Eg. Creating a process.

- User executes a process. (User space)
- Gets system call. (US)
- Exec system call to create a process. (KS)
- Return to US.

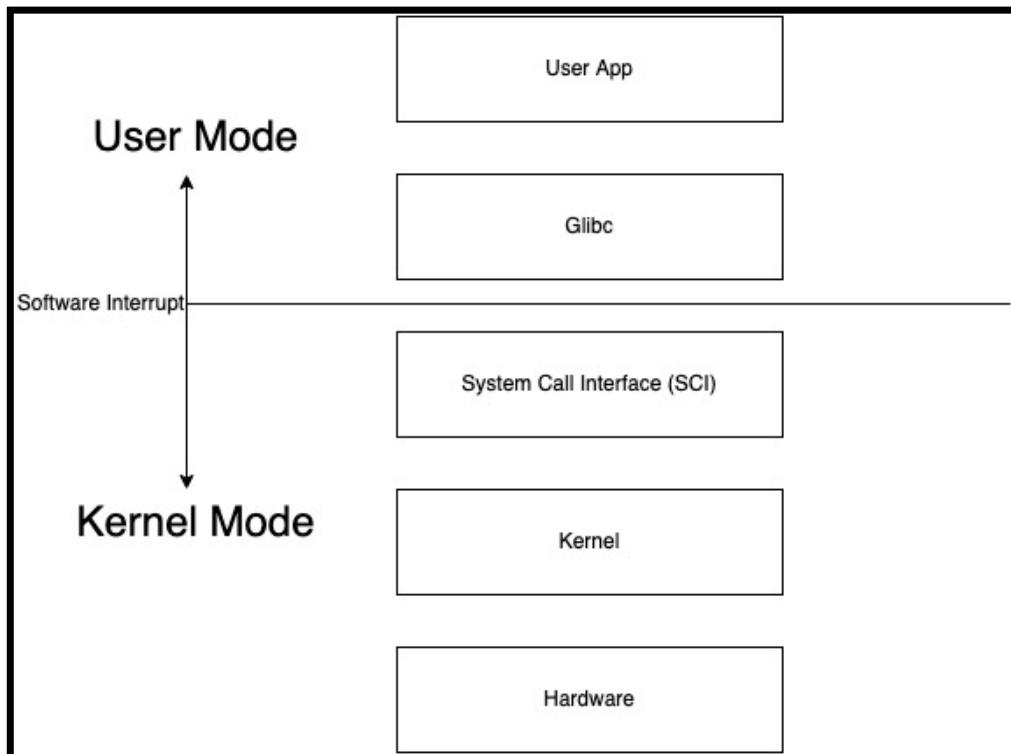
**Transitions from US to KS done by software interrupts.**

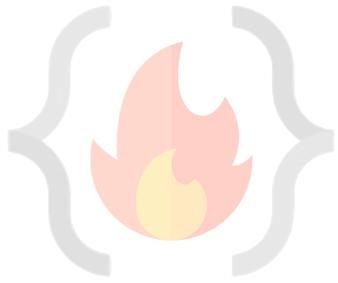
**System calls** are implemented in C.

A **system call** is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform.

User programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs.

**System Calls** are the only way through which a process can go into **kernel mode from user mode**.





### Types of System Calls:

- 1) Process Control
  - a. end, abort
  - b. load, execute
  - c. create process, terminate process
  - d. get process attributes, set process attributes
  - e. wait for time
  - f. wait event, signal event
  - g. allocate and free memory
- 2) File Management
  - a. create file, delete file
  - b. open, close
  - c. read, write, reposition
  - d. get file attributes, set file attributes
- 3) Device Management
  - a. request device, release device
  - b. read, write, reposition
  - c. get device attributes, set device attributes
  - d. logically attach or detach devices
- 4) Information maintenance
  - a. get time or date, set time or date
  - b. get system data, set system data
  - c. get process, file, or device attributes
  - d. set process, file, or device attributes
- 5) Communication Management
  - a. create, delete communication connection
  - b. send, receive messages
  - c. transfer status information
  - d. attach or detach remote devices

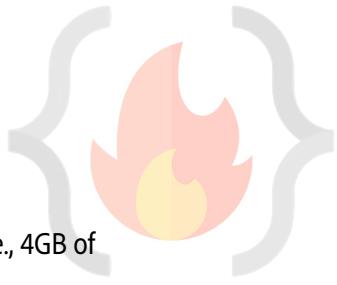
### Examples of Windows & Unix System calls:

Category	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle() SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	open () read () write () close () chmod() umask() chown()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Management	GetCurrentProcessID() SetTimer() Sleep()	getpid () alarm () sleep ()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe () shmget () mmap()

## LEC-6: What happens when you turn on your computer?

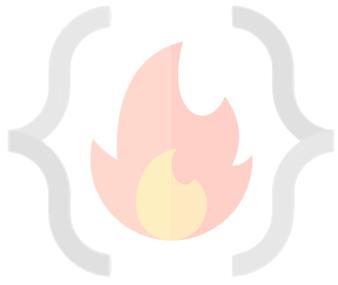


- i. PC On
- ii. CPU initializes itself and looks for a firmware program (BIOS) stored in BIOS Chip (Basic input-output system chip is a ROM chip found on mother board that allows to access & setup computer system at most basic level.)
  - 1. In modern PCs, CPU loads UEFI (Unified extensible firmware interface)
- iii. CPU runs the BIOS which tests and initializes system hardware. Bios loads configuration settings. If something is not appropriate (like missing RAM) error is thrown and boot process is stopped.  
This is called **POST** (Power on self-test) process.  
(UEFI can do a lot more than just initialize hardware; it's really a tiny operating system. For example, Intel CPUs have the [Intel Management Engine](#). This provides a variety of features, including powering Intel's Active Management Technology, which allows for remote management of business PCs.)
- iv. BIOS will handoff responsibility for booting your PC to your OS's bootloader.
  - 1. BIOS looked at the [MBR \(master boot record\)](#), a special boot sector at the beginning of a disk. The MBR contains code that loads the rest of the operating system, known as a "bootloader." The BIOS executes the bootloader, which takes it from there and begins booting the actual operating system—Windows or Linux, for example.  
In other words,  
the BIOS or UEFI examines a storage device on your system to look for a small program, either in the MBR or on an EFI system partition, and runs it.
- v. The bootloader is a small program that has the large task of booting the rest of the operating system (Boots Kernel then, User Space). Windows uses a bootloader named Windows Boot Manager (Bootmgr.exe), most Linux systems use [GRUB](#), and Macs use something called boot.efi



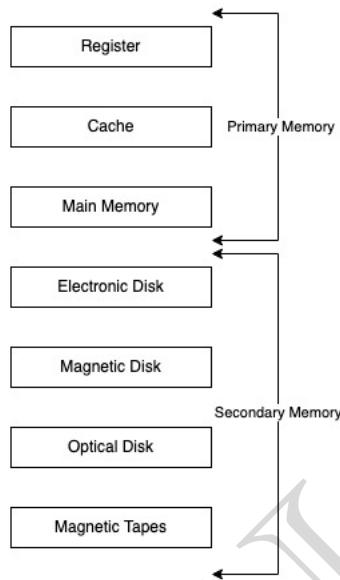
## Lec-7: 32-Bit vs 64-Bit OS

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 billions 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

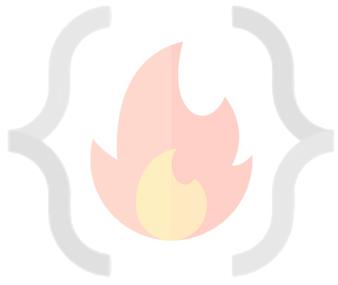
What are the different memory present in the computer system?



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.

