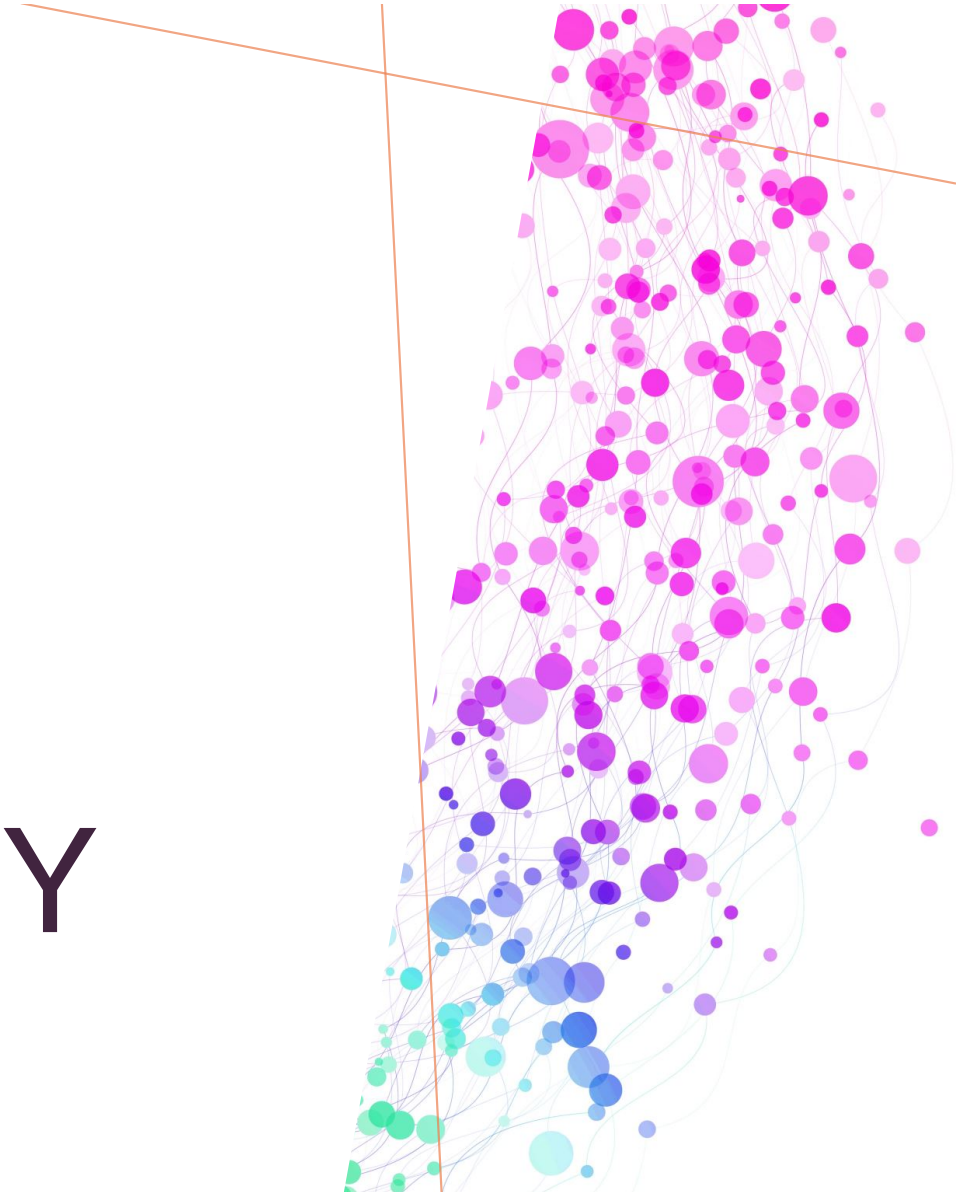
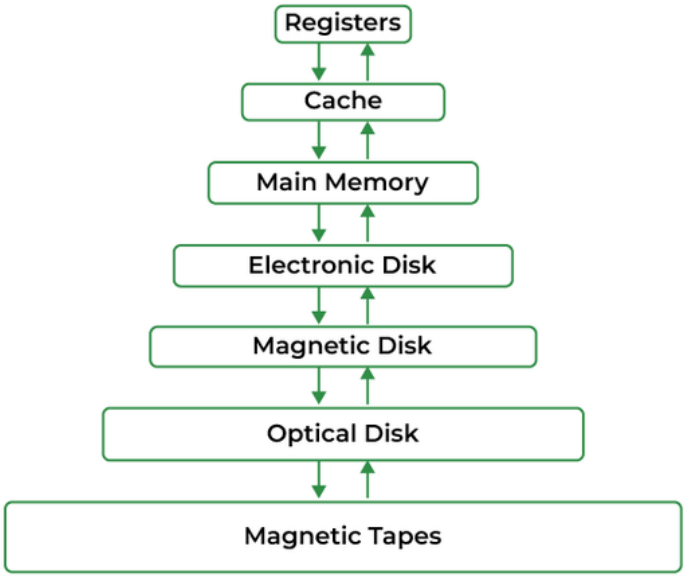
# MAIN MEMORY



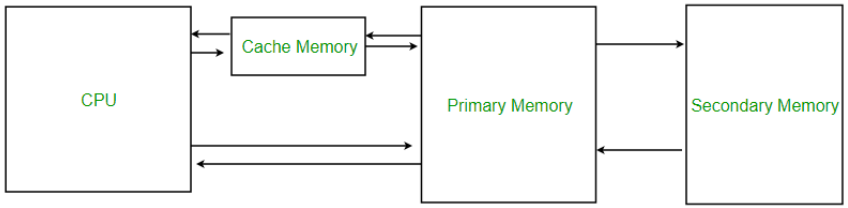
# WHAT IS MAIN MEMORY?

- **Central Role**: Key to computer operation.
- **Structure**: Large array of words/bytes.
- **Function**: Rapidly accessible info for CPU and I/O devices.
- **Usage**: Stores active programs and data.
- **Speed**: Fast data transfer with the processor.
- **Also Known As**: RAM (Random Access Memory).
- **Volatility**: Loses data on power loss.



# CACHE

- **High-Speed Memory**: Special, very fast memory.
- **Smaller & Faster**: Stores copies of frequently used main memory data.
- **Multiple Caches**: Independent caches in a CPU for instructions and data.
- **Purpose**: Reduces average time to access data from main memory.

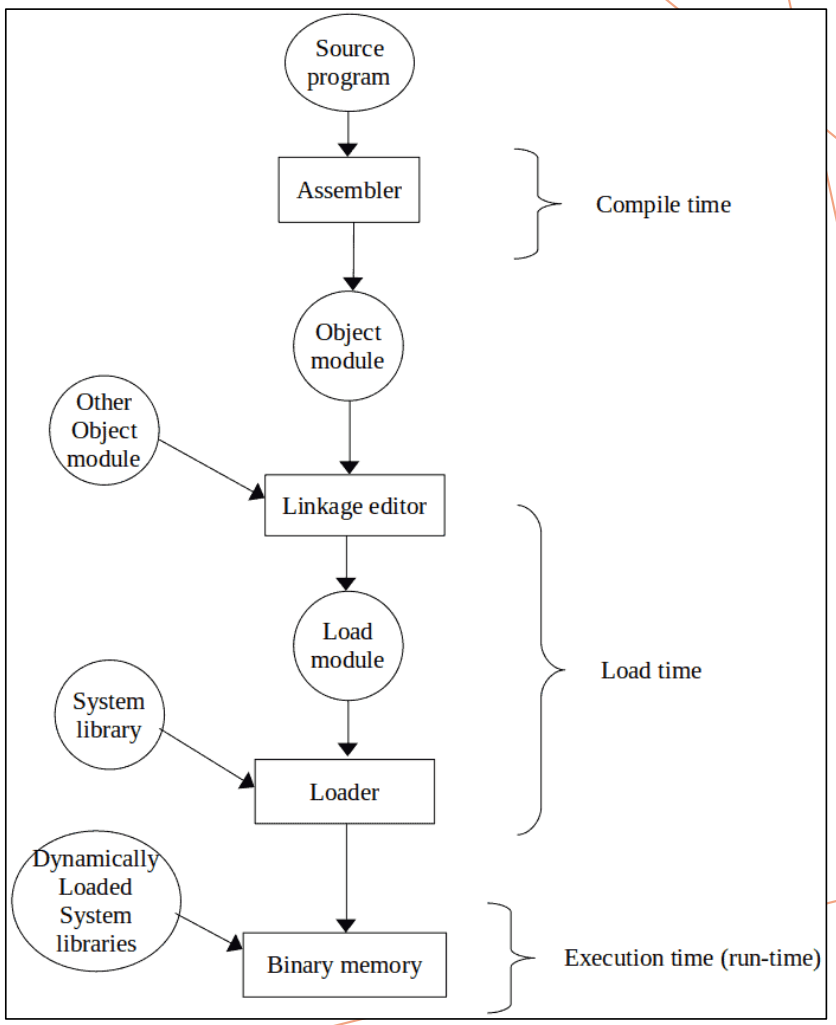


# LOGICAL AND PHYSICAL ADDRESS SPACE

- **Logical Address Space**
  - Generated by the CPU
  - Also called a Virtual address
  - Defines the size of the process
  - Can be changed
- **Physical Address Space**
  - Seen by the memory unit
  - Also called a Real address
  - Computed by the Memory Management Unit (MMU)
  - Remains constant
  - Maps from logical addresses

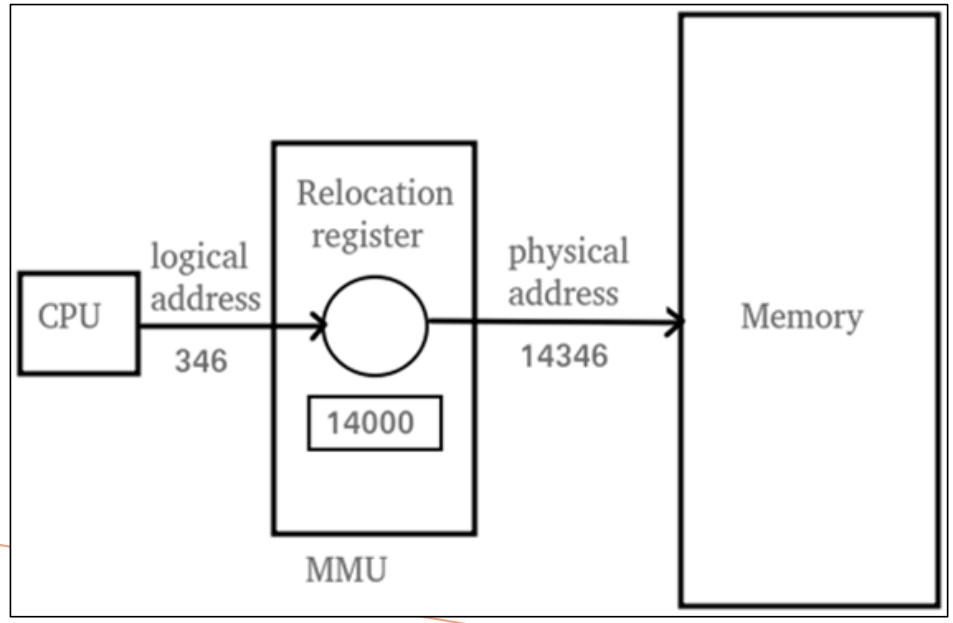
# ADDRESS BINDING

- Compiled code addresses bind to relocatable addresses
- Can happen at three stages: Compile time, Load time, Execution time



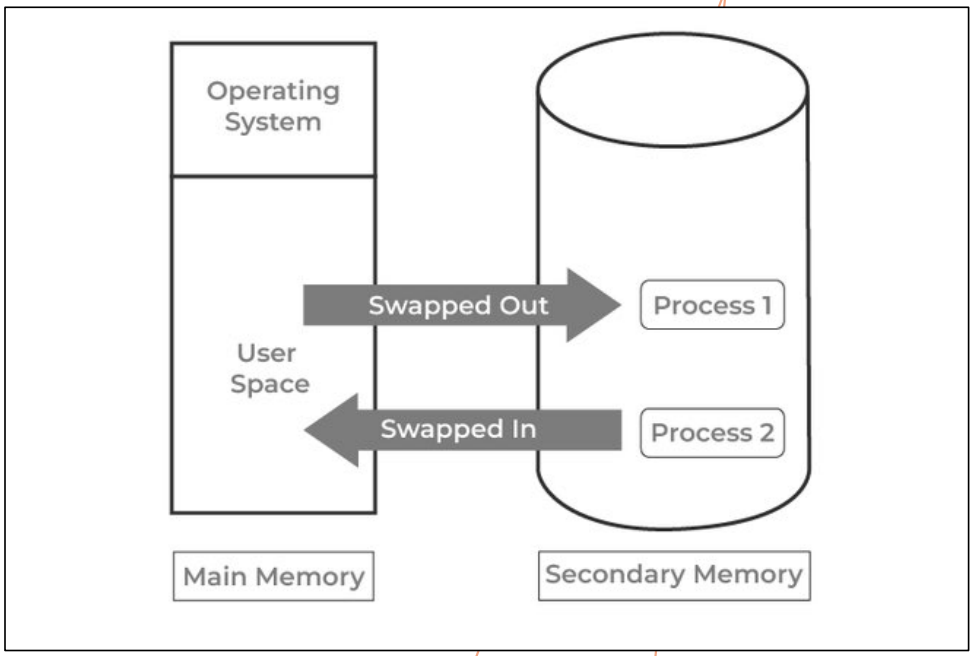
# MEMORY-MANAGEMENT UNIT (MMU)

- Maps virtual to physical address
- Simple scheme: relocation register adds base value to address



# SWAPPING

- Allows total physical memory space of processes to exceed physical memory
- Process swapped out temporarily to backing store, then brought back for execution



# BACKING STORE AND SWAPPING VARIANTS

### Backing Store

- **Fast disk** for storing copies of all memory images.
- **Function**: Holds overflow data from RAM.

### Roll Out, Roll In

- **Swapping variant** for priority-based scheduling.
- **Roll Out**: Moves low-priority processes to backing store.
- **Roll In**: Loads high-priority processes into main memory.

# DYNAMIC STORAGE-ALLOCATION PROBLEM

**First-Fit**

- **Method**: First available block.
- **Pro**: Fast.
- **Con**: Fragmentation.

**Best-Fit**

- **Method**: Smallest suitable block.
- **Pro**: Minimizes wasted space.
- **Con**: Slow, small fragments.

**Worst-Fit**

- **Method**: Largest available block.
- **Pro**: Reduces small fragments.
- **Con**: Inefficient use of memory.

# Example

Consider a swapping system in which memory consists of the following whole sizes in memory order: 10K, 4k, 20k, 18k, 7k, 9k, 12k, and 15k. Which hole is taken for successive segment request of i)12k, ii)10k, iii)9k for first fit? Now repeat the question for best fit and worst fit.

**First Fit**

| | | |
|---|---|---|
| 12k | → | 20k |
| 10k | → | 10k |
| 9k | → | 18k |

**Best Fit**

| | | |
|---|---|---|
| 12k | → | 12k |
| 10k | → | 10k |
| 9k | → | 9k |

**Worst Fit**

| | | |
|---|---|---|
| 12k | → | 20k |
| 10k | → | 18k |
| 9k | → | 15k |

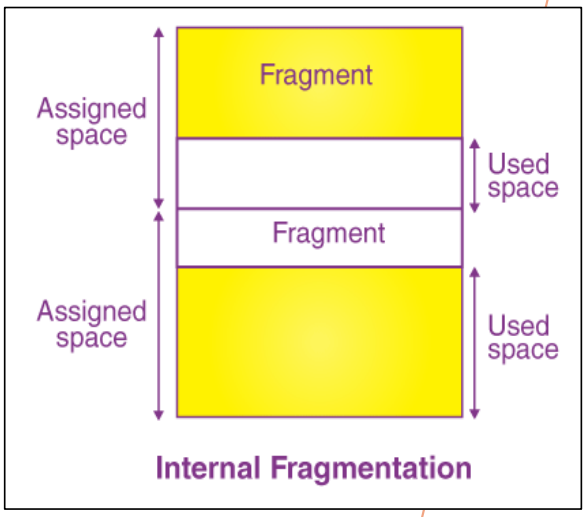
# FRAGMENTATION

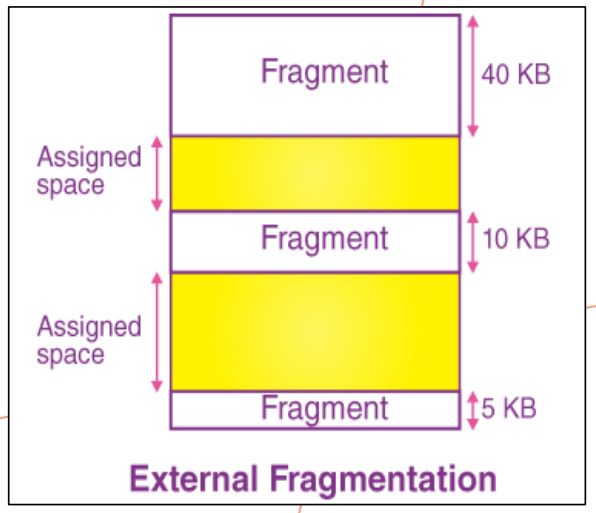
**External Fragmentation**

- **Definition**: Non-contiguous free memory blocks.
- **Cause**: Allocation and deallocation of memory over time.
- **Problem**: Insufficient continuous space for new allocations despite total free space being adequate.

**Internal Fragmentation**

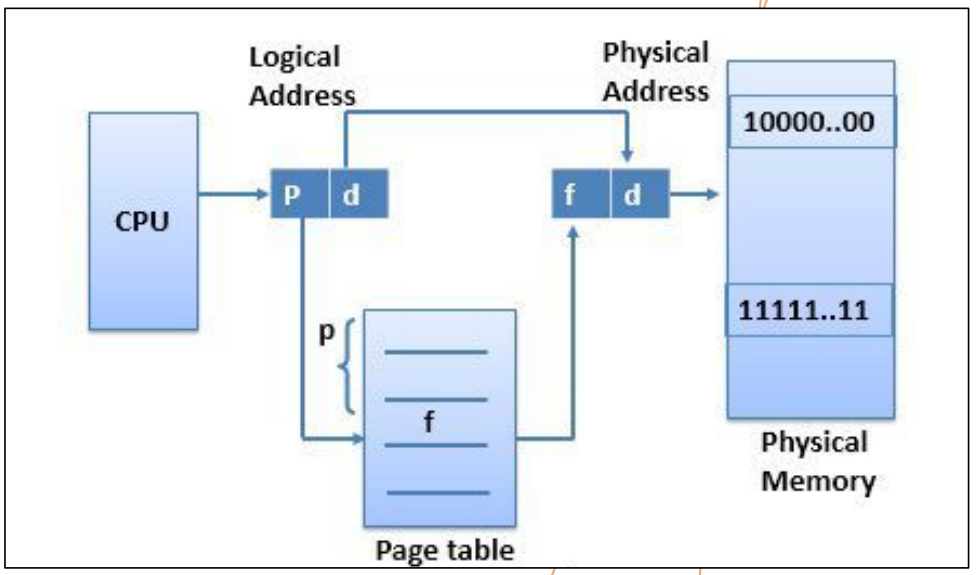
- **Definition**: Allocated memory slightly larger than requested.
- **Cause**: Fixed-sized memory blocks.
- **Problem**: Wasted space within allocated regions.