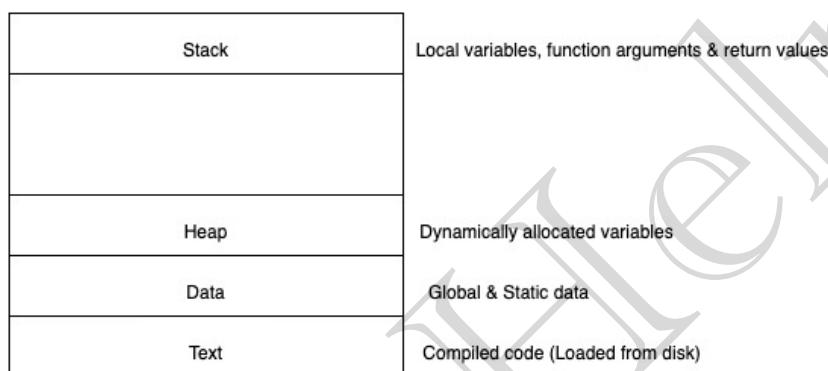


## Lec-9: Introduction to Process

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:**



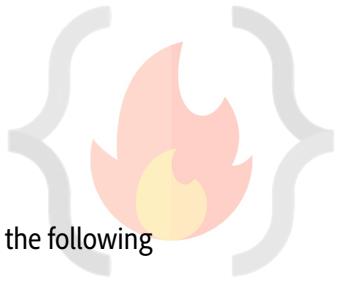
5. **Attributes of process:**

- 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:**

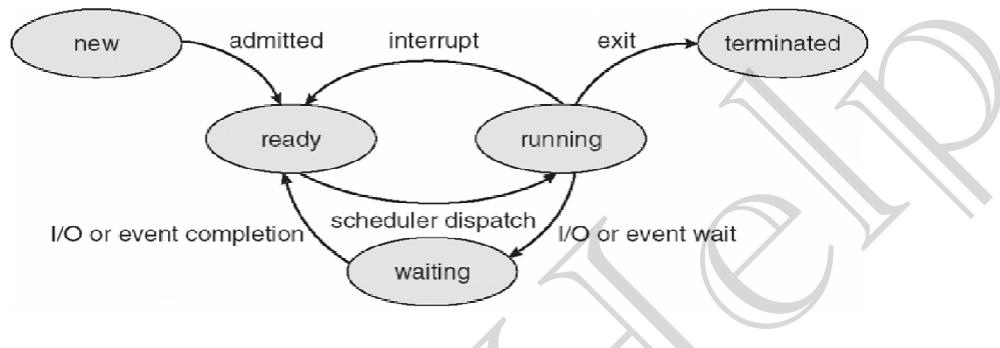
Process ID	Unique identifier
Program Counter (PC)	Next instruction address of the program
Process State	Stores process state
Priority	Based on priority a process gets CPU time
Registers	
List of open files	
List of open devices	

**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.

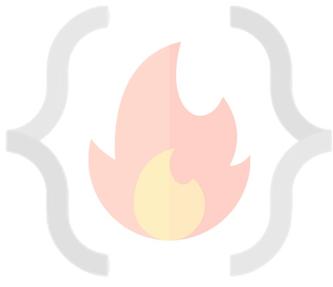


## Lec-10: Process States | Process Queues

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.

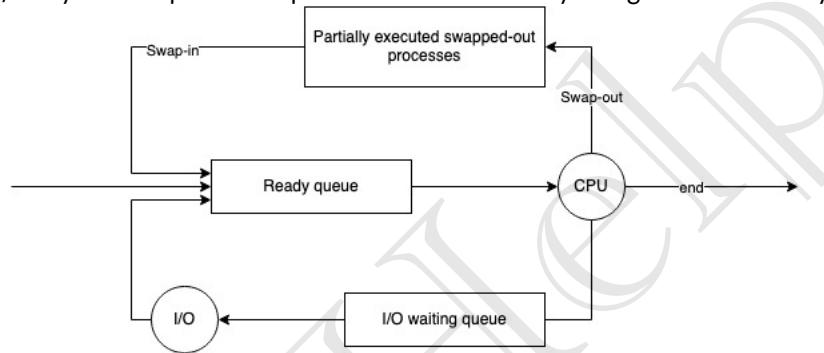


2. **Process Queues:**
  - a. **Job Queue:**
    - i. Processes in new state.
    - ii. Present in secondary memory.
    - iii. **Job Scheduler (Long term scheduler (LTS))** picks process from the pool and loads them into memory for execution.
  - b. **Ready Queue:**
    - i. Processes in Ready state.
    - ii. Present in main memory.
    - iii. **CPU Scheduler (Short-term scheduler)** picks process from ready queue and dispatches 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**.



### 1. Swapping

- a. Time-sharing system may have medium term scheduler (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

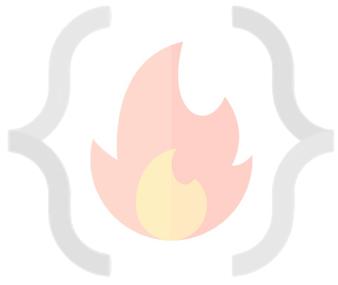
- a. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
- b. When this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- c. It is pure 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-12: Intro to Process Scheduling | FCFS | Convoy Effect

### 1. Process Scheduling

- a. Basis of Multi-programming OS.
- b. By switching the CPU among processes, the OS can make the computer more productive.
- c. Many processes are kept in memory at a time, when a process must wait or time quantum expires, the OS takes the CPU away from that process & gives the CPU to another process & this pattern continues.

### 2. CPU Scheduler

- a. Whenever the CPU becomes idle, OS must select one process from the ready queue to be executed.
- b. Done by STS.

### 3. Non-Preemptive scheduling

- a. Once CPU has been allocated to a process, the process keeps the CPU until it releases CPU either by terminating or by switching to wait-state.
- b. Starvation, as a process with long burst time may starve less burst time process.
- c. Low CPU utilization.

### 4. Preemptive scheduling

- a. CPU is taken away from a process after time quantum expires along with terminating or switching to wait-state.
- b. Less Starvation
- c. High CPU utilization.

### 5. Goals of CPU scheduling

- a. Maximum CPU utilization
- b. Minimum Turnaround time (TAT).
- c. Min. Wait-time
- d. Min. response time.
- e. Max. throughput of system.

6. **Throughput:** No. of processes completed per unit time.

7. **Arrival time (AT):** Time when process is arrived at the ready queue.

8. **Burst time (BT):** The time required by the process for its execution.

9. **Turnaround time (TAT):** Time taken from first time process enters ready state till it terminates. ( $CT - AT$ )

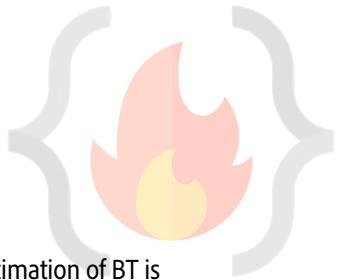
10. **Wait time (WT):** Time process spends waiting for CPU. ( $WT = TAT - BT$ )

11. **Response time:** Time duration between process getting into ready queue and process getting CPU for the first time.

12. **Completion Time (CT):** Time taken till process gets terminated.

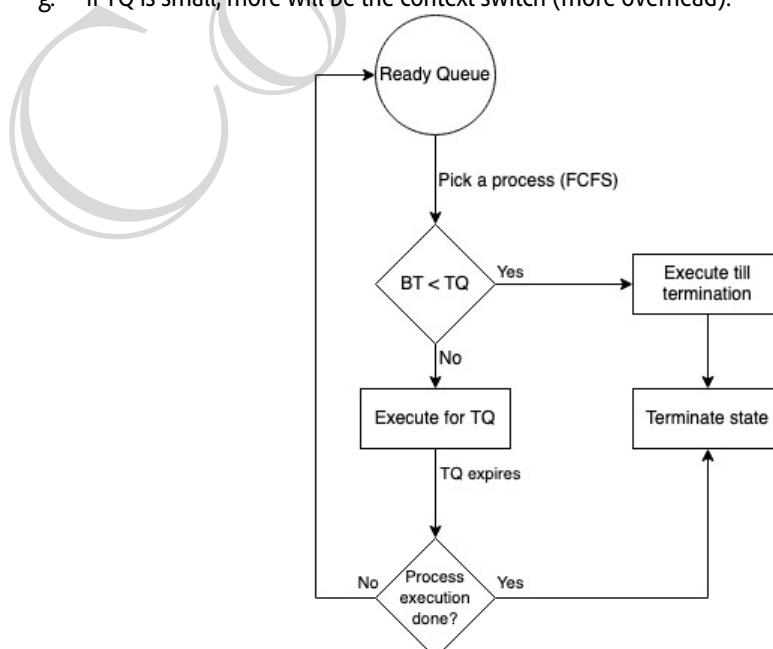
13. **FCFS (First come-first serve):**

- a. Whichever process comes first in the ready queue will be given CPU first.
- b. In this, if one process has longer BT. It will have major effect on average WT of diff processes, called **Convoy effect**.
- c. Convoy Effect is a situation where many processes, who need to use a resource for a short time, are blocked by one process holding that resource for a long time.
  - i. This causes poor resource management.