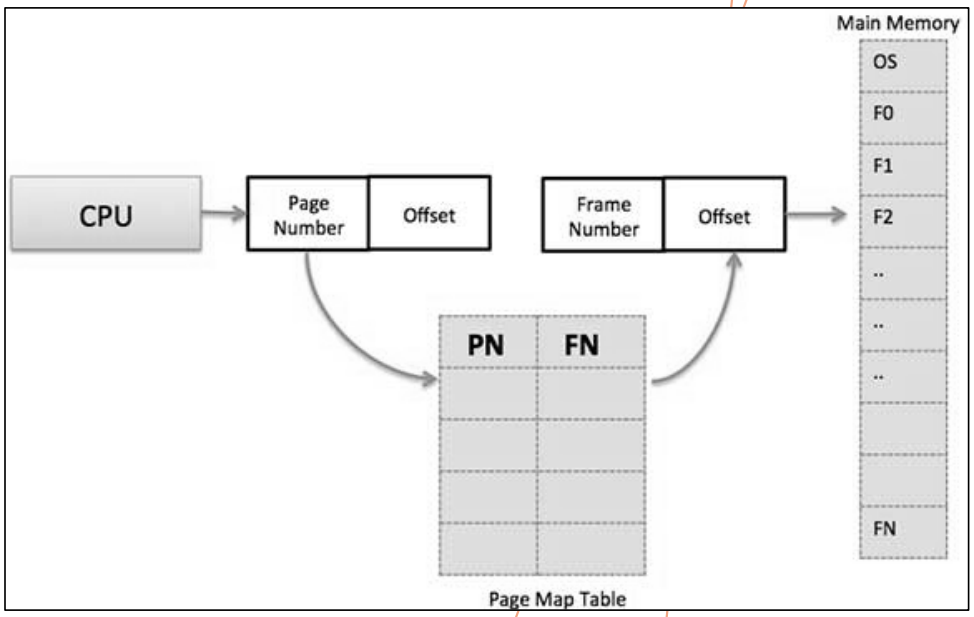
# PAGING

- Physical memory divided into fixed-sized frames
- Logical memory divided into pages
- Page table translates logical to physical addresses



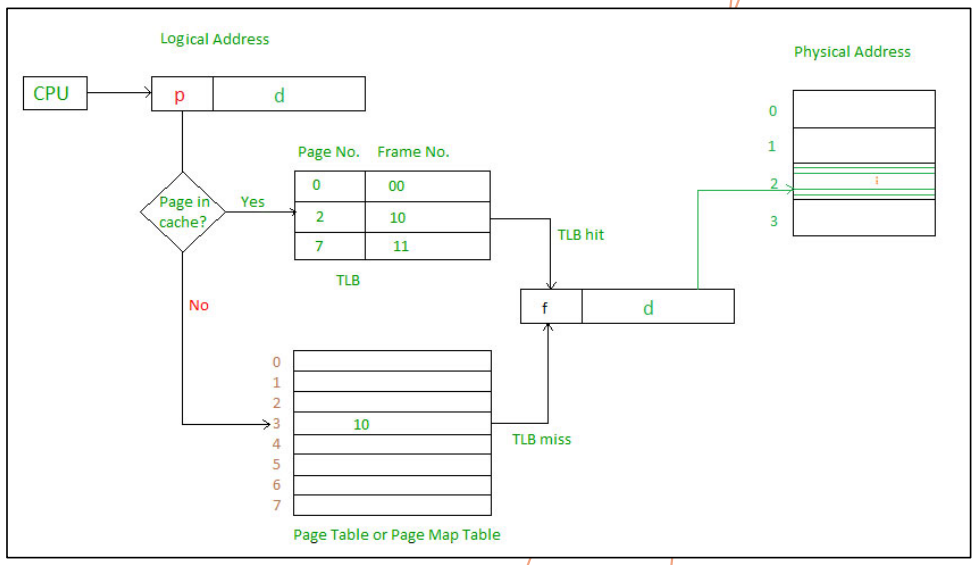
# ADDRESS TRANSLATION

- **Page Number (p)**:
  - Identifies the page in memory.
- **Page Offset (d)**:
  - Specifies the position within the page.
- **Address Translation**:
  - Uses page number and offset to locate data in memory.



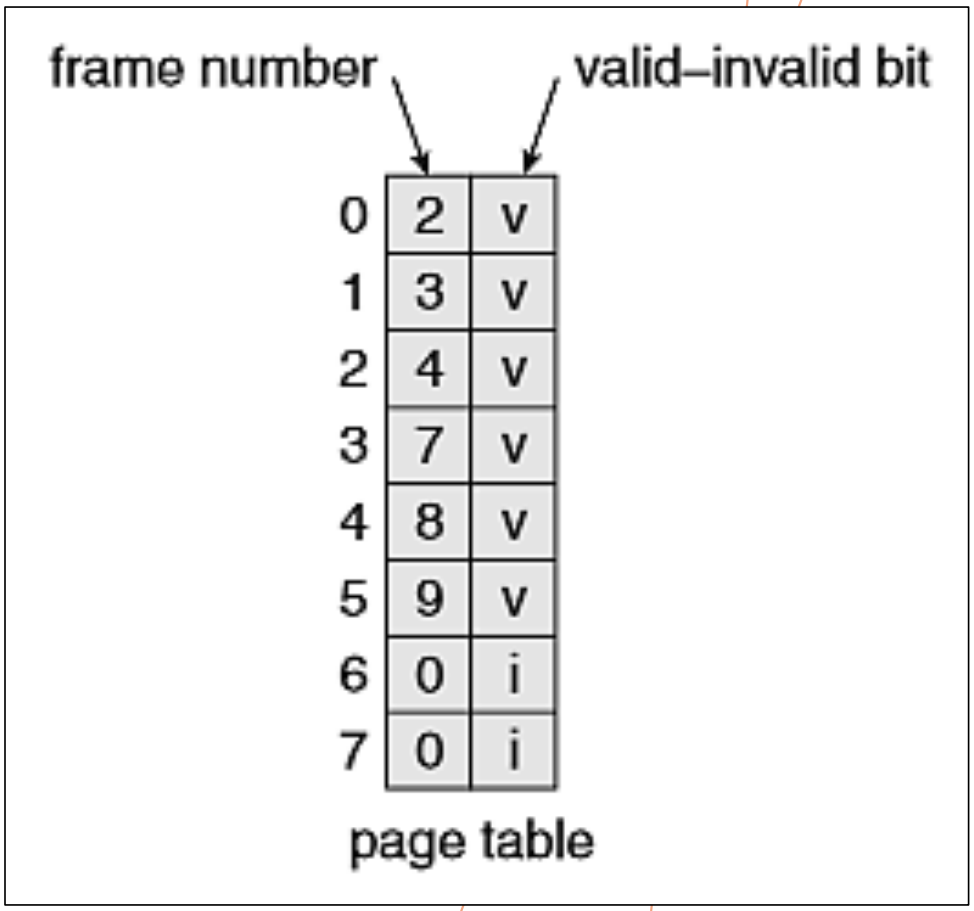
# TRANSITION LOOK-ASIDE BUFFER (TLB)

- **CPU cache** for fast virtual address translation.
- **Features**:
  - Small size
  - Associative mapping
  - Loads data on TLB miss.



# PAGE TABLE PROTECTION

- **Components**:
  - **Valid bit**: Indicates if the mapping is valid.
  - **Invalid bit**: Marks the mapping as invalid.
- **Purpose**:
  - Protects memory regions from unauthorized access.
  - Ensures data integrity by controlling access permissions.



# MULTILEVEL AND HASHED PAGE TABLES

- **Multilevel Page Tables**:
  - Hierarchical organization for efficient memory management.
  - Divides large address spaces into smaller, manageable units.
- **Hashed Page Tables**:
  - Virtual page number hashed into page table for faster access.
  - Offers a more direct mapping approach, reducing lookup time.

# SEGMENT TABLE

- **Function**:
  - Maps two-dimensional physical addresses efficiently.
  - Organizes memory in a structured manner for easy access.
- **Protected Entries**:
  - Valid bits and access privileges (read/write/execute) safeguard entries.
  - Ensures data integrity and security by controlling access permissions.

# MEMORY ADDRESS DECOMPOSITION

**Page Number (PN):**

- Definition: Indicates the page where the data is stored.

Formula:

$$\text{Page number} = \left\lfloor \frac{\text{Memory Address}}{\text{Page Size}} \right\rfloor$$

**Offset (O):**

- Definition: Specifies the position of the data within the page.

Formula:

$$\text{Offset} = \text{Memory Address} \mod \text{Page Size}$$

# EXAMPLE

1. Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a) 3085
b) 42095
c) 215201

# ADDRESS: 3085

## Step 1: Calculate Page Number

- Page number=3085 / 1024=3

## Step 2: Calculate Offset

- Offset=3085 mod 1024=3085−(1024×⌊3085 / 1024⌋)=13

# ADDRESS: 42095

**Step 1: Calculate Page Number**

- Page number = 42095 / 1024 = 41

**Step 2: Calculate Offset**

- Offset = 42095 mod 1024 = $42095 - (1024 \times \lfloor 42095/1024 \rfloor)$ = 111

# ADDRESS: 215201

**Step 1: Calculate Page Number**

- Page number = 215201 / 1024 = 210

**Step 2: Calculate Offset**

- Offset = 215201 mod

  $1024 = 215201 - (1024 \times \lfloor 215201/1024 \rfloor) = 161$

# EXAMPLE TRANSLATING LOGICAL ADDRESSES

•To solve the memory management exercise, start by determining the page size and identifying the page number and offset for each address. Then, refer to the provided page/frame table to translate logical addresses into physical addresses, using the frame's base address and offset. If an entry for the page number doesn't exist in the table, mark it as a page fault. Repeat this process for all given addresses, ensuring accuracy in calculations and entries. (**Page size:** 0.75 KB (768 bytes)).

•Addresses are:

1) 0x3A7
2) 0x1FFC
3) 0x14D3

- Note that the page table has two columns, the page number in decimal, and the frame’s base address in hexadecimal.

# PAGE TABLE

| Page # (dec) | Frame's base address |
|---|---|
| 1 | 0x100 |
| 3 | 0x600 |
| 15 | 0x900 |
| 20 | 0xAA00 |
| 32 | 0x3E500 |

# SOLUTION

| Address (Hex) | Address (Dec) | Page # (Dec) | Frame's Base (Hex) | Offset (Hex) | Physical Address ( Hex) | Page Fault |
|---|---|---|---|---|---|---|
| 0x3A7 | 935 | 935 / 768 = 1 | 0x100 | 935 % 768 = 167 (0xA7) | 0x1A7 | No |
| 0x1FFC | 8188 | 10 | - | 0xFC | - | Yes |
| 0x14D3 | 5331 | 6 | - | 0xD3 | - | Yes |

# ACTIVITY TRANSLATING LOGICAL ADDRESSES

•To solve the memory management exercise, start by determining the page size and identifying the page number and offset for each address. Then, refer to the provided page/frame table to translate logical addresses into physical addresses, using the frame's base address and offset. If an entry for the page number doesn't exist in the table, mark it as a page fault. Repeat this process for all given addresses, ensuring accuracy in calculations and entries. (**Page size:** 0.75 KB (768 bytes)).

•Addresses are:

1) 0x6B3
2) 0x12FF
3) 0x60B4

- Note that the page table has two columns, the page number in decimal, and the frame’s base address in hexadecimal.

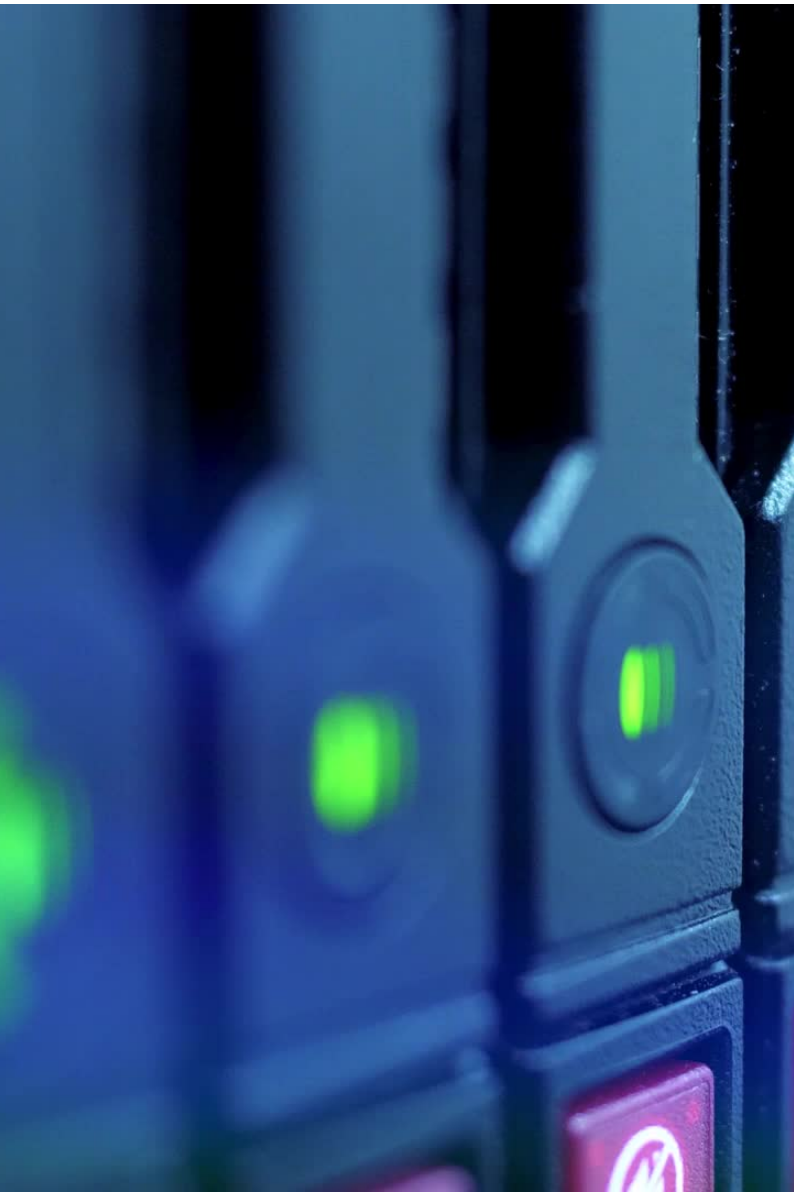
# PAGE TABLE

| Page # (dec) | Frame's base address |
|---|---|
| 1 | 0x100 |
| 3 | 0x600 |
| 15 | 0x900 |
| 20 | 0xAA00 |
| 32 | 0x3E500 |

# SOLUTION

| Address (Hex) | Address (Dec) | Page # (Dec) | Frame's Base (Hex) | Offset (Hex) | Physical Address ( Hex) | Page Fault |
|---|---|---|---|---|---|---|
| 0x6B3 | 1715 | 2 | - | 0xB3 | - | Yes |
| 0x12FF | 4863 | 6 | - | 0xFF | - | Yes |
| 0x60B4 | 24756 | 32 | 0x3E500 | 0xB4 | 0X3E5B4 | No |

# Stack Implementation in Operating Systems

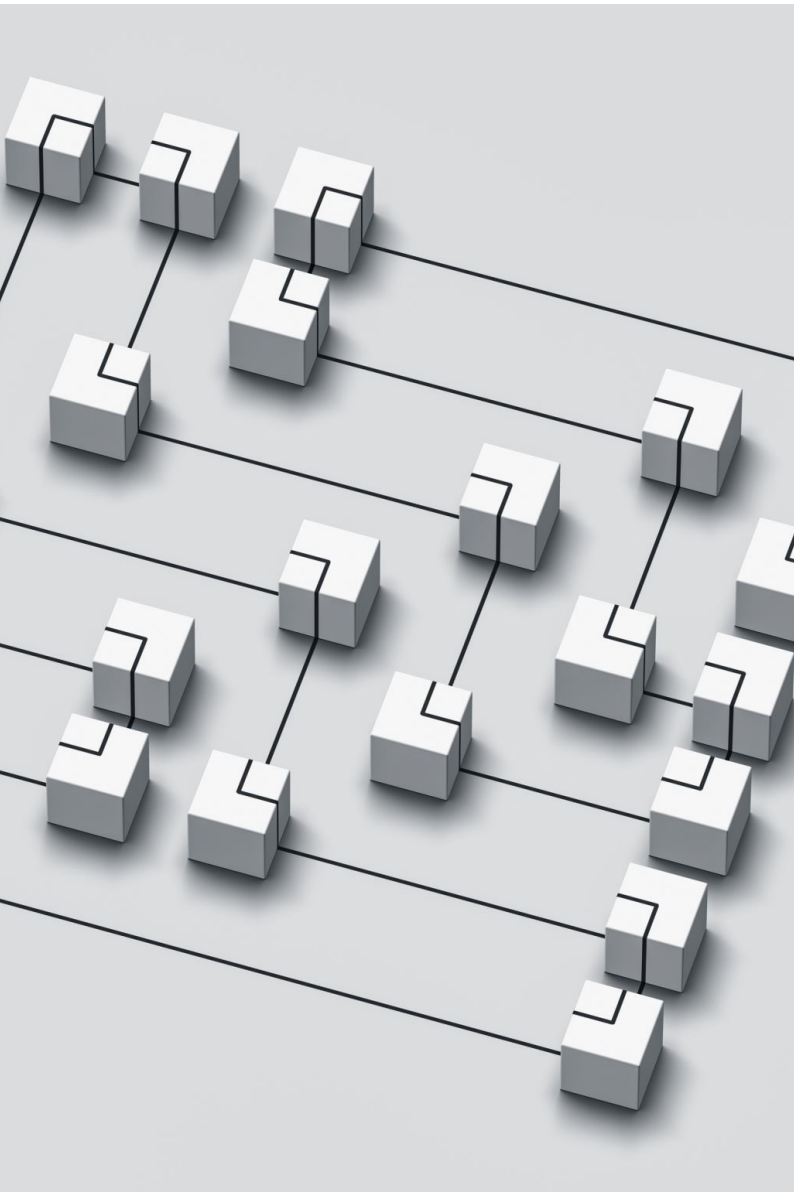


# Introduction to Stack

**Definition:** A stack is an ordered set of components with Last-In-First-Out (LIFO) access.

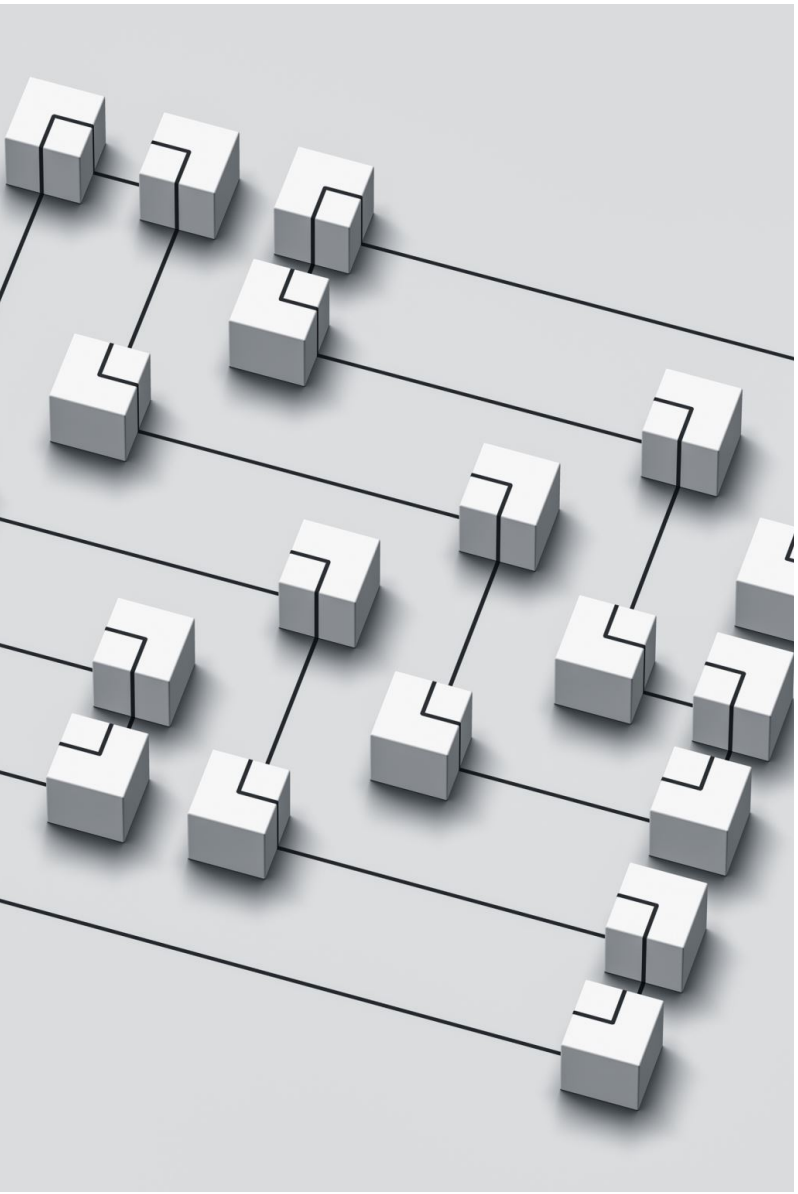
**Key Characteristics:**

- Only the last added item (top of the stack) can be accessed.
- Items can be added (PUSH) or removed (POP) only from the top.
- Also known as a pushdown list or LIFO list.



# Stack Components

- **Addresses Required:**
  - **Stack Pointer:** Contains the address of the top of the stack.
  - **Stack Base:** Contains the address of the lowest location in the reserved block.
  - **Stack Limit:** Contains the address of the top end of the reserved block.



# Address Details

- **Stack Pointer:** Updated on PUSH (decrement) and POP (increment) operations.
- **Stack Base:** First location used when adding to an empty stack.
- **Stack Limit:** Prevents pushing an item when the stack is full.

# Stack Growth

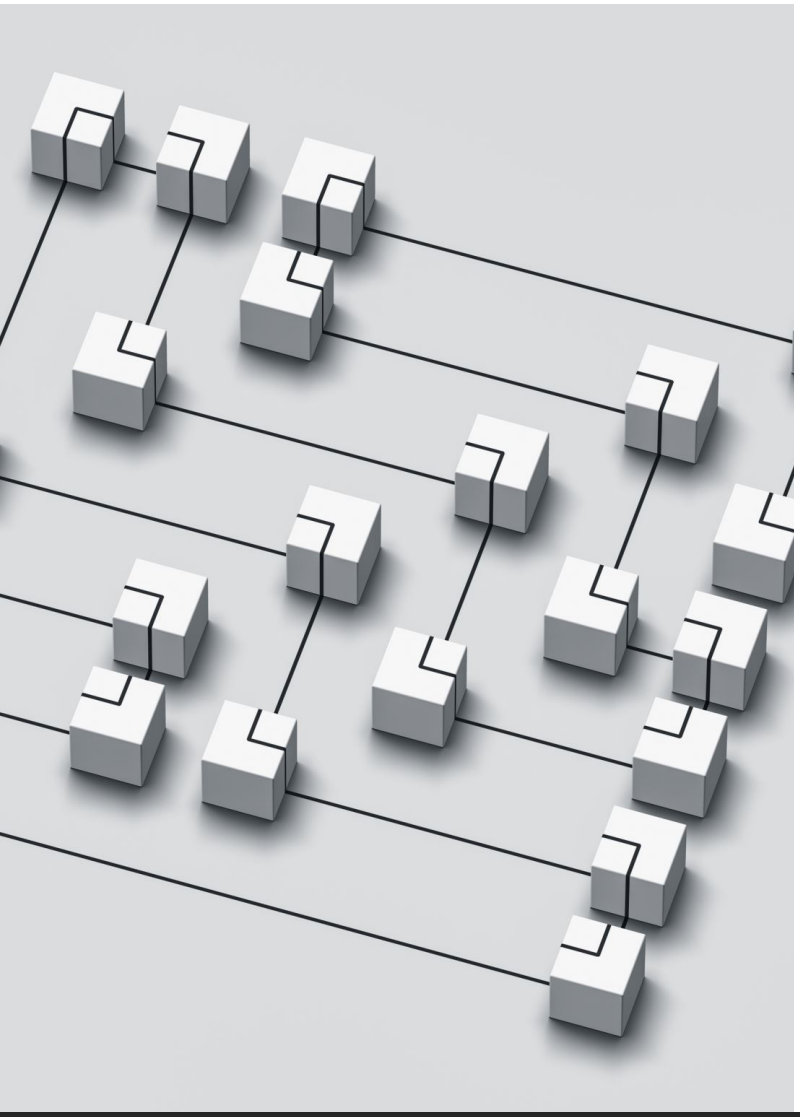
- Traditionally, the base of the stack is at the high address end.
- The stack grows from higher addresses to lower addresses.

# Role of Stack in Program Execution

**Program Calls:**
Processor pushes function parameters and return addresses onto the stack.

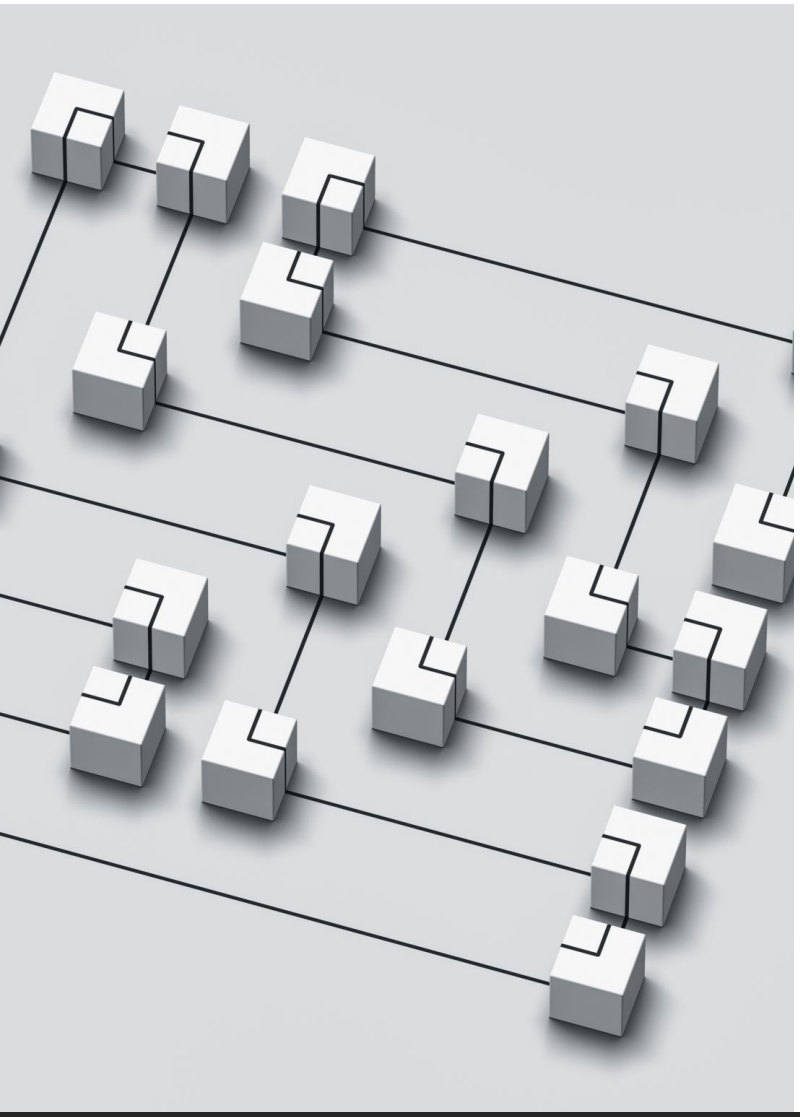
**Function Usage:**
Stack stores local variables and intermediate results.

**Return Process:**
Processor pops return address and local variables from the stack.



# Stack in Interrupts and Exceptions

- **Interrupts/Exceptions:**
  - Processor pushes the current program counter and status onto the stack.
  - Jumps to the interrupt or exception handler.
- **Handler Usage:** Stack stores the context of the interrupted program and restores it afterward.



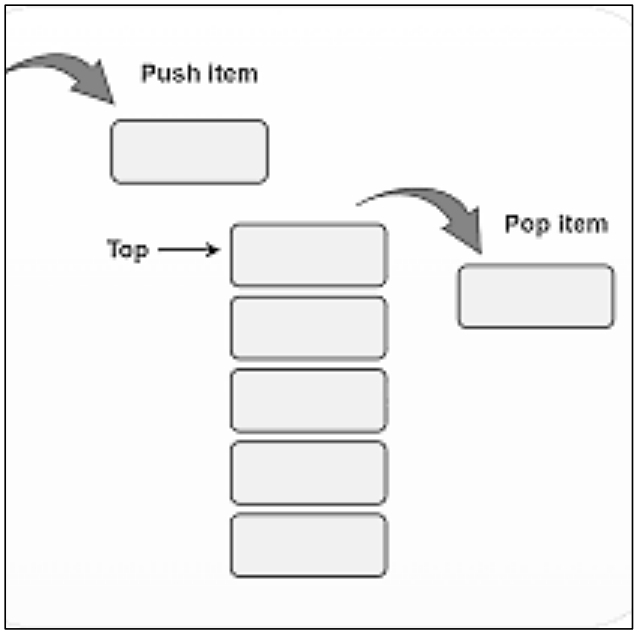
# Implementation Considerations

**Factors to Consider:**

- **Stack Size:** Must accommodate maximum stack usage.
- **Stack Pointer:** Keeps track of the current stack position.
- **Stack Frame Layout:** Organizes function parameters, local variables, and intermediate results.

# Introduction to Push and Pop

Push and pop are fundamental stack operations used in various aspects of operating systems.




They are crucial for:

- Function Call Management
- Interrupt Handling
- Context Switching

# Stack Pointer (SP)

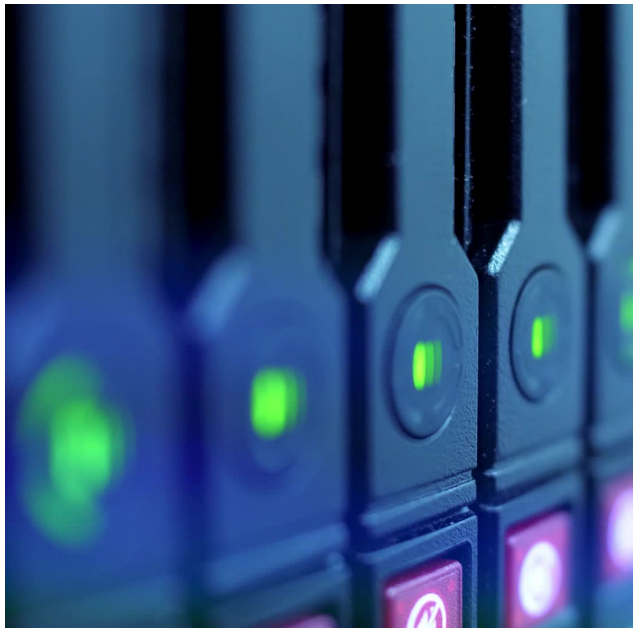
**Definition:** A register that holds the address of the top of the stack.

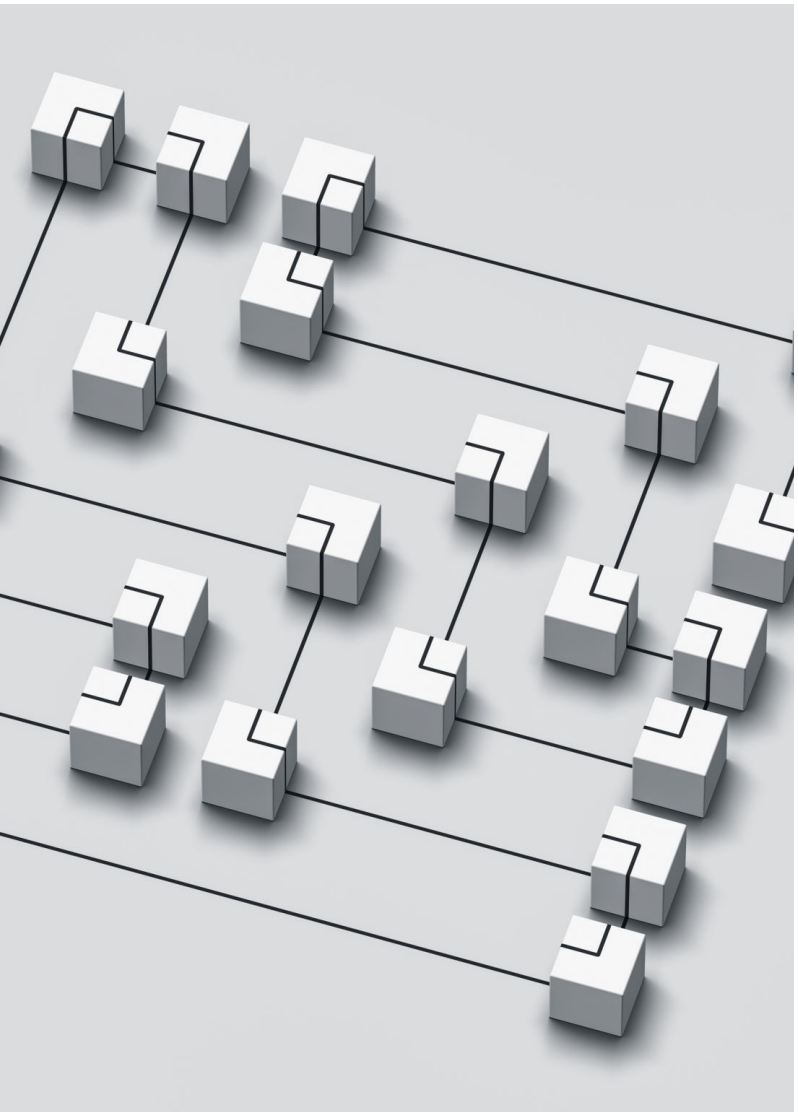
**Updates on Operations:**

- **PUSH:** Decremented to add a new item.
- **POP:** Incremented to remove the top item.

**Importance:**

- **Efficient Access:** Allows quick access to the top of the stack.
- **Error Handling:** Helps in detecting stack overflow (when SP goes below stack base) and stack underflow (when SP exceeds stack limit).



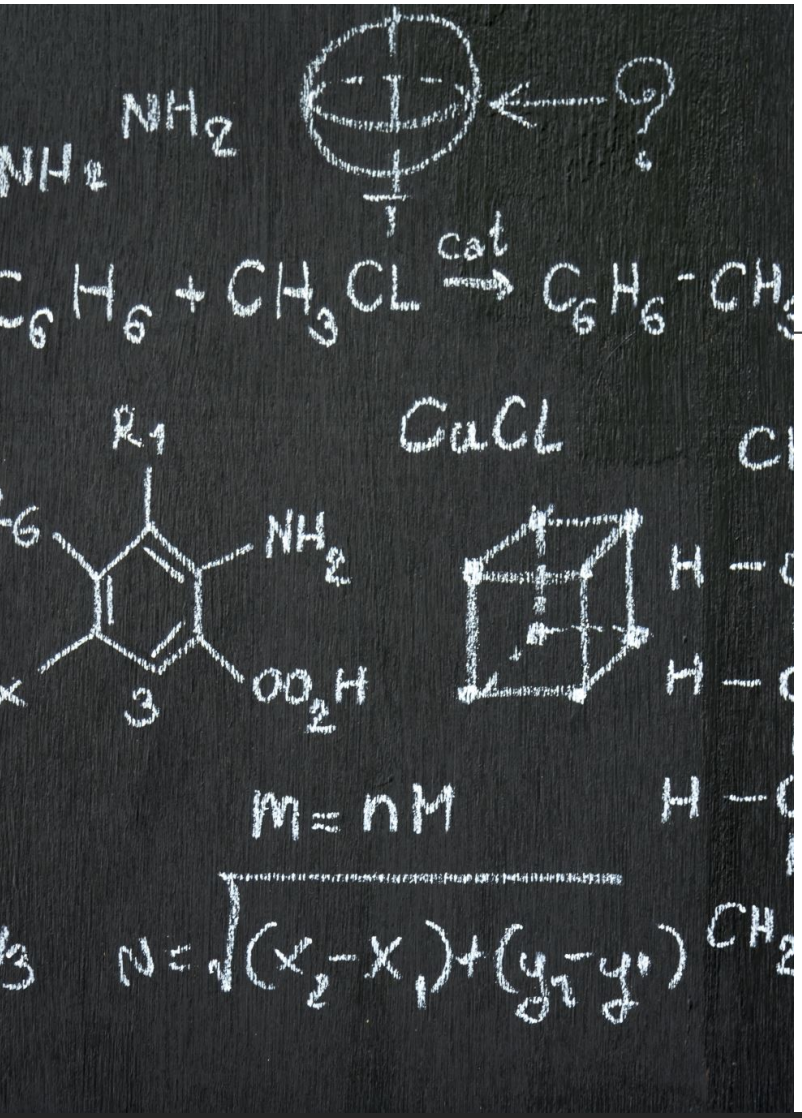


# PUSH Operation

**Definition:** Adds an item to the top of the stack.

**Process:**

1. **Check Stack Limit:** Ensure the stack is not full.
2. **Update Stack Pointer:** Decrement the stack pointer to point to the next available position.
3. **Store Item:** Place the item at the position the stack pointer indicates.