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

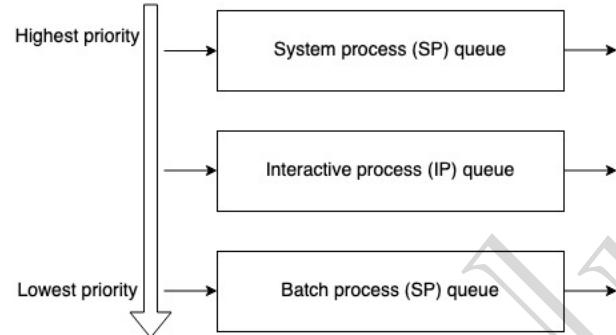
1. Shortest Job First (SJF) [Non-preemptive]
  - a. Process with least BT will be dispatched to CPU first.
  - b. Must do estimation for BT for each process in ready queue beforehand, Correct estimation of BT is an impossible task (ideally.)
  - c. Run lowest time process for all time then, choose job 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: 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.

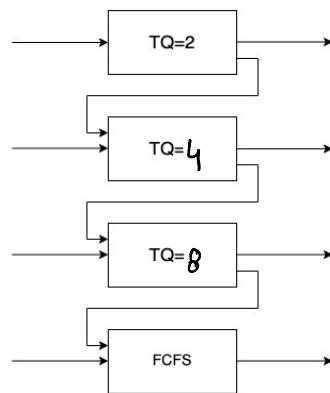
### 2. Multi-level feedback queue scheduling (MLFQ)

- 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.

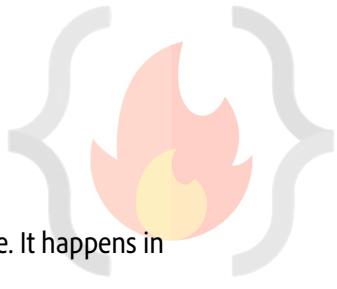
- c. Less starvation than MLQ.
- d. It is flexible.
- e. Can be configured to match a specific system design requirement.

Sample MLFQ design:



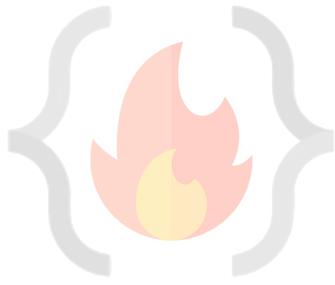
### 3. Comparison:

	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: 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 include 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 gets access to the CPU?**
  - Each thread has its own program counter.
  - Depending upon the thread scheduling algorithm, OS schedules 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 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: 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: 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 can 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 - 18

\* Producer - Consumer Problem

( Bonded Buffer Problem )

① Producer thread

② Consumer thread.

Producer



n slots

Consumer pick

Problem???

→ C.S (Buffer) → Sync. b/w Producer & consumer.

→ Producer must not insert data when the buffer is full.

→ Consumer must not pick / remove data when the buffer is empty.

## Sol<sup>n</sup>      Semaphores

- ① m, mutex → Binary sema. used to acquire lock on buffer.
- ② empty → a counting sem.  
Initial value is n  
↳ tracks empty slots.
- ③ full → tracks filled slots.  
— initial = 0.

## Producer

```
do {  
    wait(empty); // wait until  
    empty > 0  
    then, empty → value  
    wait(mutex);  
    // C.S, add data to buffer  
    signal(mutex);  
    signal(full); // increment full  
    → value,)  
} while (1)
```

## Consumer

```
do {  
    wait(full); // wait until  
    full > 0, then fill --;  
    wait(mutex);  
    // remove data from Buffer  
    signal(mutex);  
    signal(empty); // increment  
    empty,  
} while (1)
```

---

---

---

---

---



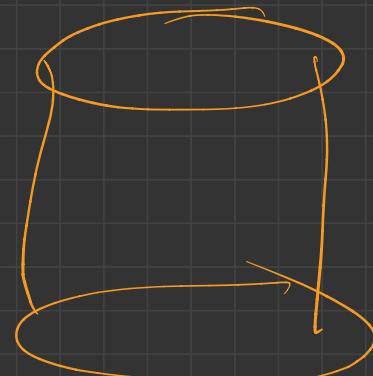
# Lec-19

## Reader writer problem

① Reader thread → Read

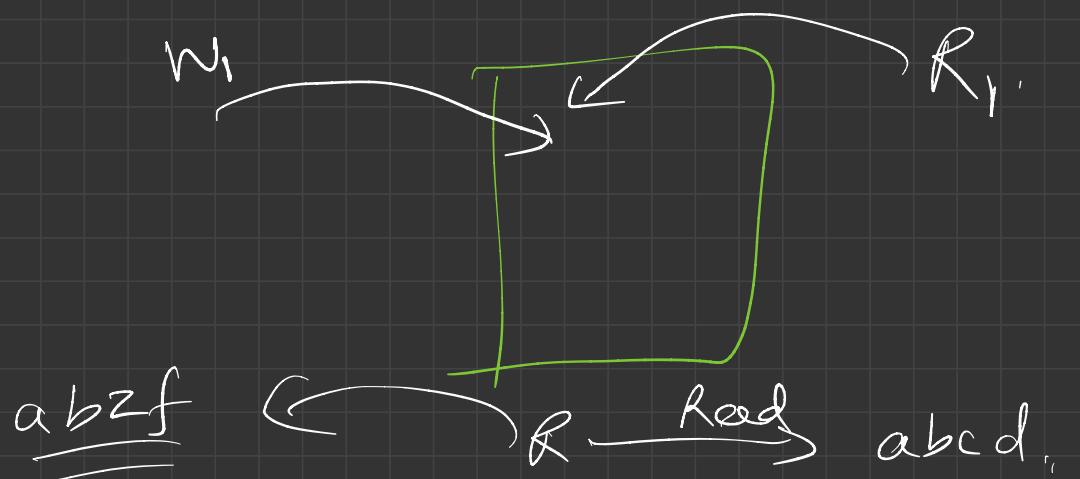
② writer threads → write  
update

① if  $> 1$  Readers are reading  
→ No issue ✕



②

> 1 writers OR 1 writer & some other  
thread (R/w), parallel,  
→ Race conditions  
& data inconsistent



# Semaphores sol<sup>n</sup> to R-w problems

①

mutex  $\rightarrow$  Binary semaphore

— to ensure mutual exclusion, when  
read count ( $rc$ ) is updated.

— No. two threads modify R.C at  
same time.

②

wrt  $\rightarrow$  Binary semaphore.

— common for both reader, writer.

③

read count ( $rc$ )  $\rightarrow$  integer 203,

tracks how many readers are  
reading in C.S



## Reader sol<sup>n</sup>

do {

```
    wait ( mutex ); // to mutex readcount variable  
    xc++;
```

```
    if ( xc == 1 )
```

```
        wait ( wrt ); // ensures no writer can enter if there is  
                      // even one reader
```

```
    signal ( mutex );
```

// C.S : Reader is reading

```
    wait ( mutex )
```

```
    xc--; // a reader leaves
```

```
    if ( xc == 0 ) // no reader is left in CS
```

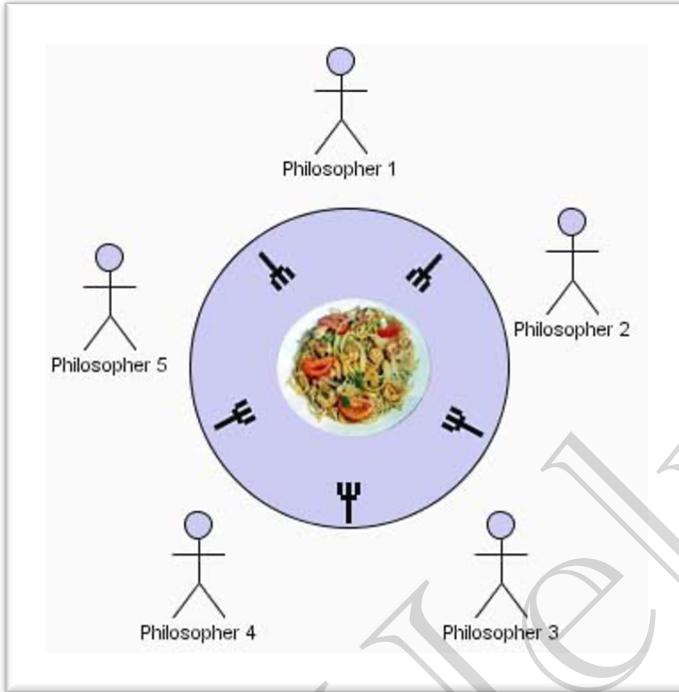
```
        signal ( wrt ); // writer can enter
```

```
        signal ( mutex ); // reader leaves
```

```
} while ( 1 )
```



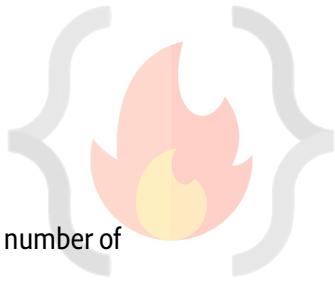
## Lec-20: The Dining Philosophers problem



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).