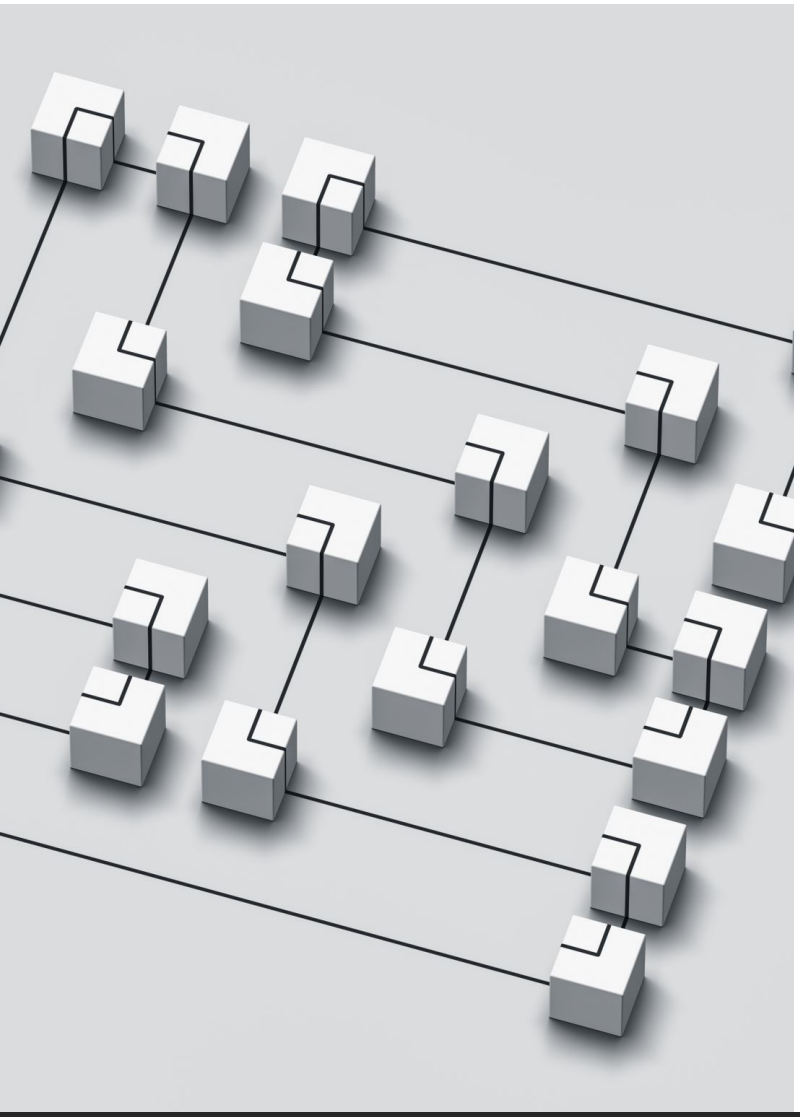



# PUSH Operation Example

- Initial Stack:
- SP = 4
- Stack: [1, 2, 3] (Top = 3)
- PUSH(4):
- SP is decremented (SP = 3).
- Item 4 is placed at position 3.
- Updated Stack: [1, 2, 3, 4] (Top = 4)

**Visual Representation:**

- Initial State: [1, 2, 3]
- After PUSH: [1, 2, 3, 4]



# POP Operation

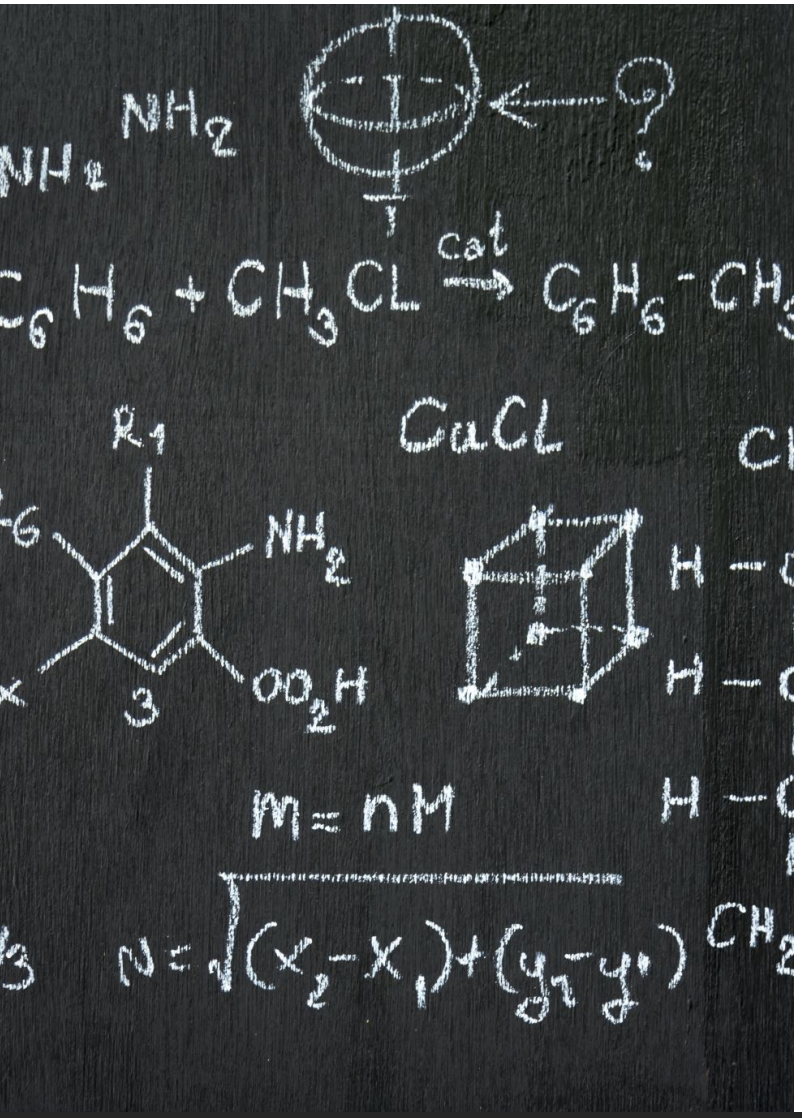
**Definition:** Removes an item from the top of the stack.

**Process:**

**1.Check Stack Base:** Ensure the stack is not empty.

**2.Retrieve Item:** Get the item from the position indicated by the stack pointer.

**3.Update Stack Pointer:** Increment the stack pointer to point to the new top of the stack.

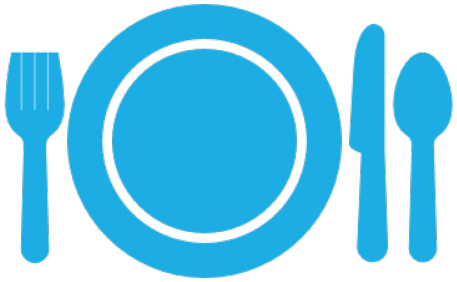



# POP Operation Example

- Initial Stack:
- SP = 3
- Stack: [1, 2, 3, 4] (Top = 4)
- POP():
- Item 4 is retrieved from position 3.
- SP is incremented (SP = 4).
- Updated Stack: [1, 2, 3] (Top = 3)

**Visual Representation:**

- Initial State: [1, 2, 3, 4]
- After POP: [1, 2, 3]



# Understanding Push and Pop Operations Using Dish Cleaning Analogy

# Push Operation in Dish Cleaning

- **Analogy: Adding a dish to the stack**
  - You wash a plate and place it on the stack of clean plates.
  - The newly washed plate is added on top of the stack.
- **Push Operation in OS**
  - Adding data (e.g., function context) to the stack.
  - Example: When a function is called, its context is pushed onto the stack.

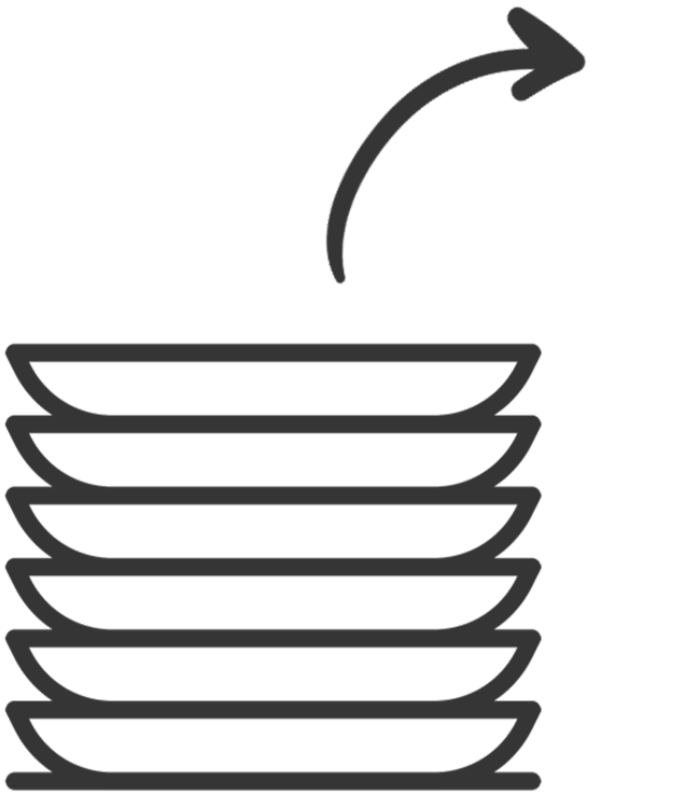
# Pop Operation in Dish Cleaning

- **Analogy: Removing a dish from the stack**
  - When you need a clean plate, you take the top one from the stack.
  - The top plate is removed from the stack.
- **Pop Operation in OS**
  - Removing data (e.g., function context) from the stack.
  - Example: When a function call is completed, its context is popped from the stack.



# Example Scenario - Push

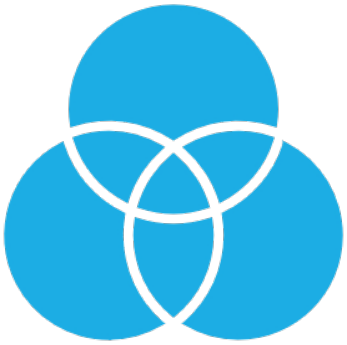
- **Starting with no dishes (empty stack)**
- **Washing dishes (Push):**
  - Wash Plate 1 (Push Plate 1 onto the stack)
  - Wash Plate 2 (Push Plate 2 onto the stack)
  - Wash Plate 3 (Push Plate 3 onto the stack)





# Example Scenario - Pop

1. **Using dishes (Pop):** Take Plate 3 (Pop Plate 3 from the stack)
2. Take Plate 2 (Pop Plate 2 from the stack)
3. Take Plate 1 (Pop Plate 1 from the stack)



# What other real-life analogy do you think exists?

# Real-life Analogies

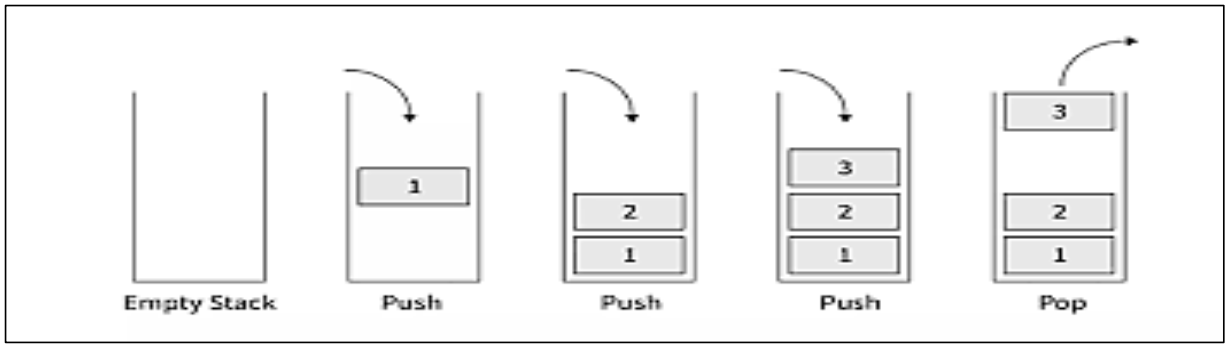
1. **Stack of Pancakes**: Imagine a stack of pancakes where you can only add new pancakes to the top of the stack (push operation) and remove the top pancake (pop operation).
2. **Stack of Building Blocks**: Children often play with building blocks, stacking them one on top of the other. Each time a new block is added, it goes on top of the stack, and when a block is removed, it's taken from the top.
3. **Line at a Ticket Counter**: In a line at a ticket counter, people join the line (push) and are served in the order they joined. When someone is served, they are removed from the front of the line (pop).
4. **Stack of Music CDs**: Stack CDs on top of each other; when you want to listen to one, you take the top CD (pop), and when you buy a new CD, you add it to the top of the stack (push).
5. **Stack of Nested Boxes**: Imagine a set of nested boxes where each box can contain smaller boxes. When you add a new box to the stack, it goes on top, and when you remove a box, you take the top one off.
6. **Stack of Coins in a Coin Dispenser**: In a coin dispenser, coins are stacked on top of each other, and when you dispense a coin, it's taken from the top of the stack.
7. **Stack of Luggage at an Airport**: Luggage is stacked on top of each other as it arrives at the airport. When someone collects their luggage, it's taken from the top of the stack.

# Function Call Management

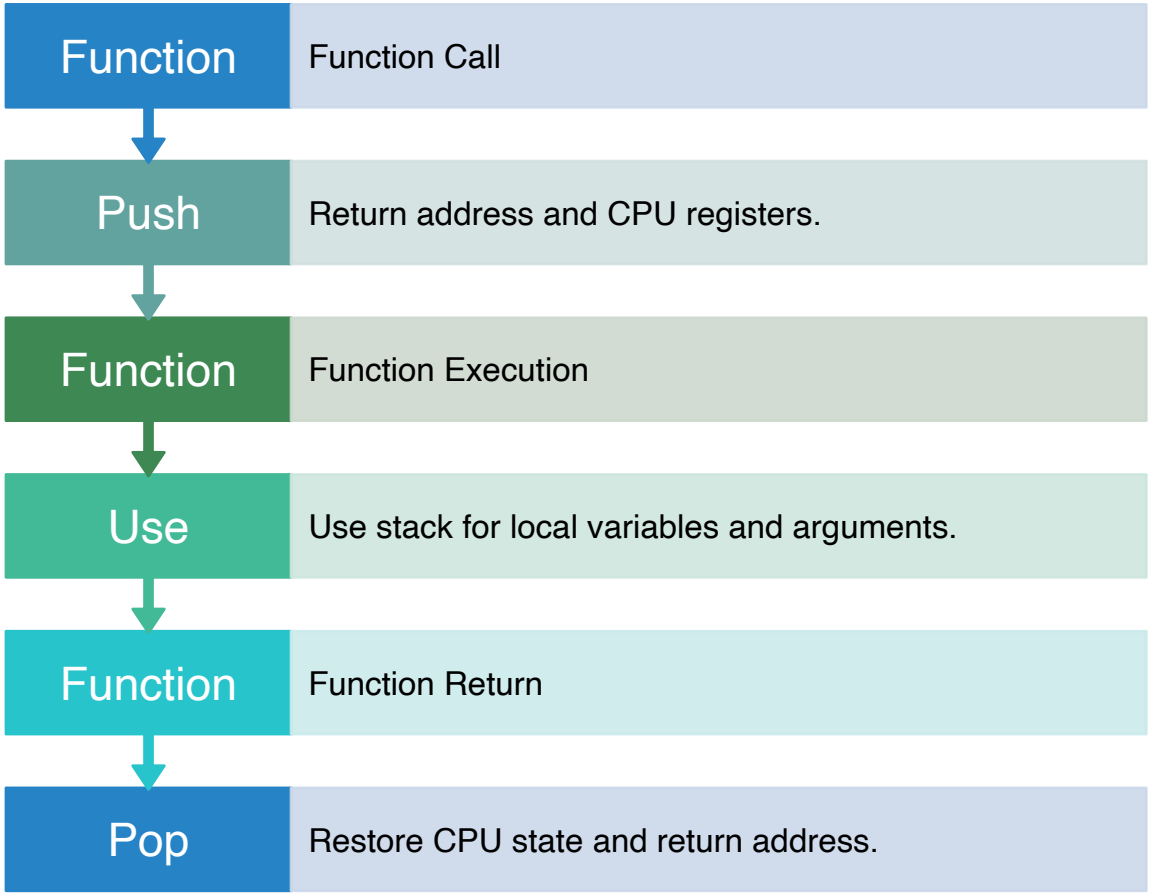
When a function is called:

- **Push**: Save return address and registers.
- **Pop**: Restore return address and registers.

This allows the function to execute without losing the context of the caller.



# Function Call Management - Detailed Process



# Interrupt Handling

During an interrupt:

- **Push**: Save program counter and status register.
- **Pop**: Restore program counter and status register.

This allows the interrupt service routine (ISR) to execute.

# Interrupt Handling - Detailed Process

**Interrupt Occurs**

- Push: Program counter and status register.

**ISR Execution**

- Handle the interrupt.

**Return from Interrupt**

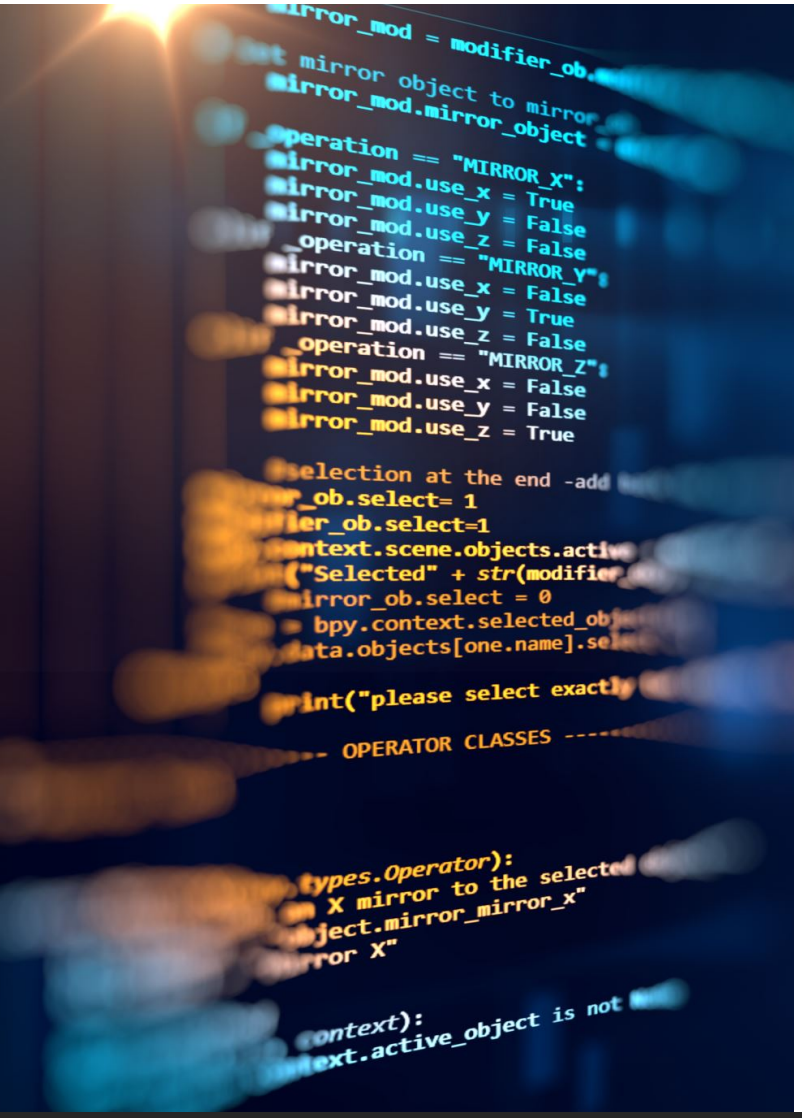
- Pop: Restore CPU state and program counter.

# Context Switching

In multitasking systems:

- **Push**: Save current process state.
- **Pop**: Restore next process state.

This allows the CPU to switch between processes.



# Context Switching - Detailed Process

1. **Save Current Process State:**
   - Push: CPU registers and program counter.
2. **Scheduler Switches Context:**
   - Select next process to run.
3. **Restore Next Process State:**
   - Pop: CPU registers and program counter.

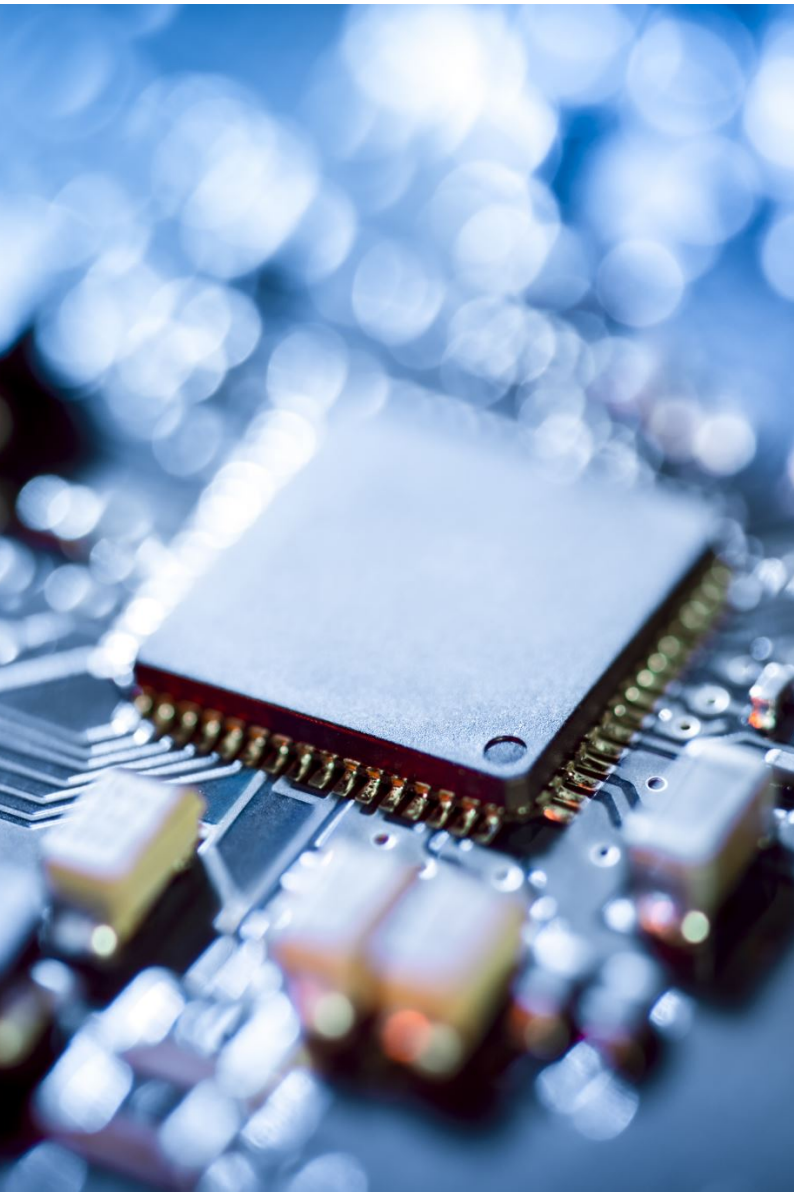
# Example in Assembly Language

Example of push and pop in x86 assembly:

1. **Push**: Save register states.
2. **Operation**: Perform tasks.
3. **Pop**: Restore register states.
4. **Exit**: End program.

# Example in Assembly Language - Code

```
section .data
    msg db 'Hello, world!', 0

section .bss

section .text
global _start

_start:
    push eax
    push ebx
    ; perform operations
    pop ebx
    pop eax
    mov eax, 1
    xor ebx, ebx
    int 0x80
```



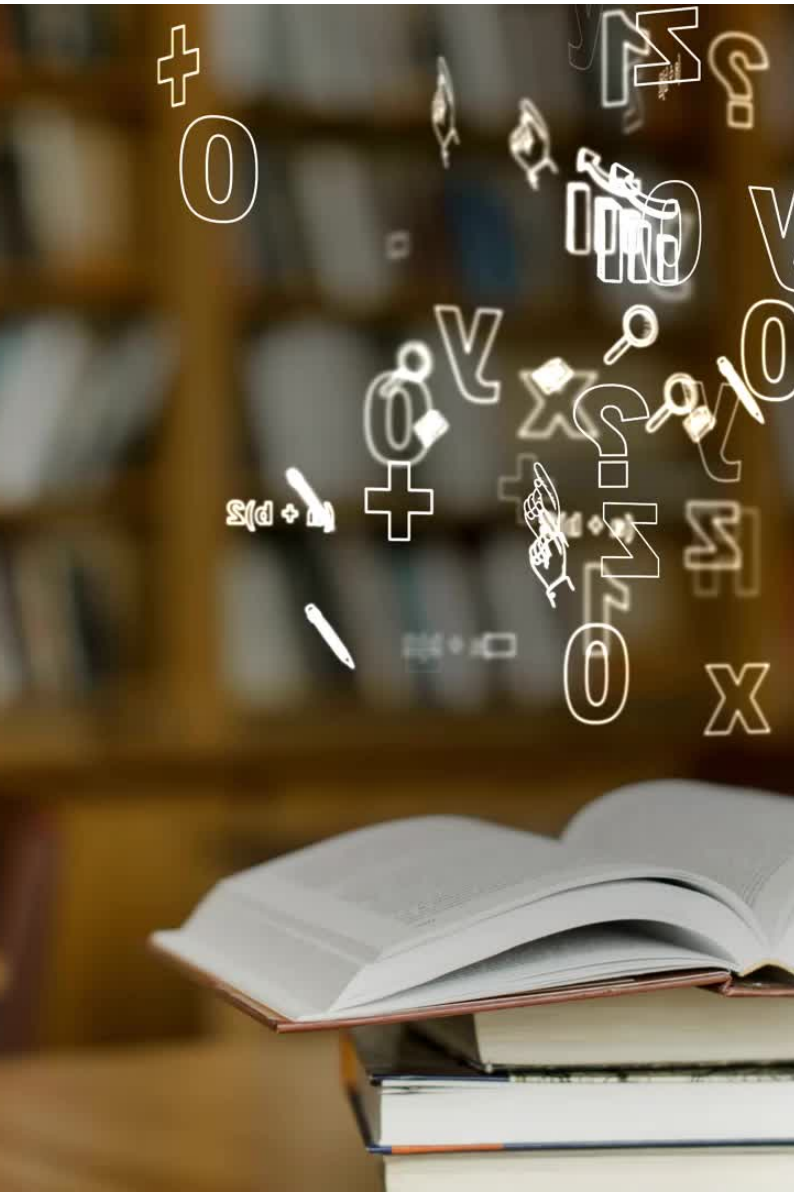
# Importance of Push and Pop

- **Memory Management:** Essential for efficient memory allocation and deallocation.
- **Function Calls:** Facilitates storing and retrieving local variables, parameters, and return addresses.
- **Recursion:** Prevents stack overflow errors by managing recursive function calls.
- **Interrupt Handling:** Utilized for saving and restoring CPU states during interrupts.
- **Context Switching:** Enables efficient process state saving and loading during multitasking.
- **Resource Management:** Crucial for allocating and deallocating system resources in a stack-like manner.

# Activity

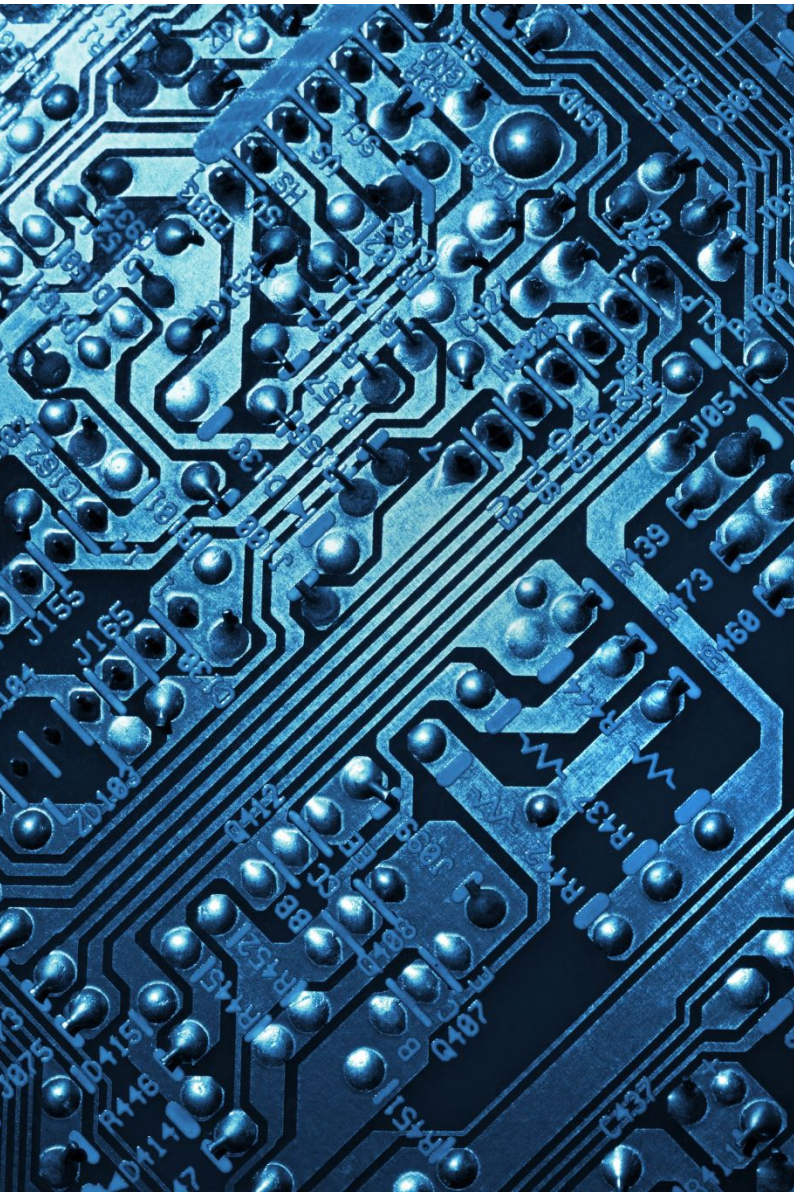
https://www.sigcis.org/files/A%20brief%20history.pdf

- Scan this QR code or copy the link above to read the article titled ‘A brief history of the stack’ by Sten Henriksson.
- Then, be prepared to answer multiple-choice questions that the professor will ask in class.



# What are the key axioms that define the behavior of a stack?

- Axiom 1: isempty(create()) - A newly created stack is empty.
- Axiom 2: ¬ isempty(push(x, S)) - A stack with an added element is not empty.
- Axiom 3: pop(push(x, S)) = S - Pushing and then popping an element leaves the stack unchanged.
- Axiom 4: top(push(x, S)) = x - The top of a stack after pushing an element is the pushed element.
- Axiom 5: ¬isempty(S) ⇒ (push(top(S),pop(S))=S) - Removing and then replacing the top element of a non-empty stack leaves it unchanged.



Who is credited with emphasizing the importance of the stack in the development of computer science?

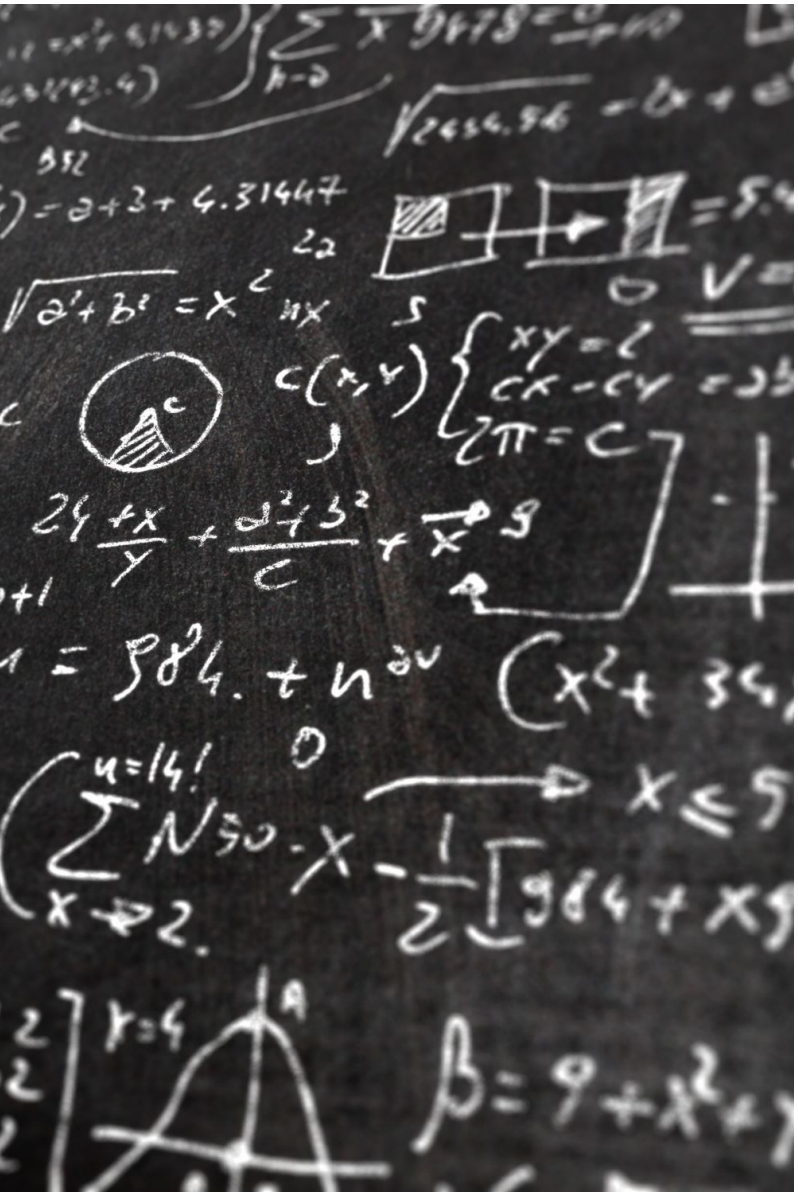
---

Michael Mahoney highlighted the role of stacks in the theory of automata and formal languages, noting their foundational importance between 1955 and 1970.

How did stacks facilitate the development of programming languages like Algol 60?

---

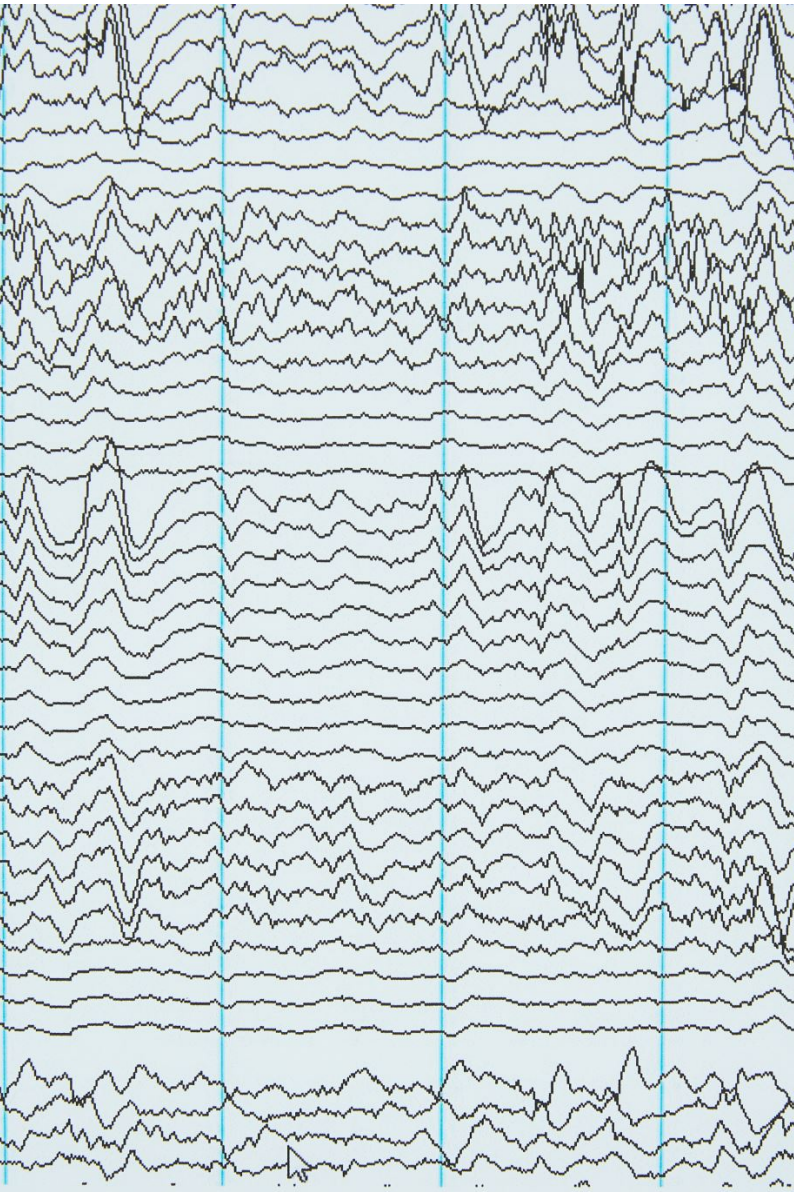
Stacks allowed Algol 60 to handle procedure calls and block structures effectively, enabling recursion and proper variable scope management.



What is the historical significance of the "cellar principle" in stack development?

---

The cellar principle, introduced by F.L. Bauer and K. Samelson, involved storing intermediate results in reverse order, facilitating the translation of infix to postfix notation and influencing the design of recursive languages.



What is a cumulative tale, and how does it illustrate the stack principle?

---

A cumulative tale, like "The Old Woman and Her Pig," adds incidents sequentially and resolves them in reverse order, akin to how a stack operates.

Who is credited with the first use of a stack in a computer science context according to the OED?