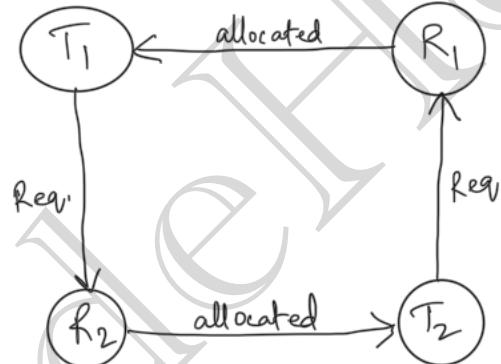
- c. **Odd-even rule.**  
an odd ph. Picks up first his left fork and then his right fork, whereas an even ph. Picks up his right fork then his left fork.
- 13. Hence, only semaphores are not enough to solve this problem.  
We must add some enhancement rules to make deadlock free solution.

CodeHelp

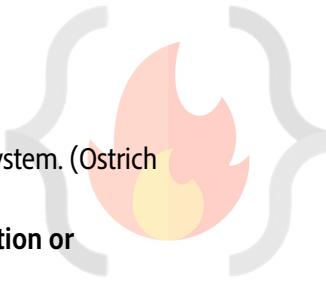
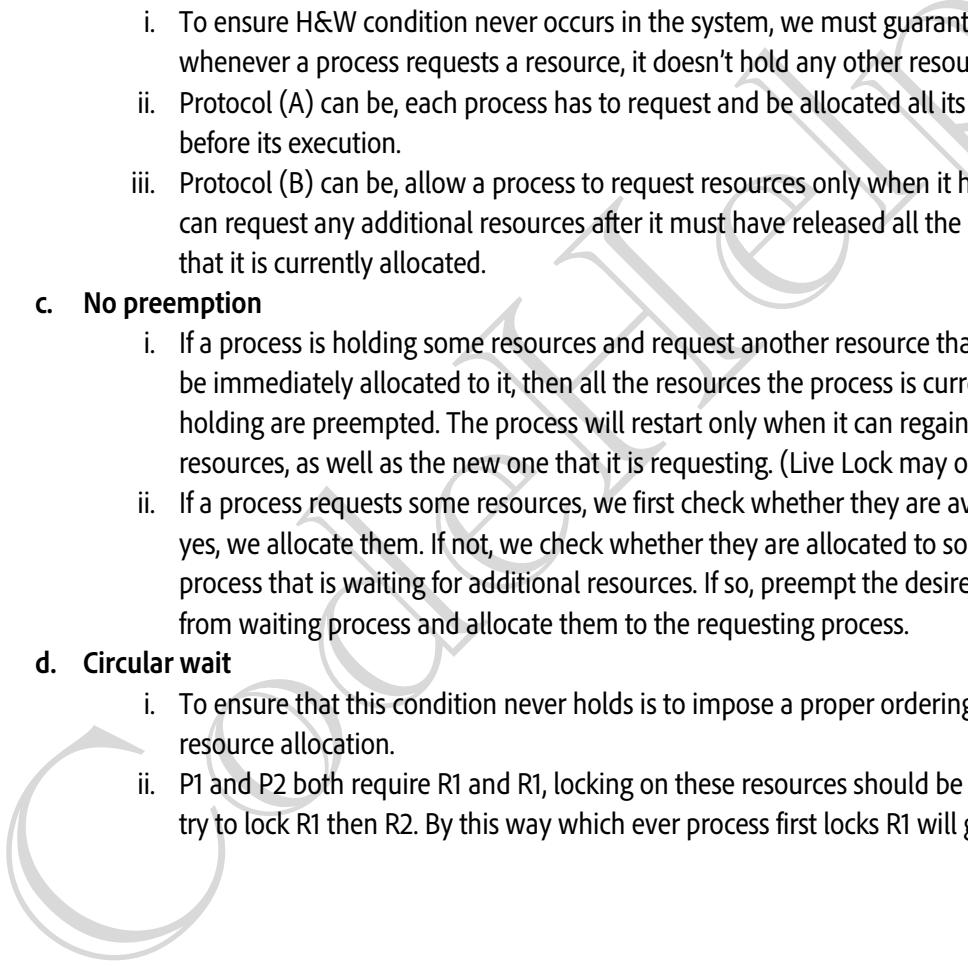


## LEC-21: Deadlock Part-1

1. In Multi-programming environment, we have several processes competing for finite number of resources
2. Process requests a **resource (R)**, if R is not available (taken by other process), process enters in a waiting state. Sometimes that waiting process is never able to change its state because the resource, it has requested is busy (forever), called **DEADLOCK (DL)**
3. Two or more processes are waiting on some resource's availability, which will never be available as it is also busy with some other process. The Processes are said to be in **Deadlock**.
4. DL is a bug present in the process/thread synchronization method.
5. In DL, processes never finish executing, and the system resources are tied up, preventing other jobs from starting.
6. **Example of resources:** Memory space, CPU cycles, files, locks, sockets, IO devices etc.
7. Single resource can have multiple instances of that. E.g., CPU is a resource, and a system can have 2 CPUs.
8. How a process/thread utilize a resource?
  - a. Request: Request the R, if R is free Lock it, else wait till it is available.
  - b. Use
  - c. Release: Release resource instance and make it available for other processes



9. **Deadlock Necessary Condition:** 4 Condition should hold simultaneously.
  - a. **Mutual Exclusion**
    - i. Only 1 process at a time can use the resource, if another process requests that resource, the requesting process must wait until the resource has been released.
  - b. **Hold & Wait**
    - i. A process must be holding at least one resource & waiting to acquire additional resources that are currently being held by other processes.
  - c. **No-preemption**
    - i. Resource must be voluntarily released by the process after completion of execution. (No resource preemption)
  - d. **Circular wait**
    - i. A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, and so on.
10. **Methods for handling Deadlocks:**
  - a. Use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlocked state.
  - b. Allow the system to enter a deadlocked state, **detect it, and recover**.

- 
- 
- c. Ignore the problem altogether and pretend that deadlocks never occur in system. (Ostrich algorithm) aka, **Deadlock ignorance**.
  - 11. To ensure that deadlocks never occur, the system can use either a **deadlock prevention** or **deadlock avoidance scheme**.
  - 12. **Deadlock Prevention:** by ensuring at least one of the necessary conditions cannot hold.
    - a. **Mutual exclusion**
      - i. Use locks (mutual exclusion) only for non-sharable resource.
      - ii. Sharable resources like Read-Only files can be accessed by multiple processes/threads.
      - iii. However, we can't prevent DLs by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.
    - b. **Hold & Wait**
      - i. To ensure H&W condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it doesn't hold any other resource.
      - ii. Protocol (A) can be, each process has to request and be allocated all its resources before its execution.
      - iii. Protocol (B) can be, allow a process to request resources only when it has none. It can request any additional resources after it must have released all the resources that it is currently allocated.
    - c. **No preemption**
      - i. If a process is holding some resources and request another resource that cannot be immediately allocated to it, then all the resources the process is currently holding are preempted. The process will restart only when it can regain its old resources, as well as the new one that it is requesting. (Live Lock may occur).
      - ii. If a process requests some resources, we first check whether they are available. If yes, we allocate them. If not, we check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resource from waiting process and allocate them to the requesting process.
    - d. **Circular wait**
      - i. To ensure that this condition never holds is to impose a proper ordering of resource allocation.
      - ii. P1 and P2 both require R1 and R2, locking on these resources should be like, both try to lock R1 then R2. By this way which ever process first locks R1 will get R2.



## 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.

---

---

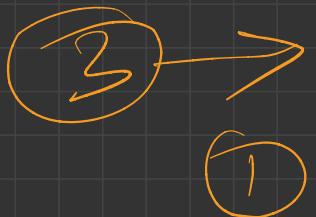
---

---

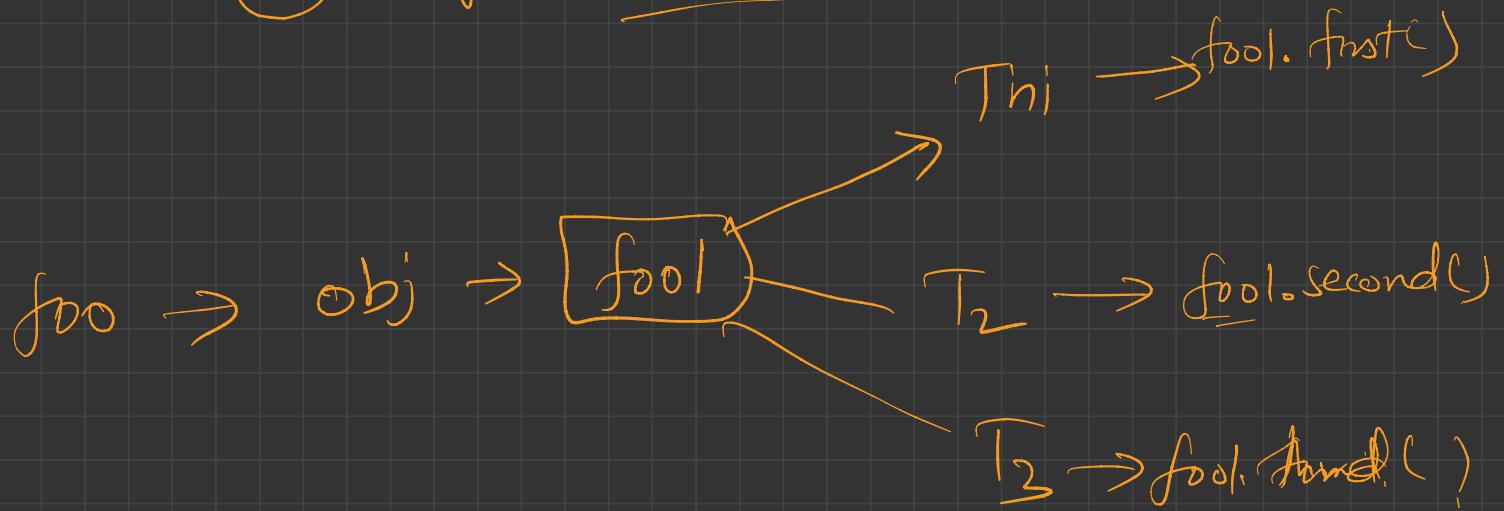
---



# Lec-23



Point Order



Locks & Condition variable

- ① first  $\rightarrow$  printfirst()  $\rightarrow$  T<sub>1</sub> execute.
- ② when first is printed  $\rightarrow$  T<sub>2</sub> execute.

```
1 class Foo {
2     std::mutex m;
3     std::condition_variable cv;
4     int turn;
5 public:
6     Foo() {
7         turn = 0;
8     }
9
10    void first(function<void()> printFirst) {
11
12        // printFirst() outputs "first". Do not change or remove this line.
13        printFirst();
14        turn = 1;
15        cv.notify_all();
16    }
17
18    void second(function<void()> printSecond) {
19        std::unique_lock<std::mutex> lock(m);
20        while(turn != 1){
21            cv.wait(lock);
22        }
23        // printSecond() outputs "second". Do not change or remove this line.
24        printSecond();
25        turn = 2;
26        cv.notify_all();
27    }
28
29    void third(function<void()> printThird) {
30        std::unique_lock<std::mutex> lock(m);
31        while(turn != 2){
32            cv.wait(lock);
33        }
34        // printThird() outputs "third". Do not change or remove this line.
35        printThird();
36    }
37};
```

T<sub>1</sub>

→

// printFirst() outputs "first". Do not change or remove this line.

T<sub>2</sub>

→

// printSecond() outputs "second". Do not change or remove this line.

T<sub>3</sub>

→

// printThird() outputs "third". Do not change or remove this line.

decide turn

② → Fizz buzz problem

```
1 class FizzBuzz {
2     private:
3         int n;
4         mutex m; // mutex
5         condition_variable c; // condition_variable
6         int i; // integer
7     public:
8         FizzBuzz(int n) {
9             this->n = n;
10            this->i = 1; // i = 1
11        }
12
13        // printFizz() outputs "fizz".
14        void fizz(function<void()> printFizz) {
15            while(i<=n){
16                {
17                    unique_lock<mutex> lock(m);
18                    while(i<=n && ((i%3) == 0) && ((i%5) != 0)) == 0) {
19                        c.wait(lock);
20                    }
21                    if(i<=n){
22                        printFizz();
23                        ++i;
24                    }
25                    c.notify_all();
26                }
27            }
28
29        // printBuzz() outputs "buzz".
30        void buzz(function<void()> printBuzz) {
31            while(i<=n){
32                {
33                    unique_lock<mutex> lock(m);
34                    while(i<=n && ((i%5) == 0) && ((i%3) != 0)) == 0) {
35                        c.wait(lock);
36                    }
37                    if(i<=n){
38                        printBuzz();
39                        ++i;
40                    }
41                    c.notify_all();
42                }
43            }
44
45        // printFizzBuzz() outputs "fizzbuzz".
46        void fizzbuzz(function<void()> printFizzBuzz) {
47            while(i<=n){
48                {
49                    unique_lock<mutex> lock(m);
50                    while(i<=n && ((i%5) == 0) && ((i%3) == 0)) == 0) {
51                        c.wait(lock);
52                    }
53                    if(i<=n){
54                        printFizzBuzz();
55                        ++i;
56                    }
57                    c.notify_all();
58                }
59            }
60
61        }
62
63        // printNumber(x) outputs "x", where x is an integer.
64        void number(function<void(int)> printNumber) {
65            while(i<=n){
66                {
67                    unique_lock<mutex> lock(m);
68                    while(i<=n && ((i%5) != 0) && ((i%3) != 0)) == 0) {
69                        c.wait(lock);
70                    }
71                    if(i<=n){
72                        printNumber(i++);
73                    }
74                    c.notify_all();
75                }
76            }
77        }
78    }
79 }
```

A →

B →

C →

D →

③

# Dining philosopher's problem

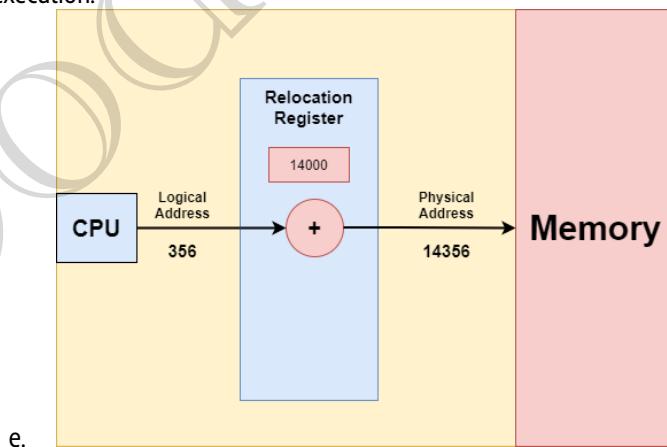
Video no - 20

```
1 class Semaphore
2 {
3     public:
4         Semaphore() {}
5         Semaphore(int c) : count(c){};
6
7         void setCount(int a)
8         {
9             count = a;
10        }
11
12        inline void signal()
13        {
14            std::unique_lock<std::mutex> lock(mtx);
15            count++;
16            if(count <= 0)
17                cv.notify_one();
18        }
19
20        inline void wait()
21        {
22            std::unique_lock<std::mutex> lock(mtx);
23            count--;
24            while (count < 0)
25            {
26                cv.wait(lock);
27            }
28        }
29
30    private:
31        std::mutex mtx;
32        std::condition_variable cv;
33        int count;
34    };
35
36    class DiningPhilosophers {
37        Semaphore fork[5];
38        std::mutex m, l;
39    public:
40        DiningPhilosophers() {
41            for (int i = 0; i < 5; ++i) {
42                fork[i].setCount(1);
43            }
44        }
45
46        void wantsToEat(int philosopher,
47                        function<void()> pickLeftFork,
48                        function<void()> pickRightFork,
49                        function<void()> eat,
50                        function<void()> putLeftFork,
51                        function<void()> putRightFork) {
52            std::lock_guard<std::mutex> lock(m);
53            fork[(philosopher + 1) % 5].wait();
54            fork[philosopher].wait();
55        }
56        pickLeftFork();
57        pickRightFork();
58
59        eat();
60
61        putLeftFork();
62        fork[(philosopher + 1) % 5].signal();
63        putRightFork();
64        fork[philosopher].signal();
65    }
66
67 };
```

## LEC-24: Memory Management Techniques | Contiguous Memory Allocation



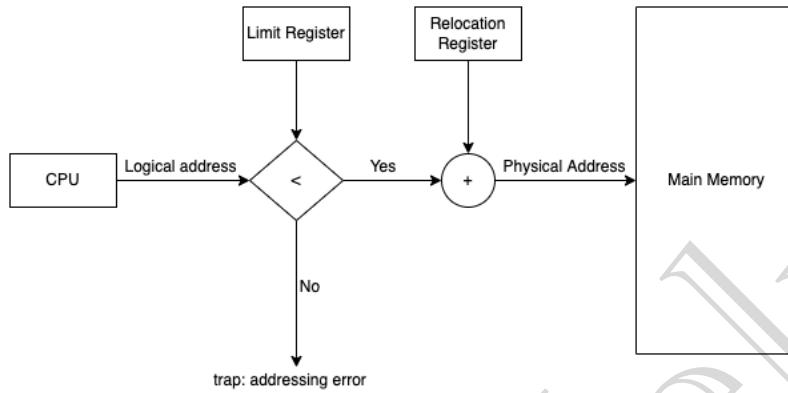
1. In Multi-programming environment, we have multiple processes in the main memory (Ready Queue) to keep the CPU utilization high and to make computer responsive to the users.
2. To realize this increase in performance, however, we must keep several processes in the memory; that is, we must **share** 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.