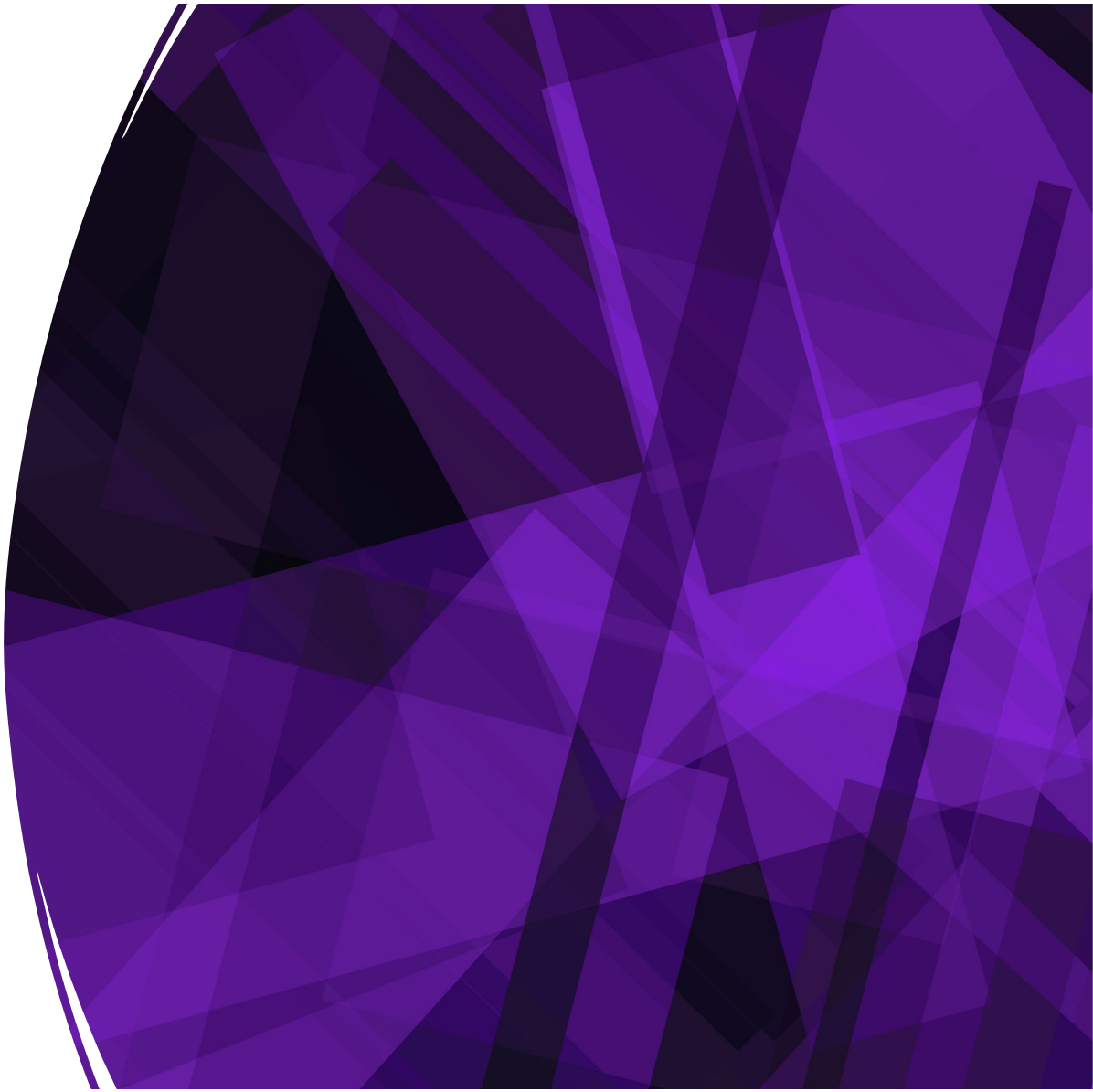
a) Donald Knuth

b) F.L. Bauer

c) E.W. Dijkstra 

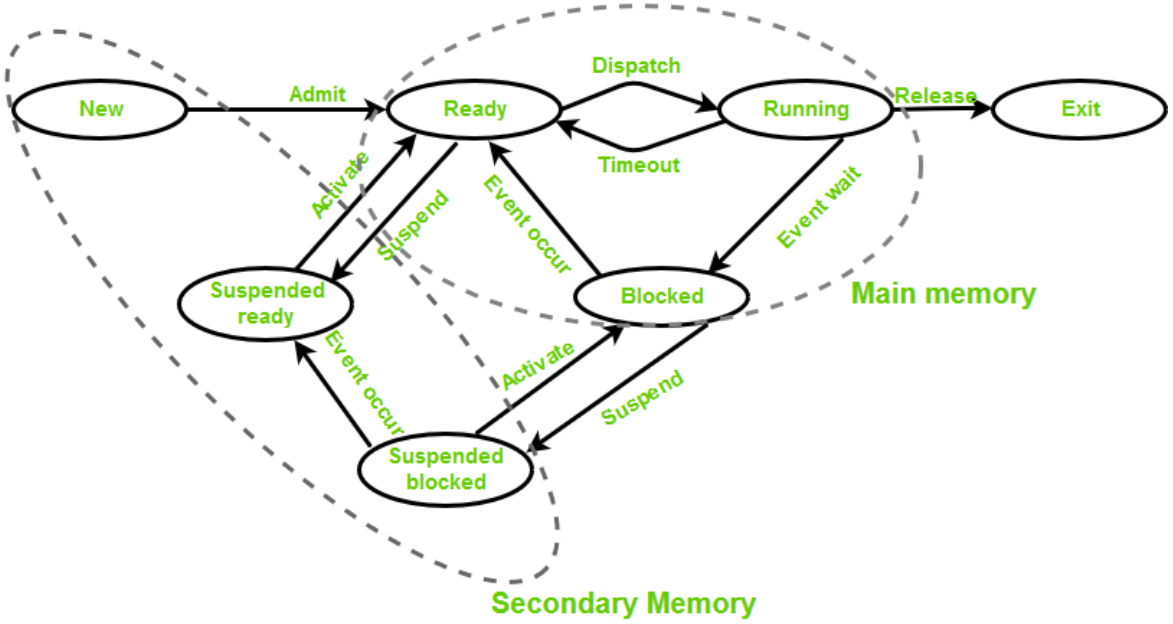
d) John Backus

# CPU Scheduling



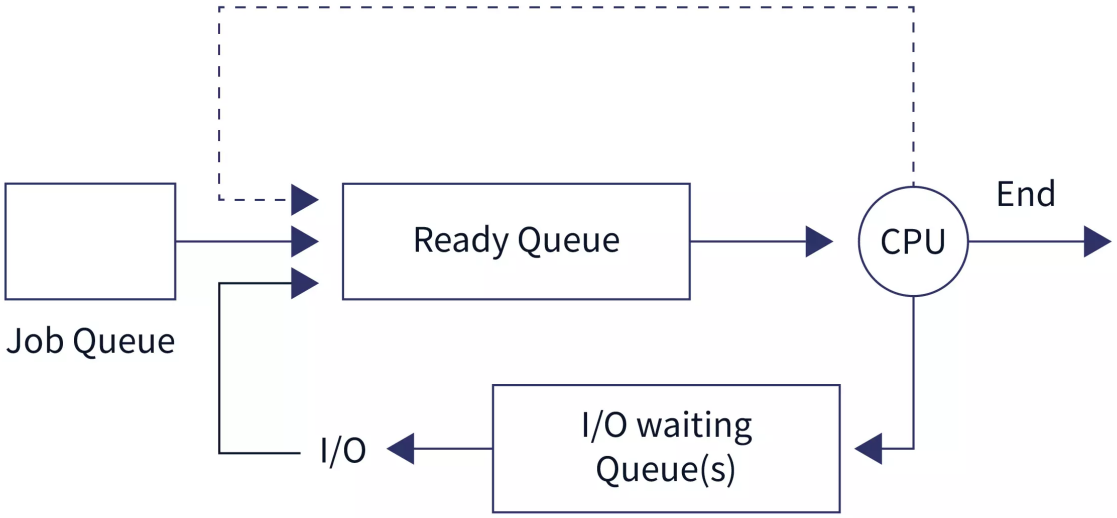
# Process Execution Cycle

- Process execution consists of a cycle of CPU execution and I/O wait.



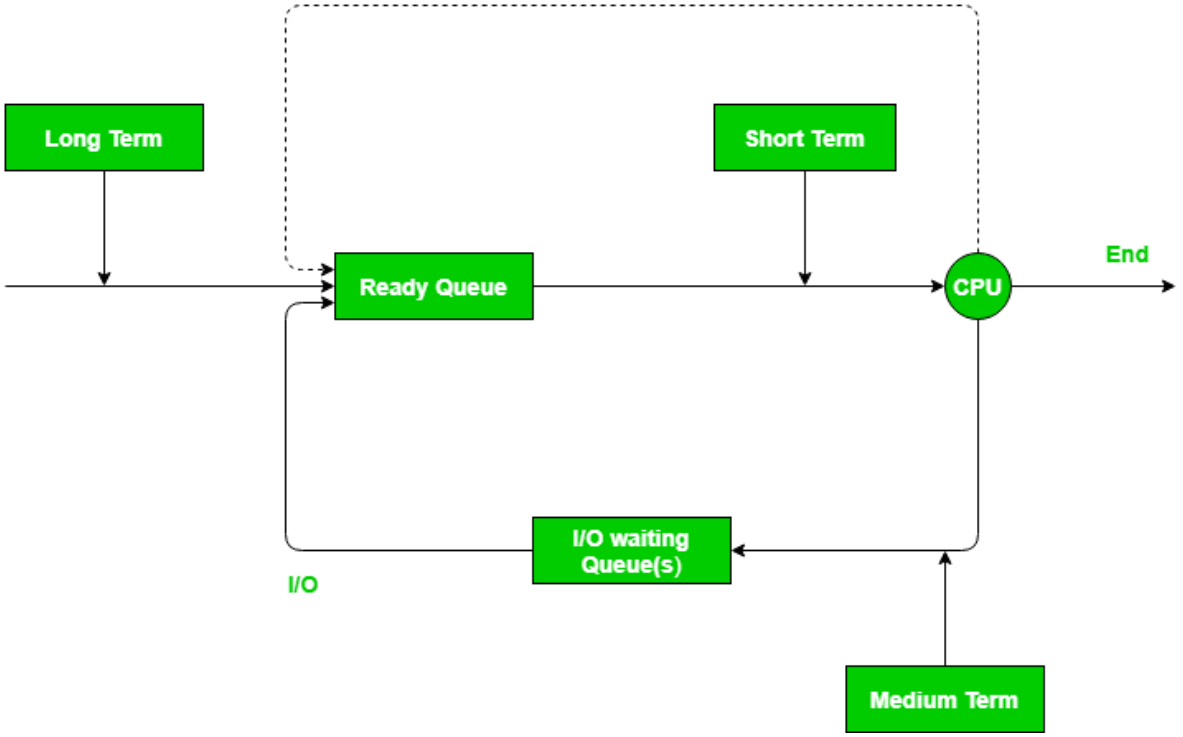
# CPU Scheduling Decisions

- Scheduling decisions occur when a process:
  - Switches from running to waiting (nonpreemptive)
  - Switches from running to ready (preemptive)
  - Switches from waiting to ready (preemptive)
  - Terminates (nonpreemptive)



# Dispatcher Module

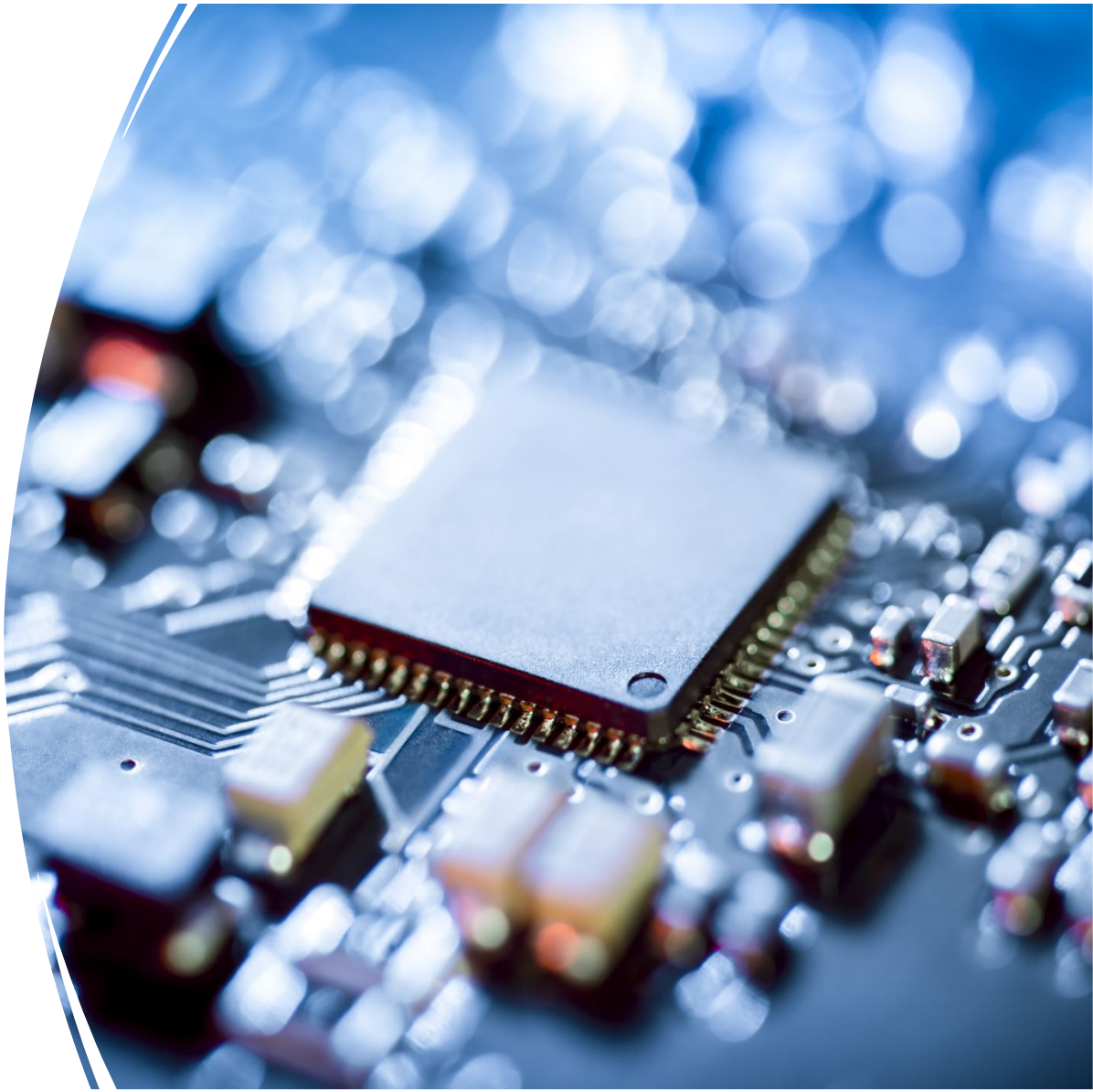
- The dispatcher module gives control of the CPU to the process selected by the short-term scheduler.
- **Dispatch latency**: the time it takes for the dispatcher to stop one process and start another.



# CPU Scheduling Algorithms

Many CPU scheduling algorithms can be classified into two categories:

- Non-preemptive
- Preemptive.



# Scheduling Algorithms (Continuation)

- Non-Preemptive Scheduling

Once a process starts its execution, it runs to completion without being interrupted by other processes.

**Examples:**

➢First-Come, First-Served (FCFS): Processes are executed in the order they arrive.

➢Shortest Job First (SJF): The process with the smallest burst time is executed next.

➢Priority Scheduling (Non-Preemptive): The highest priority process that is ready to run will execute until completion.

# Scheduling Algorithms (Continuation)

- Preemptive Scheduling

A running process can be interrupted (preempted) if a new process arrives with a higher priority or shorter remaining time.

Examples:

➢Round Robin (RR): Each process is assigned a fixed time slice and can be preempted after its time slice expires.

➢Shortest Remaining Time First (SRTF): A process is preempted if a new process arrives with a shorter remaining burst time.

➢Priority Scheduling (Preemptive): A higher priority process can interrupt a currently running lower priority process.


# Choosing between Preemptive and Non-Preemptive

The choice of whether to use a preemptive or non-preemptive algorithm often depends on the specific requirements of the system, such as response time for interactive processes or throughput for batch processes.

# Real-World Systems

Most modern operating systems use preemptive scheduling to ensure responsiveness, particularly in multi-tasking environments where multiple processes need to share CPU time effectively.

# Preemptive Scheduling Systems

- Windows OS: Uses preemptive scheduling to allow high-priority tasks to interrupt lower-priority ones for a responsive user experience.
- Linux OS: Employs a Completely Fair Scheduler (CFS) that allows tasks to be preempted, ensuring fairness and responsiveness.
- Mobile OS (Android/iOS): Both use preemptive scheduling to manage multiple applications, ensuring quick response to user interactions.

# Non-Preemptive Scheduling Systems

- Embedded Systems: Simple microcontrollers often use non-preemptive scheduling to execute tasks sequentially without interruptions.
- Batch Processing Systems: Mainframe systems process jobs in the order they arrive, using non-preemptive scheduling to utilize resources efficiently.
- Hard Real-Time Systems: Certain industrial control systems use non-preemptive scheduling to ensure critical tasks are completed in a strict order without interruption.

# Scheduling Algorithms (Continuation)

- Scheduling algorithms are chosen based on optimization criteria (e.g., throughput, turnaround time, etc.):
  - FCFS (First-Come, First-Served)
  - SJF (Shortest Job First)
  - Round Robin
  - Priority

# First Come First Serve (FCFS)

- **Definition**: Allocates CPU to the process that requests it first, using a FIFO queue.
- **Characteristics:**
- Executes tasks on a First-come, First-serve basis
- Easy to implement
- High wait time; less efficient in performance

# Example fcfs

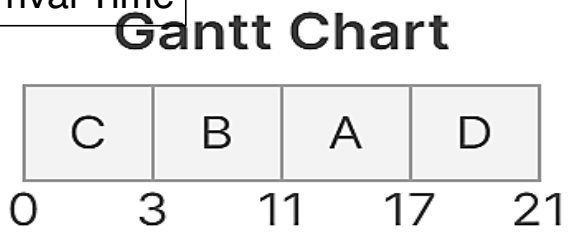
Let's consider an example where four processes with different burst times arrive at different times. Find the finish time, turnaround time, and waiting time for each process. Also, draw a Gantt chart.

| | Process | Burst Time | Arrival time |
|---|---|---|---|
| A | P1 | 6 | 2 |
| B | P2 | 8 | 1 |
| C | P3 | 3 | 0 |
| D | P4 | 4 | 4 |

# Answer

Waiting Time = Turnaround Time – Burst Time

Turnaround Time = Finish time – Arrival Time




| Job | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| C | 0 | 3 | 3 | 3 – 0 = 3 | 3 – 3 = 0 |
| B | 1 | 8 | 11 | 11 – 1 = 10 | 10 – 8 = 2 |
| A | 2 | 6 | 17 | 17 – 2 = 15 | 15 – 6 = 9 |
| D | 4 | 4 | 21 | 21 – 4 = 17 | 17 – 4 = 13 |

# Explanation

- P3 arrives first at time 0 and starts executing immediately. It has a burst time of 3.
- P2 arrives at time 1 but must wait until P3 finishes. Since FCFS does not allow preemption, P2 will have to wait until P3 completes its execution.
- P1 arrives at time 2 and also has to wait for P3 and then P2.
- P4 arrives at time 4 and waits for P3,P2, and P1 to finish before it can start.

# Shortest Job First (SJF)

- **Definition**: Selects the waiting process with the smallest execution time to execute next.
- **Characteristics:**
- Minimizes average waiting time among all scheduling algorithms
- Each task is associated with a unit of time to complete
- May cause starvation if shorter processes keep arriving (solvable with aging)

# Example Non-Preemptive SJF

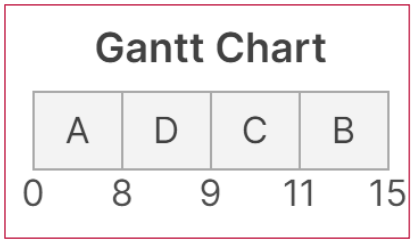
Let's consider an example where four processes with different burst times arrive at different times. Find the finish time, turnaround time, and waiting time for each process. Also, draw a Gantt chart.

| | Process | Arrival Time | Burst Time |
|---|---|---|---|
| A | P1 | 0 | 8 |
| B | P2 | 1 | 4 |
| C | P3 | 2 | 2 |
| D | P4 | 3 | 1 |

Waiting Time = Turnaround Time – Burst Time

Turnaround Time = Finish time – Arrival Time

# Answer




| Job | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| A | 0 | 8 | 8 | 8 | 0 |
| B | 1 | 4 | 15 | 14 | 10 |
| C | 2 | 2 | 11 | 9 | 7 |
| D | 3 | 1 | 9 | 6 | 5 |

# Explanation

- **At time 0**: **P1** is the only process that has arrived, so it starts execution.
- **At time 8**: **P1** completes. The remaining processes (P2, P3, P4) have arrived.
- Among P2 (4), P3 (2), and P4 (1), **P4** has the shortest burst time. It starts execution.
- **At time 9**: **P4** completes. The remaining processes are P2 and P3.
- **P3** has the next shortest burst time (2), so it starts execution next.
- **At time 11**: **P3** completes. Finally, **P2** starts execution.
- **At time 15**: **P2** completes.

# Preemptive Priority Scheduling

•**Definition**: Preemptive method based on process priority.
•**Mechanism**: Higher priority processes preempt lower priority ones. If priorities are equal, FCFS is used.
**Characteristics:**
•Schedules tasks based on priority.
•Higher priority process preempts lower priority one.
•Lower priority process is suspended until the higher priority task completes.
•Lower number = higher priority level.

# who gives the priority in a priority scheduling algorithm?

Defined by the Problem or System:

- **Gi**ven in the Problem Statement: In many scheduling problems, such as the one we're discussing, the priority of each process is provided as part of the problem's input data. This means the priority levels are predefined and given to you to solve the scheduling problem.
- **Assigned by the Operating System**: In real-world scenarios, priorities can be assigned by the operating system based on various criteria. These criteria might include the importance of the task, user-defined settings, or system policies.

# How Priorities Work?

- Priority Number: Each process is assigned a priority number. In most systems, a lower number indicates a higher priority.
- Scheduling Decision: The scheduler (a component of the operating system) uses these priority numbers to decide which process to run. The process with the highest priority (lowest number) is selected to run first.