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 users' memory results in a trap in the OS, which treat the attempt as a fatal error.
- h. **Address Translation**



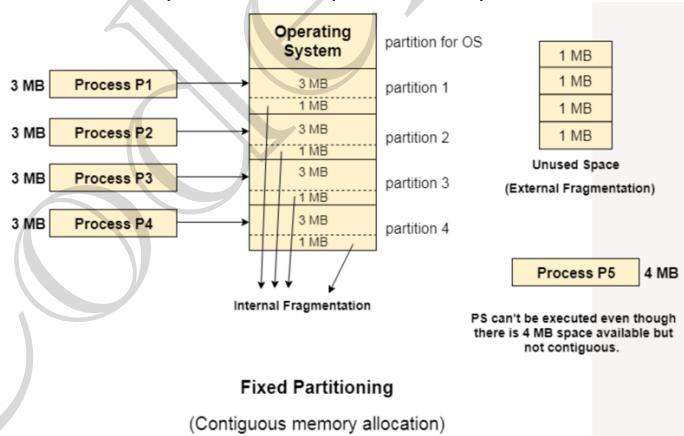
## 5. Allocation Method on Physical Memory

- 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:

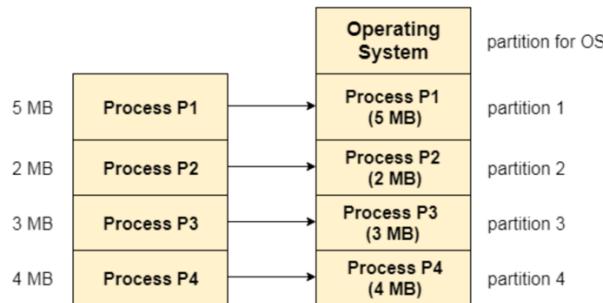
1. **Internal Fragmentation:** if the size of the process is lesser then the total size of the partition then some size of the partition gets wasted and remain unused. This is wastage of the memory and called internal fragmentation.
2. **External Fragmentation:** The total unused space of various partitions 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

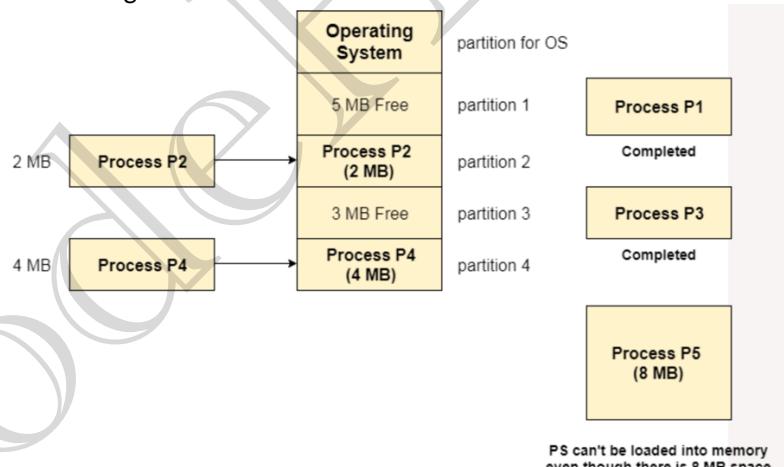
- i. In this technique, the partition size is not declared initially. It is declared at the time of process loading.



**Dynamic Partitioning**

(Process Size = Partition Size)

- ii. 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**



## 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**

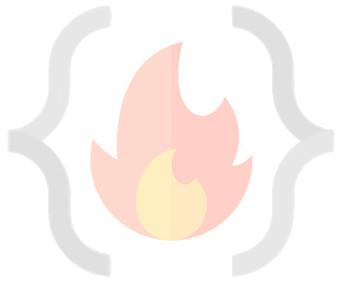
- i. Enhancement on First fit but starts search always from last allocated hole.
- ii. Same advantages of First Fit.

#### **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**

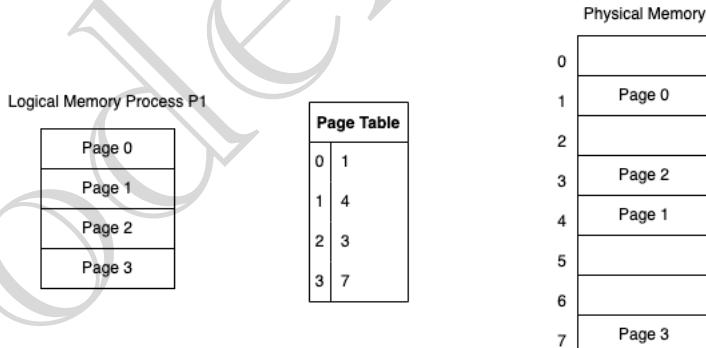
- i. Allocate the largest hole that is big enough.
- ii. Slow, as required to iterate whole free holes list.
- iii. Leaves larger holes that may accommodate other processes.



## LEC-26: 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.

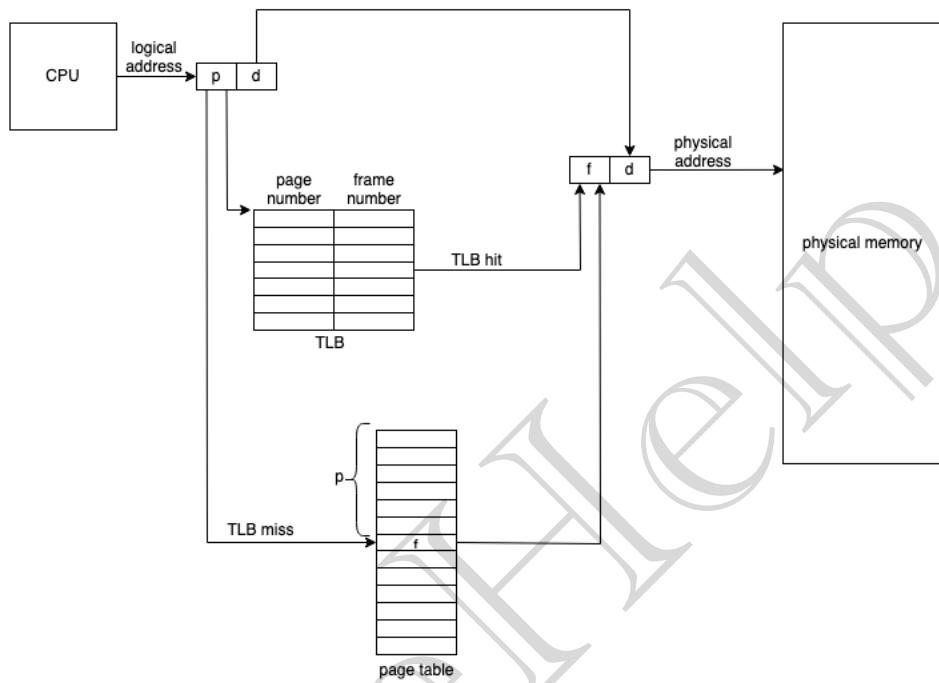
Paging model of logical and physical memory



- 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.