# Example Preemptive Priority

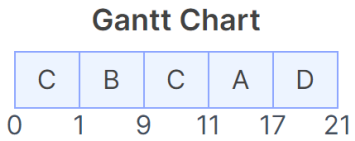
Let's consider an example where four processes with different burst times arrive at different times. Find the finish time, turnaround time, and waiting time for each process. Also, draw a Gantt chart.

| | Process | Burst Time | Arrival time | Prio |
|---|---|---|---|---|
| A | P1 | 6 | 2 | 3 |
| B | P2 | 8 | 1 | 1 |
| C | P3 | 3 | 0 | 2 |
| D | P4 | 4 | 4 | 4 |

Waiting Time = Turnaround Time – Burst Time

Turnaround Time = Finish time – Arrival Time

# Answer




| ID | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| C | 0 | 3 | 11 | 11 | 8 |
| B | 1 | 8 | 9 | 8 | 0 |
| A | 2 | 6 | 17 | 15 | 9 |
| D | 4 | 4 | 21 | 17 | 13 |

# Explanation

- **P3** starts at time 0 because it's the only process.
- **P2** arrives at time 1 and preempts P3 due to higher priority.
- **P2** continues running until it finishes at time 9.
- **P3** resumes from where it left off and runs from time 9 to 12.
- **P1** then starts at time 12 and runs until it finishes at time 18.
- **P4** finally runs from time 18 to 22.

# Traditional UNIX Scheduling

- Provides a clear illustration of how process priorities can be adjusted based on CPU usage.
- Formula:

$$\text{Priority} = \left(\frac{\text{recent CPU usage}}{2}\right) + \text{base}$$

- The scheduler aims to prevent any single process from monopolizing the CPU, thereby ensuring fairer distribution of CPU time among all processes.
- Modern systems employ advanced algorithms for fair and efficient scheduling.

# Round Robin Scheduling

- **Definition**: Each process is cyclically assigned a fixed time slot. Preemptive version of FCFS.
- **Focus**: Time Sharing technique.
- **Characteristics:**
- Simple, easy to use, and starvation-free.
- Widely used in CPU scheduling.
- Preemptive: Processes are given limited CPU time.

# Example Round Robin

Let's consider an example where four processes with different burst times arrive at different times. Find the finish time, turnaround time, and waiting time for each process. Also, draw a Gantt chart. Time Quantum = 4

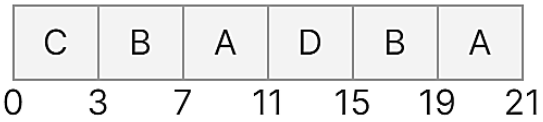
| | Process | Burst Time | Arrival time |
|---|---|---|---|
| A | P1 | 6 | 2 |
| B | P2 | 8 | 1 |
| C | P3 | 3 | 0 |
| D | P4 | 4 | 4 |

Waiting Time = Turnaround Time – Burst Time

Turnaround Time = Finish time – Arrival Time

# Answer

**Gantt Chart**




| Job | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| C | 0 | 3 | 3 | 3 – 0 = 3 | 3 – 3 = 0 |
| B | 1 | 8 | 19 | 19 – 1 = 18 | 18 – 8 = 10 |
| A | 2 | 6 | 21 | 21 – 2 = 19 | 19 – 6 = 13 |
| D | 4 | 4 | 15 | 15 – 4 = 11 | 11 – 4 = 7 |

# Explanation

- **At time 0**: P3 starts executing.
- Remaining Burst Time for P3 = 3 (finishes in the first time slice).
- **C** runs from 0 to 3.
- **At time 3**: P2 arrives (burst time = 8). P3 finishes, and now it's P2's turn.
- **P2** runs from 3 to 7 (4 time units used).
- Remaining Burst Time for P2 = 4.
- **At time 7**: P1 arrives (burst time = 6). Now it's P1's turn.
- **P1** runs from 7 to 11 (4 time units used).
- Remaining Burst Time for P1 = 2.
- **At time 11**: P4 arrives (burst time = 4).
- **P4** runs from 11 to 15 (finishing now).
- Remaining Burst Time for P4 = 0.
- **At time 15**: P2 is next.
- **P2** runs from 15 to 19 (finishing now).
- Remaining Burst Time for P2 = 0.
- **At time 19**: P1 is next.
- **P1** runs from 19 to 21 (finishing now).
- Remaining Burst Time for P1 = 0.

# Multilevel Queues

•**Structure**: Fixed multiple queues based on criteria (e.g., priority, process type).
•**Assignment**: Processes assigned to a queue remain there.

**Scheduling**:
•**Fixed Priority**: Higher priority queues are always chosen first.
•**Time Slicing**: Queues scheduled in a round-robin fashion.

**Advantages**: Simple and clear separation of processes.
**Disadvantages**: Inflexible, may cause starvation of lower-priority processes.

# Multilevel Feedback Queues (MLFQ)

- **Structure**: Multiple dynamic queues with feedback.
- **Assignment**: Processes can move between queues based on behavior.

**Scheduling**:
- **Feedback**: Processes using too much CPU time move to lower-priority queues; long-waiting processes move to higher-priority queues.
- **Aging**: Prevents starvation by increasing priority of long-waiting processes.

**Advantages**: Flexible, dynamic, fair CPU time allocation.
**Disadvantages**: Complex to implement and manage.

# Multiprocessor Scheduling

**Overview**:

- Manages processes across multiple CPUs/cores for optimal performance.

**Types**:

## 1.Asymmetric Multiprocessing (AMP):

1. **One master processor** handles all system tasks.
2. **Pros**: Simpler, reduced complexity.
3. **Cons**: Bottleneck risk, underutilization of other processors.

## 2.Symmetric Multiprocessing (SMP):

1. **All processors** share memory and I/O, each with its own scheduler.
2. **Pros**: Balanced load, better performance.
3. **Cons**: More complex, potential resource contention.

# Little's Formula

- Helps determine average wait time per process in any scheduling algorithm:
  - $n=\lambda \times W$
  - $n$ = avg queue length; $W$ = avg waiting time in queue; $\lambda$ = average arrival rate into queue.
- Essential for understanding and optimizing queuing systems across various applications.
- Applications: Call Centers, Computer Networks, and Manufacturing

# Example

- **Scenario:** A bank teller system where customers arrive at an average rate of 10 per hour ($\lambda = 10$ customers/hour) and each customer spends an average of 6 minutes (0.1 hours) in the system ($W = 0.1$ hours).
- **Calculation**: $L = \lambda \cdot W = 10 \cdot 0.1 = 1$
- **Interpretation**: On average, there is 1 customer in the system.

# Simulations

- **Definition**: Programmed models of a computer system with variable clocks, mimicking real-world processes.
- **Purpose**: Gather statistics indicating algorithm performance.
- **Advantages**: **High Accuracy**: More accurate than queuing models (e.g., Little's Law).
- **Disadvantages**: **High Cost & Risk**: Resource-intensive and complex to manage.

# Comparison to Queuing Models

- **Little's Law**: Quick, approximate analysis; easier and cheaper, but less detailed. Queuing Models are suited for simpler, stable environments
- **Simulations**: Detailed insights and high accuracy; require significant resources. Simulations are ideal for complex, dynamic scenarios.

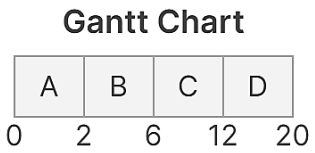
# Activity

Let's consider an example where four processes with different burst times arrive at different times. Find the finish time, turnaround time, and waiting time for each process. Also, draw a Gantt chart and give me the average turnaround time and average waiting time. Please do it for FCFS and RR. Time Quantum = 3.

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 2 |
| P2 | 2 | 4 |
| P3 | 4 | 6 |
| P4 | 6 | 8 |

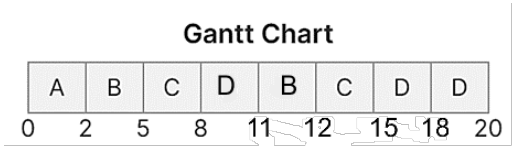
# Answer

## FCFS




| Job | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| A | 0 | 2 | 2 | 2 | 0 |
| B | 2 | 4 | 6 | 4 | 0 |
| C | 4 | 6 | 12 | 8 | 2 |
| D | 6 | 8 | 20 | 14 | 6 |
| Average | | | | 28 / 4 = 7 | 8 / 4 = 2 |

## RR




| Job | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| A | 0 | 2 | 2 | 2 | 0 |
| B | 2 | 4 | 12 | 10 | 6 |
| C | 4 | 6 | 15 | 11 | 5 |
| D | 6 | 8 | 20 | 14 | 6 |
| Average | | | | 9.25 | 4.25 |

# Process Synchronization

# What is process synchronization?

Mechanism to ensure that multiple processes or threads can execute concurrently without conflicting with each other, especially when accessing shared resources.

**Goal:** avoid race conditions, ensure data consistency, and coordinate the execution order of processes.

# Race Condition

- A race condition occurs when multiple processes or threads read and write shared data concurrently, and the final outcome depends on the sequence of their execution.
- This can lead to unpredictable behavior and bugs that are hard to reproduce and debug.

# Example

Consider two threads trying to increment a shared counter:

```
int counter = 0;
void increment() {
    counter++;
}
```

If **both threads read the counter's value simultaneously**, say counter = 0, and then both increment it, the final value might be 1 instead of 2 because they might **overwrite each other's increments**.

# Critical Section

- A critical section is a segment of code that accesses shared resources (like data structures or hardware devices) that must not be concurrently accessed by more than one thread or process.
- The main goal is to prevent data inconsistency and corruption.

# Example

Imagine a bank account shared between multiple ATMs. The critical section here would be the part of the code that updates the account balance. If two ATMs try to update the **balance simultaneously without proper synchronization**, it might lead to **inconsistent balance updates.**

```
int balance = 1000; // Shared resource

void withdraw(int amount) {
    pthread_mutex_lock(&mutex); // Enter critical section
    if (balance >= amount) {
        balance -= amount;
    }
    pthread_mutex_unlock(&mutex); // Exit critical section
}
```

# Mutual Exclusion

- Mutual exclusion (often shortened to "mutex") is a principle used to prevent race conditions by ensuring that only one process or thread can access the critical section at any one time.
- This guarantees that the critical section is executed atomically.

# Example

Using a mutex lock to ensure mutual exclusion:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void increment() {
    pthread_mutex_lock(&mutex); // Lock to enter critical section
    counter++;
    pthread_mutex_unlock(&mutex); // Unlock to exit critical section
}
```

In this example, **pthread_mutex_lock ensures that only one thread can increment the counter at a time**, thus **preventing race conditions**.

# Peterson's Solution for Mutual Exclusion

- **Goal**: Ensure only one of two processes enters the critical section at a time.

**Pros**

- Simple and elegant.
- No complex atomic operations or special hardware needed.
- Good theoretical foundation for mutual exclusion.

**Cons**

- Limited to two processes.
- Busy waiting wastes CPU cycles.
- Not suitable for modern multiprocessor systems.

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

**turn:** whose turn it is to enter the critical section
**flag:** indication of whether or not a process is ready to enter critical section

# Synchronization Mechanisms

**Locks:**

- **Mutex**: Provides mutual exclusion; only one thread can lock at a time.
- **Spinlock**: Threads spin in a loop, checking for lock availability; efficient for short waits.

**Semaphores:**

- **Binary Semaphore**: Like a mutex; can be 0 or 1.
- **Counting Semaphore**: Manages access for multiple threads by counting available resources.

**Monitors:** High-level construct with a lock and condition variables; only one thread can execute at a time.

**Condition Variables:** Used with mutexes to let threads wait for specific conditions.

**Barriers:** Threads wait until all reach a synchronization point before continuing.

# Solution Using Locks “Mutexes”

- **Implementation:** Only one thread can acquire the lock and enter the critical section at a time.
- **Status Indicators:**
  - **Lock = 0:** Critical section is vacant (initial value).
  - **Lock = 1:** Critical section is occupied.

```
do {
    acquire lock;
    // Critical section
    release lock;
    // Remainder section
} while (TRUE);
```

# Modern Machines: Atomic Hardware Instructions

- **Definition**: Atomic operations are non-interruptible and complete in a single step.

**Examples of Atomic Instructions**

- **Test-and-Set**: Sets a value and returns the old value.
- **Compare-and-Swap (CAS)**: Compares a variable to an expected value and swaps it with a new value if they match.
- **Fetch-and-Add**: Adds a value to a variable and returns the old value.

**Benefits**

- **Efficient Synchronization**: Avoids complex locking mechanisms.
- **Non-Interruptible**: Ensures operations are complete without interruption, preventing race conditions.

# Solution Using Test-And-Set

- **Lock Variable:** Initialized to false.
- **Algorithm:** Returns current value and sets lock to true.
- **Critical Section Entry:**
  - **First Process:** Enters immediately (TestAndSet(lock) returns false).
  - **Others:** Wait (lock is true).

```
do {
    while (TestAndSet(&lock))
        ; // Do nothing
    // Critical section
    lock = FALSE;
    // Remainder section
} while (TRUE);
```

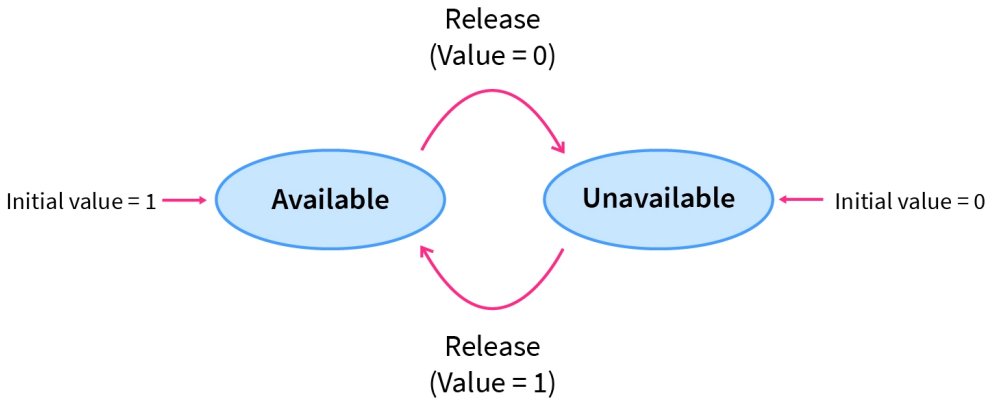
# Solution Using Swap

- **Mechanism:**
  - **Initial: key = true**, then swap with **lock**.
  - **Execution:**
    - **First Process:**
      - **key = false**, **lock = true**
      - Enters critical section (**while(key)** breaks).
    - **Others:**
      - **key = true**, **while(key)** continues (lock remains true).

```
do {
    &key;
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);
    // Critical section
    lock = FALSE;
    // Remainder section
} while (TRUE);
```

# Semaphore

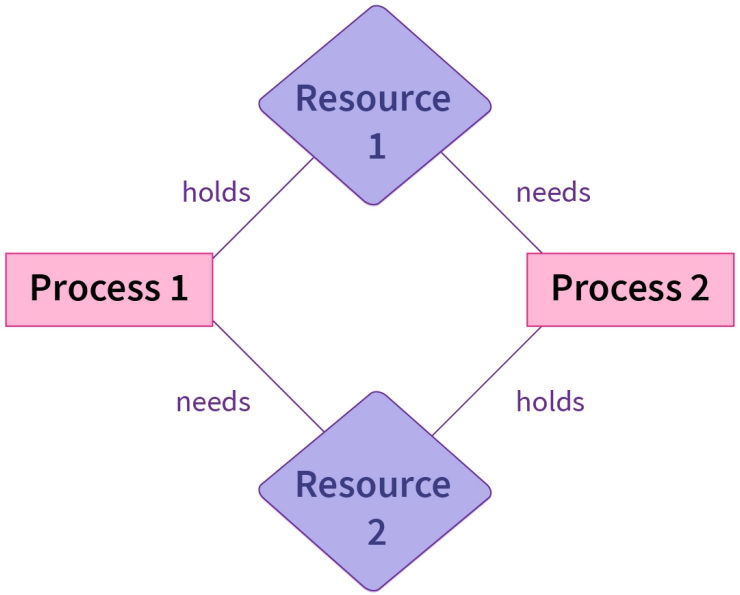
- Synchronization tool that doesn't require busy waiting.
- **Operations:** Wait () -> Decreases value; if negative, waits. & Signal () -> Increases value; wakes up waiting processes/threads.
- Can be counting or binary.

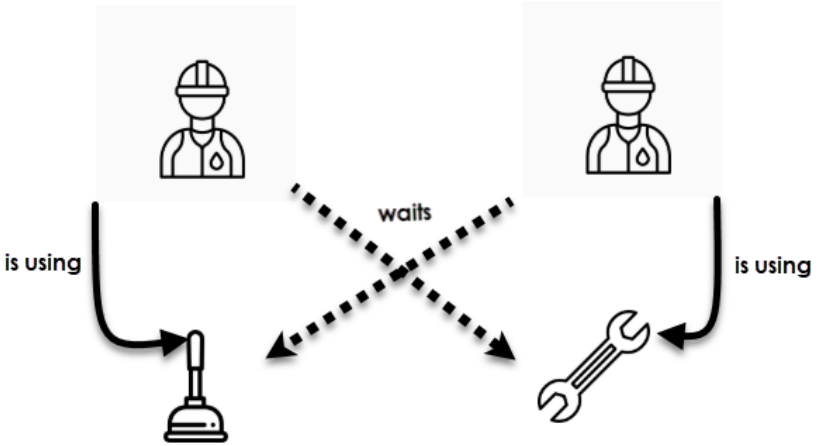



```
Semaphore mutex(1); // Binary semaphore

// Thread 1
mutex.Wait(); // Enter critical section
// Critical Section
mutex.Signal(); // Exit critical section