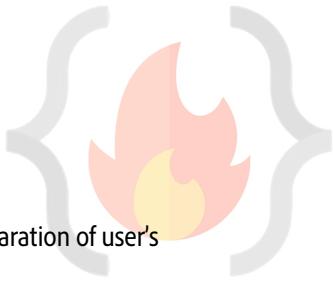
- d. Page table is stored in main memory & because of this when the memory references are made the translation is slow.
- e. When we are retrieving physical address using page table, after getting frame address corresponding to the page number, we put an entry of it into the TLB. So that next time, we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

**Paging hardware with TLB**

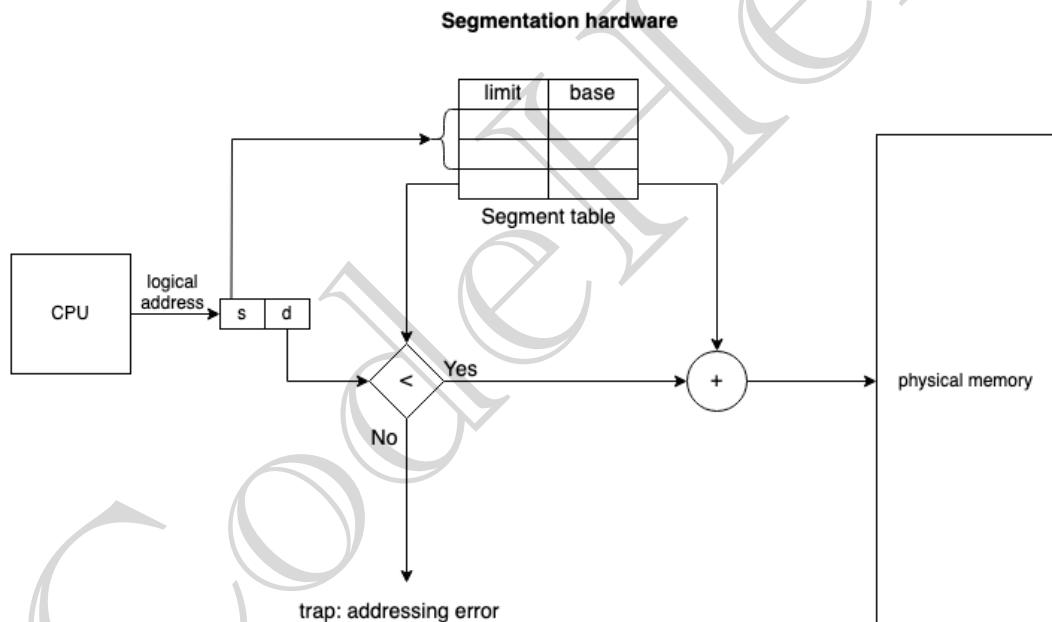


- f. **TLB hit**, TLB contains the mapping for the requested logical address.
- g. **Address space identifier (ASIDs)** is stored in each entry of TLB. ASID uniquely identifies each process and is used to **provide address space protection** and allow to TLB to contain entries for **several different 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: Segmentation | Non-Contiguous Memory Allocation



1. An important aspect of memory management that becomes unavoidable with paging is separation of user's view of memory from the actual physical memory.
2. Segmentation is a 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.  
 $\langle \text{segment-number}, \text{offset} \rangle \{s,d\}$
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 pages** 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.



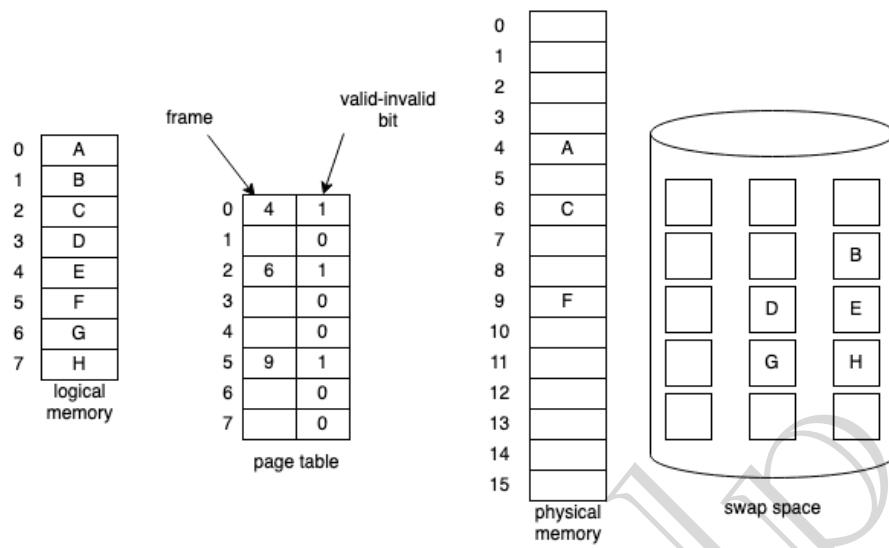
- 9.
10. **Advantages:**
  - a. No internal fragmentation.
  - b. One segment has a contiguous allocation, hence efficient working within segment.
  - c. The size of segment table is generally less than the size of page table.
  - d. It results in a more efficient system because the compiler keeps the same type of functions in one segment.
11. **Disadvantages:**
  - a. External fragmentation.
  - b. The different size of segment is not good for the time of swapping.
12. Modern System architecture provides both segmentation and paging implemented in some hybrid approach.



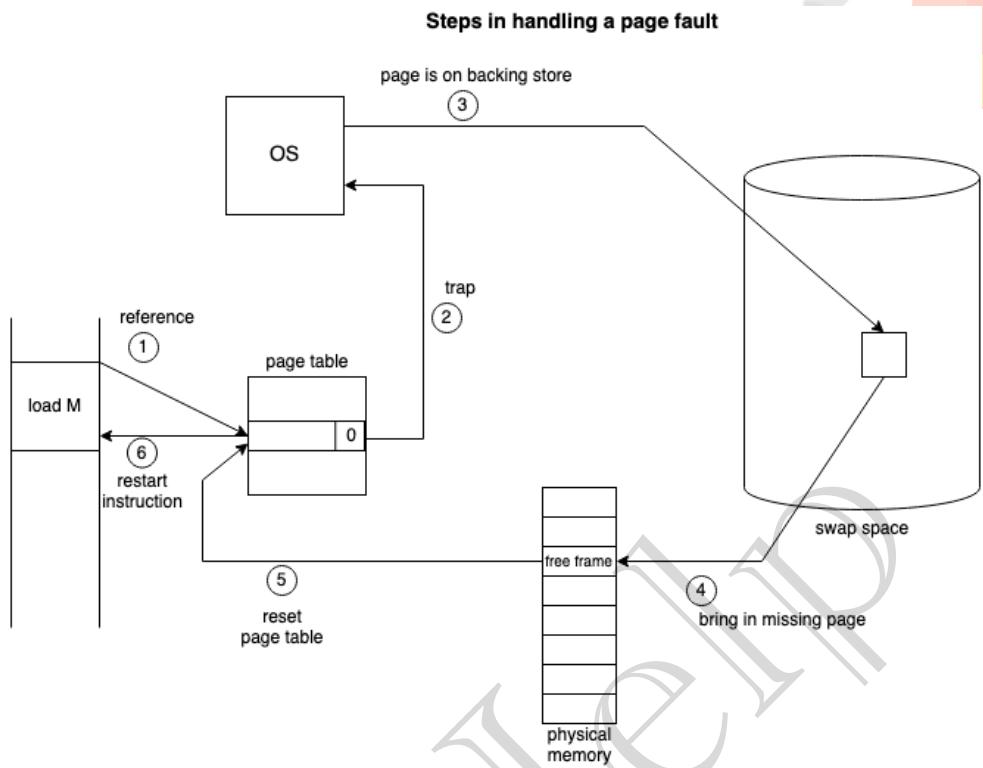
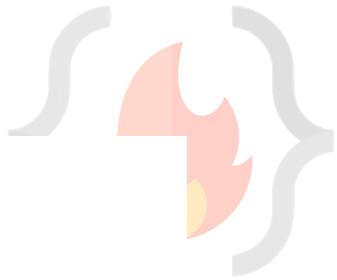
## LEC-28: What is Virtual Memory? || Demand Paging || Page Faults

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 though 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.

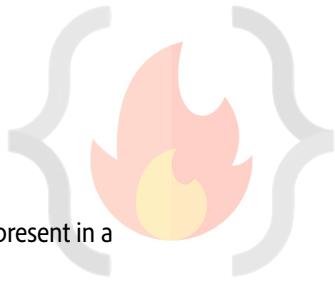
k. We use **locality of reference** to bring out reasonable performance from demand paging.

**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.

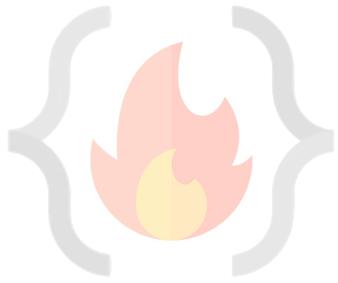
**12. Disadvantages of Virtual Memory**

- a. The system can become slower as swapping takes time.
- b. **Thrashing** may occur.



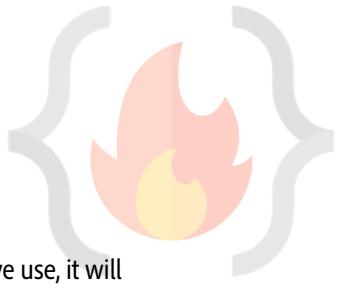
## LEC-29: Page Replacement Algorithms

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.

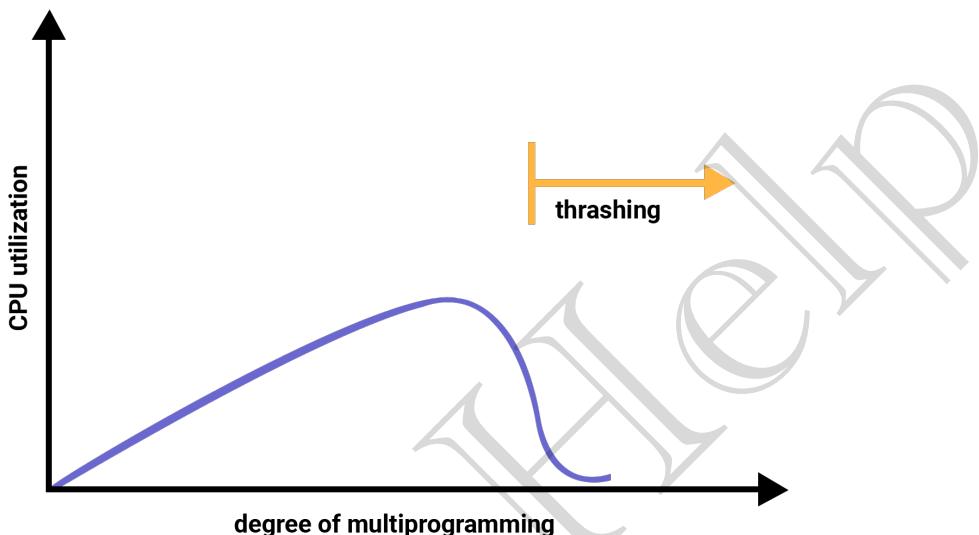
CodeHelp



## 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

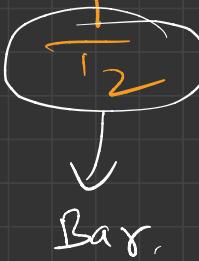
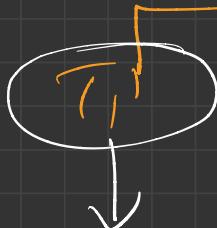
③ Building H2O ~~\*~~

①

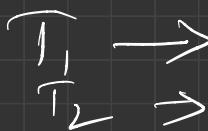
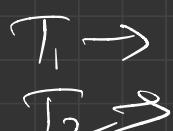
First →

FooBar

FooBar → class → instance → 



Single Process/thread → Foo Bar Foo Bar.

Two threads →  

Code

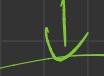
Watch  
Lec-23

```
1 class FooBar {
2     private:
3         int n;
4         std::mutex m;
5         std::condition_variable cv;
6         bool turn;
7     public:
8         FooBar(int n) {
9             this->n = n;
10            turn = 0;
11        }
12
13        void foo(function<void()> printFoo) {
14
15            for (int i = 0; i < n; i++) {
16                std::unique_lock<std::mutex> lock(m);
17                while(turn == 1){
18                    cv.wait(lock);
19                }
20                printFoo();
21                turn = 1;
22                cv.notify_all();
23            }
24        }
25
26        void bar(function<void()> printBar) {
27
28            for (int i = 0; i < n; i++) {
29                std::unique_lock<std::mutex> lock(m);
30                while(turn == 0){
31                    cv.wait(lock);
32                }
33                printBar();
34                turn = 0;
35                cv.notify_all();
36            }
37        }
38    };
```

initially  
 $turn = 0$



$T_1$



$turn = 1$

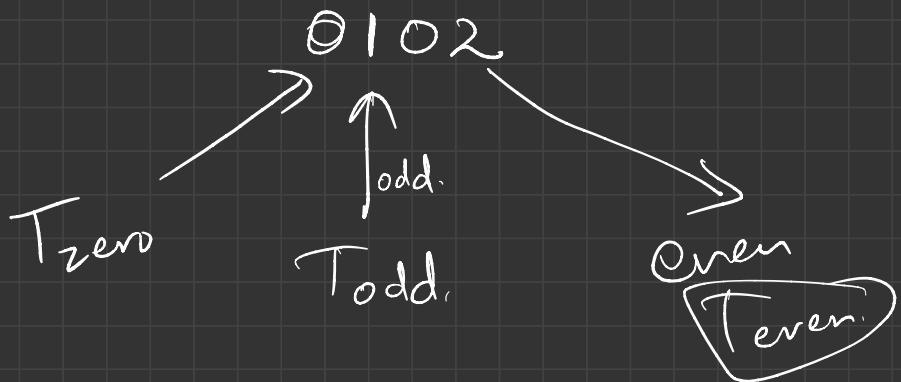


$T_2$

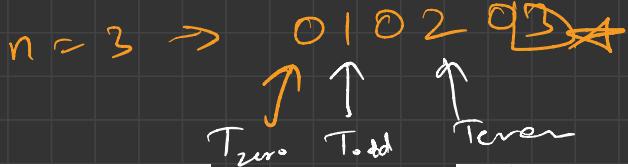
$turn = 0$

## Problem 2 Print zero Even odd

2).  $n = 2$



# Solution 1



$n = 2$

```

1 class ZeroEvenOdd {
2 private:
3     int n;
4     std::mutex m;
5     std::condition_variable cv;
6     int turn;
7     int i;
8 public:
9     ZeroEvenOdd(int n) {
10         this->n = n;
11         turn = 0;
12         i = 1;
13     }
14
15     // printNumber(x) outputs "x", where x is an integer
16     void zero(function<void(int)> printNumber) {
17         while(i <= n){
18             std::unique_lock<std::mutex> lock(m);
19             while(turn != 0 && i <= n){
20                 cv.wait(lock);
21             }
22             if(i > n){
23                 break;
24             }
25             printNumber(0);
26             turn = (i % 2) == 0 ? 2 : 1;
27             cv.notify_all();
28         }
29     }
30 }
```

$\boxed{\text{Turn} = 0}$

```

31 void even(function<void(int)> printNumber) {
32     while(i <= n){
33         std::unique_lock<std::mutex> lock(m);
34         while(turn != 2 && i <= n){
35             cv.wait(lock);
36         }
37         if(i > n){
38             break;
39         }
40         printNumber(i++);
41         turn = 0;
42         cv.notify_all();
43     }
44 }
45
46 void odd(function<void(int)> printNumber) {
47     while(i <= n){
48         std::unique_lock<std::mutex> lock(m);
49         while(turn != 1 && i <= n){
50             cv.wait(lock);
51         }
52         if(i > n){
53             break;
54         }
55         printNumber(i++);
56         turn = 0;
57         cv.notify_all();
58     }
59 }
60 }
```

$\boxed{\boxed{i=3}} \quad \boxed{\text{turn} = 2}$

$\boxed{\boxed{\quad}} \quad \boxed{\text{turn} = 1}$

Case  $\boxed{n = 2}$

- ①  $i = 1 \rightarrow \text{turn} = 1$
- ②  $i = 2 \rightarrow \text{turn} = 2$
- ③  $i = 3$

Output  $\boxed{0 \ 1 \ 0 \ 2}$

③  $\text{N}_2\text{O}$   
→  $\text{N}$  → Three  $\text{N}$  - multiple  
④ → Three  $\text{N}$  → multiple

Thilo Hon

The diagram illustrates a parallel processing model. On the left, the word "Input" is followed by a large bracketed box containing the sequence "JOHNHNHN". Inside this box, four arrows point from the labels "1 thread", "2 threads", "3 threads", and "4 threads" to the corresponding segments of the string. A large curved arrow on the right points from the input box to the label "4 threads". Below the input box, another large bracketed box contains the string "JOHNHNHN" and is labeled "20 threads" below it.

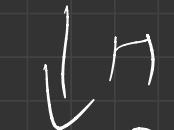
Solutions

```
1 class H2O {
2     std::mutex m;
3     std::condition_variable cv;
4     int turn;
5 public:
6     H2O() {
7         turn = 0;
8     }
9
10    void hydrogen(function<void()> releaseHydrogen) {
11        std::unique_lock<std::mutex> lock(m);
12        while(turn == 2){
13            cv.wait(lock);
14        }
15        releaseHydrogen();
16        ++turn;
17        cv.notify_all();
18    }
19
20    void oxygen(function<void()> releaseOxygen) {
21        std::unique_lock<std::mutex> lock(m);
22        while(turn < 2){
23            cv.wait(lock);
24        }
25        releaseOxygen();
26        turn = 0;
27        cv.notify_all();
28    }
29};
```

turn = 0



turn = 0

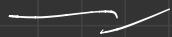


turn = 2



turn = 0

Conditions



turn →

metaphor

conditions