// Thread 2
mutex.Wait(); // Enter critical section
// Critical Section
mutex.Signal(); // Exit critical section
```

# Deadlock

When two or more processes wait indefinitely for each other to release resources, causing all of them to be stuck.



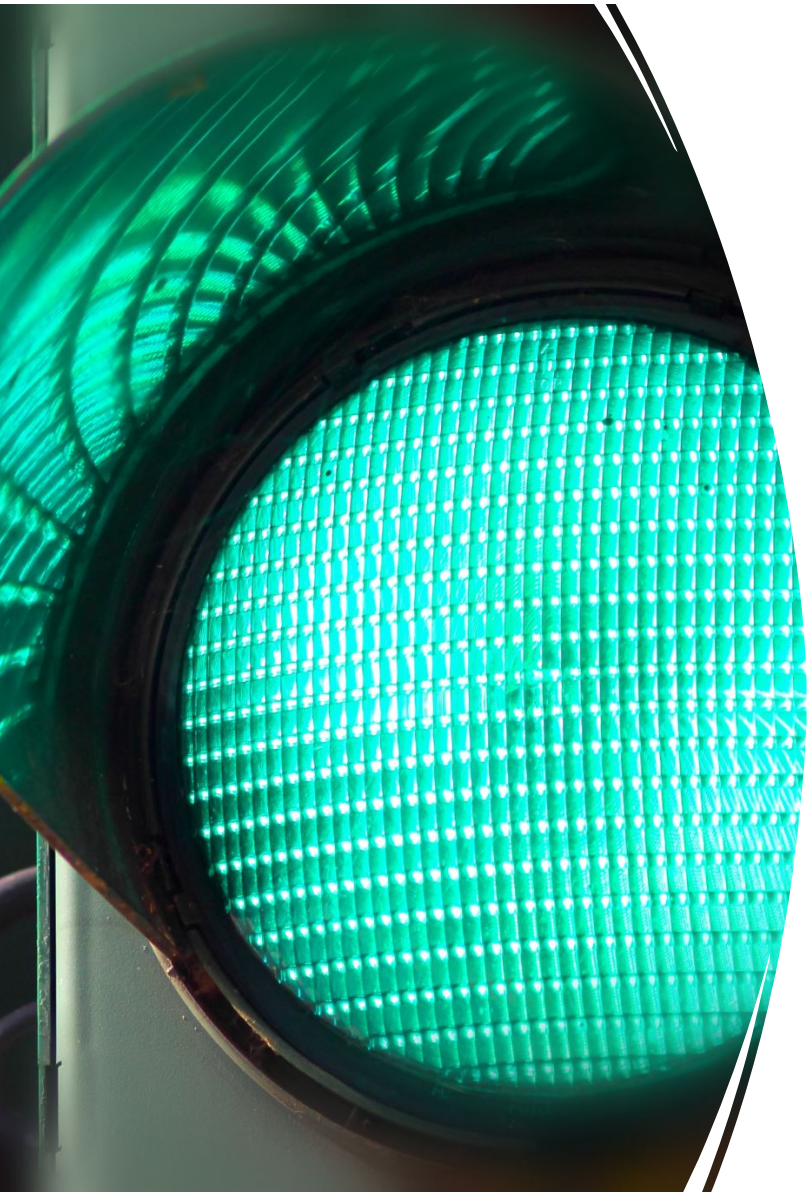
# Conditions

- **Mutual Exclusion:** Only one process can use a resource at a time.
- **Hold and Wait:** Processes hold resources while waiting for more.
- **No Preemption:** Resources can't be forcibly taken away from processes.
- **Circular Wait:** A set of processes are waiting for each other in a circular chain.

# Necessary Conditions for Deadlock (Traffic Analogy)

# Mutual Exclusion (One-lane Bridge)

- **Explanation:** Only one car can be on a one-lane bridge at a time.
- **Traffic Analogy:** If a car is on the bridge, no other car can enter the bridge until the first car exits.



# Hold and Wait (Intersection with Traffic Lights)

- **Explanation:** Cars already in the intersection hold their position and wait for the light to turn green.
- **Traffic Analogy:** A car is waiting at a red light, holding its place in the intersection while waiting for the green light to proceed. Meanwhile, it blocks the way for other cars that need to pass through the intersection.

# No Preemption (Toll Booth)

- **Explanation:** Once a car is paying at a toll booth, it cannot be forced to leave until it finishes the transaction.
- **Traffic Analogy:** A car at the toll booth can't be pushed out of the way by another car; the waiting cars must wait until the car at the booth finishes paying and moves on.

# Circular Wait (Roundabout with Four Cars)

- **Explanation:** Each car in a circular roundabout is waiting for the next car to move before it can proceed.
- **Traffic Analogy:** Four cars are in a roundabout, each waiting for the car in front to move. Car A is waiting for Car B, Car B is waiting for Car C, Car C is waiting for Car D, and Car D is waiting for Car A. None of the cars can move because they are all waiting for each other, creating a deadlock.

# What is Starvation in OS?

- **Definition:** A process never gets the resources it needs to run because other processes keep getting those resources.
- **Cause:** Typically occurs in priority-based scheduling systems where high-priority requests are processed first.
- **Problems Caused:** Reduced system responsiveness, Violation of response time guarantees, and Potential system failure.

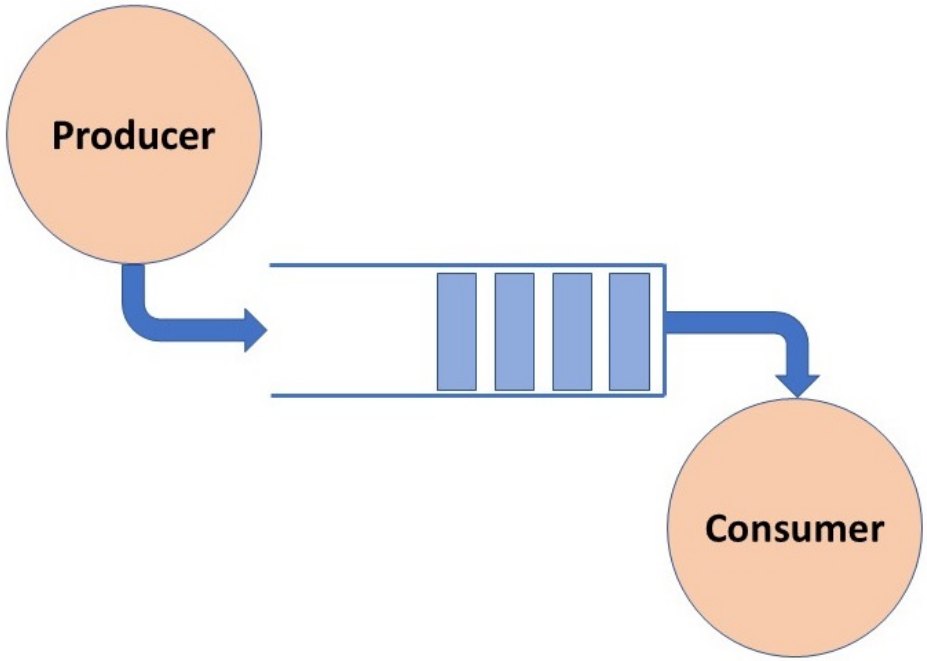
# Concept of Priority Aging

- **Definition:** A scheduling approach used to prevent starvation.
- **Mechanism:** As a process waits in the system, its priority gradually increases.
- **Purpose:** Ensures fair resource distribution by improving the priority of processes that have been waiting longer, reducing the chance of indefinite waiting.

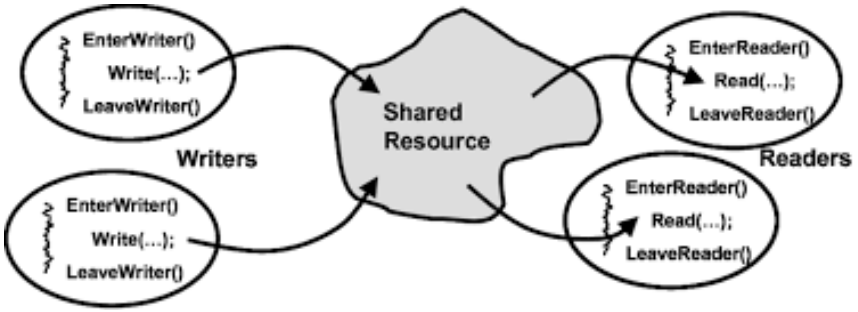
# Classic Synchronization Problems

**Producer-Consumer Problem:**

- Involves synchronization between producer and consumer threads that share a common buffer.
- The producer adds items to the buffer, and the consumer removes items from it.
- **Solution**: Use semaphores to track the number of empty and full slots and a mutex for mutual exclusion.



# Classic Synchronization Problems



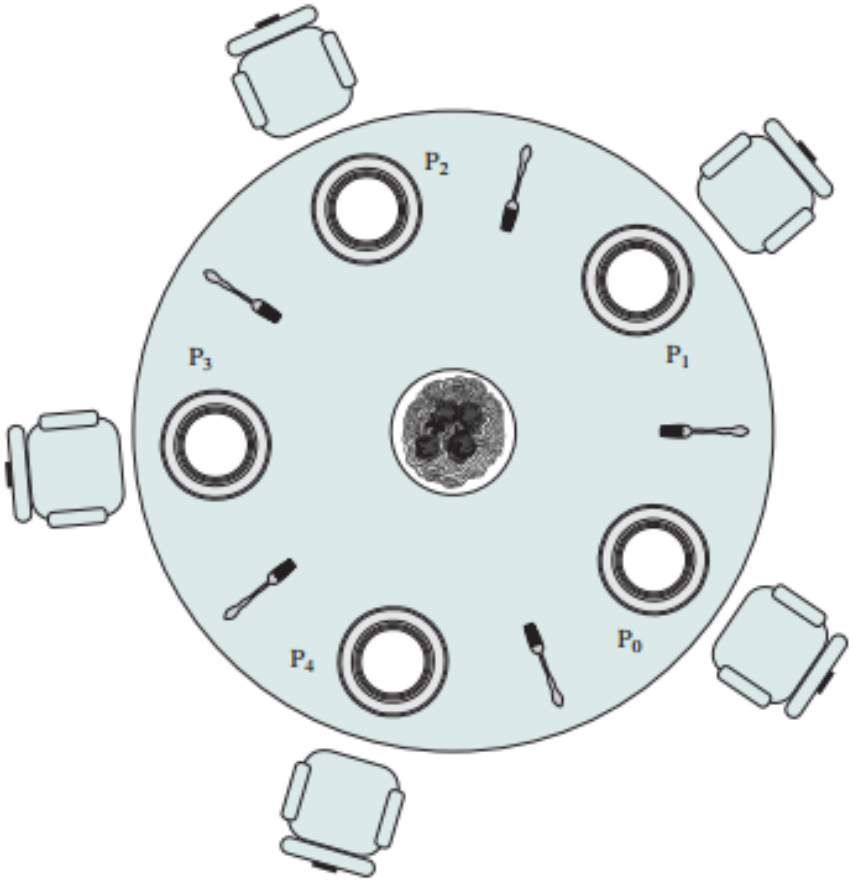

**Readers-Writers Problem:**

- Involves synchronization between readers and writers of a shared resource.
- Multiple readers can read simultaneously, but writers require exclusive access.
- **Solution**: Use read and write locks to manage access.

# Classic Synchronization Problems

**Dining Philosophers Problem**:

- Models five philosophers sitting at a table, each needing two forks to eat.
- The challenge is to design a protocol such that no philosopher starves.
- **Solution**: Various approaches like resource hierarchy, waiter, or Chandy/Misra solution.



Another example of deadlock that is not related to a computer system environment.

**Resource Allocation in Businesses:**

- Two departments wait on resources held by the other, such as a printer and a scanner, causing a deadlock in operations.

# Implementing Synchronization in OS

Operating systems provide several APIs and constructs for synchronization:

1. **POSIX Threads (pthreads)**:
   - Functions like pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_wait, and pthread_cond_signal.
2. **Windows Threads**:
   - Functions like CreateMutex, WaitForSingleObject, ReleaseMutex, InitializeCriticalSection, EnterCriticalSection, and LeaveCriticalSection.
3. **Java**:
   - synchronized keyword, wait, notify, and notifyAll methods for object synchronization.

# Activity: Understanding the Sleeping Barber Problem

**Scenario:**

You own a small barber shop with one barber, one barber chair, and a waiting room with three chairs. When a customer arrives, they will either take a seat in the waiting room or leave if no chairs are available. If the barber is idle and a customer arrives, the barber begins cutting their hair immediately. Once a customer's hair is cut, they leave, and the barber checks for waiting customers before either starting on the next customer or going to sleep if none are waiting.

**Instructions:**

1. **Read the scenario carefully**: Understand the flow of events in the barber shop.
2. **Answer the questions**: Think critically about the synchronization challenges and propose solutions.

# Questions

1. Describe the main entities in this problem and their roles.
2. What synchronization issues can arise in this scenario?
3. How would you use semaphores to manage access to the barber chair and the waiting room chairs?
4. Explain how the barber transitions between sleeping, cutting hair, and checking for waiting customers.
5. What happens if a customer arrives and all waiting room chairs are occupied?
6. How would you ensure that the barber does not starve and gets time to rest?
7. Discuss potential improvements to this basic synchronization scheme.

## Describe the main entities in this problem and their roles.

**Entities**: Barber, Customers, Barber Chair, Waiting Room Chairs.

**Roles**:

- The **Barber** cuts hair and sleeps if there are no customers.
- **Customers** come to get a haircut, wait if necessary, or leave if the waiting room is full.
- The **Barber Chair** is where haircuts take place.
- **Waiting Room Chairs** are where customers wait if the barber chair is occupied.

## What synchronization issues can arise in this scenario?

—

- **Race Conditions**: Multiple customers arriving at the same time.
- **Deadlock**: Barber waiting for customers, while customers leave due to perceived unavailability.
- **Starvation**: If customers continually arrive when the barber is about to sleep, the barber might never get a chance to rest.

How would you use semaphores to manage access to the barber chair and the waiting room chairs?

—

- **Barber Chair Semaphore**: Initialize to 1 to ensure only one customer can be in the chair.
- **Waiting Room Semaphore**: Initialize to the number of waiting room chairs (3 in this case) to limit the number of customers waiting.
- **Customer Semaphore**: Incremented when a customer arrives and decremented when the barber starts a haircut, signaling the barber when customers are present.
- **Barber Semaphore**: Signaled when the barber is ready to cut hair, ensuring synchronization between customer arrival and barber's readiness.

Explain how the barber transitions between sleeping, cutting hair, and checking for waiting customers.

—

- The barber starts by checking the **Customer Semaphore**:
- If the semaphore indicates customers are waiting, the barber proceeds to decrement the **Waiting Room Semaphore** and increments the **Barber Semaphore**.
- If no customers are waiting, the barber goes to sleep (blocks on the **Customer Semaphore**).
- When a customer arrives, they increment the **Customer Semaphore**. If the barber is sleeping, this wakes the barber.
- The barber then performs the haircut, and upon finishing, checks the **Customer Semaphore** again to see if more customers are waiting.

## What happens if a customer arrives and all waiting room chairs are occupied?

—

- The customer checks the **Waiting Room Semaphore**:
  - If the semaphore value is zero (all chairs are occupied), the customer leaves the shop.
  - Otherwise, the customer takes a seat in the waiting room (decrements the semaphore) and waits for their turn.

How would you ensure that the barber does not starve and gets time to rest?

—

- Use a condition where after a certain number of haircuts or after a set period, the barber takes a mandatory break.
- Implement a fair scheduling policy to prevent the barber from continuously working without rest due to a steady stream of customers.

Discuss potential improvements to this basic synchronization scheme.

—

- **Priority Queue**: Implement a priority queue for customers, prioritizing those who have waited longer.
- **Timed Semaphores**: Use timed semaphores to allow the barber to rest periodically.
- **Additional Barbers**: If the shop is busy, consider adding more barbers and expanding the synchronization mechanism to handle multiple barbers and chairs.

Write pseudocode to synchronize the barber and the customers using semaphores.

—

**Pseudocode Requirements:**

1. Use semaphores to manage the barber chair and the waiting room chairs.
2. Ensure that the barber goes to sleep when no customers are present and wakes up when a customer arrives.
3. Ensure that customers leave if no waiting room chairs are available.
4. Prevent race conditions and ensure mutual exclusion where necessary.

# Pseudocode Structure: Barber Process

```
initialize semaphore barberReady to 0
initialize semaphore accessWRSeats to 1
initialize semaphore customers to 0
initialize integer waitingCustomers to 0
initialize integer numberOfSeats to 3

procedure Barber()
    while true do
        wait(customers)             // Wait for a customer to arrive
        wait(accessWRSeats)         // Acquire access to waiting room seats
        waitingCustomers = waitingCustomers - 1
        signal(barberReady)         // Notify a customer that barber is ready
        signal(accessWRSeats)       // Release access to waiting room seats
        // Cut hair
        cutHair()
end procedure

procedure cutHair()
    // Barber cuts the customer's hair
end procedure
```

# Pseudocode Structure: Customer Process

```
procedure Customer()
    wait(accessWRSeats)            // Acquire access to waiting room seats
    if waitingCustomers < numberOfSeats then
        waitingCustomers = waitingCustomers + 1
        signal(customers)          // Notify the barber that there is a customer
        signal(accessWRSeats)      // Release access to waiting room seats
        wait(barberReady)          // Wait for the barber to be ready
        // Get hair cut
        getHairCut()
    else
        signal(accessWRSeats)      // Release access to waiting room seats
        // Leave the shop as no seats are available
        leaveShop()
end procedure

procedure getHairCut()
    // Customer gets their hair cut
end procedure

procedure leaveShop()
    // Customer leaves the shop because no waiting room seats are available
end procedure
```

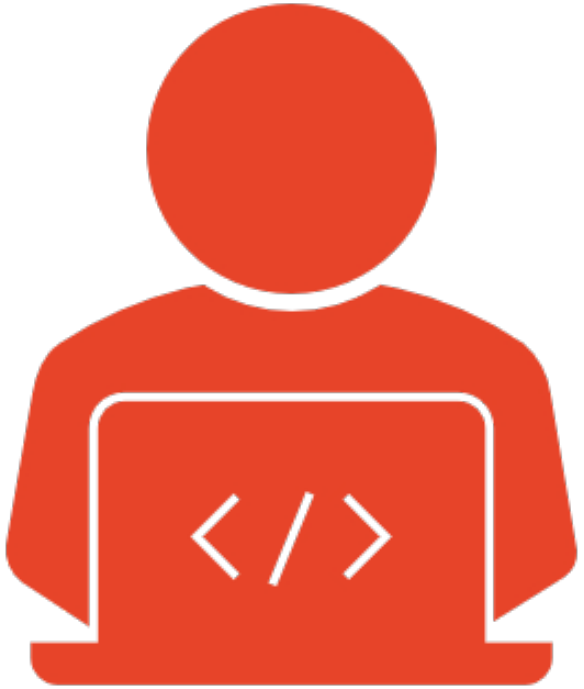
# Best Practices

- **Minimize Locking**: Lock only when necessary and keep the critical section as short as possible.
- **Avoid Deadlocks**: Ensure that locks are always acquired and released in a consistent order.
- **Use High-Level Abstractions**: Where possible, use higher-level synchronization constructs like monitors or condition variables instead of low-level locks.

# Understanding Threading

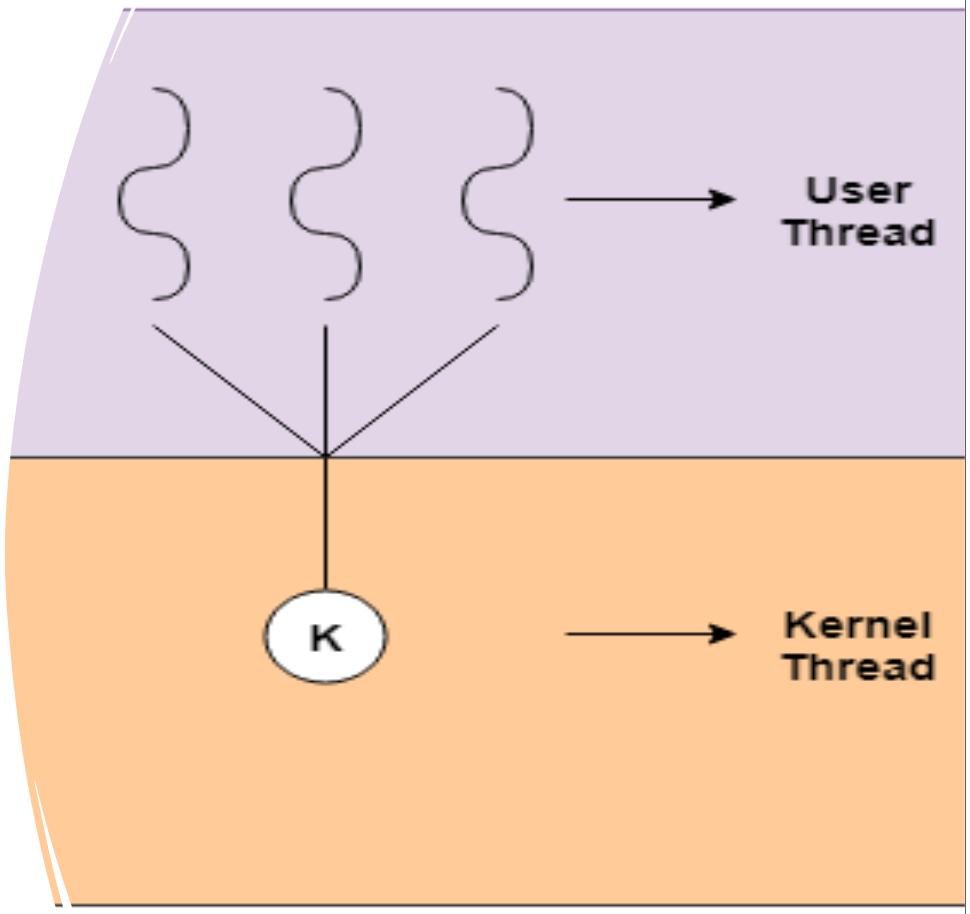
# Introduction to Threads

- **Definition:** A thread refers to a single sequential flow of activities being executed in a process
- **Shares:** information like data segment, code segment files, etc. with its peer threads.
- **Contain:** its registers, stack, counter, etc.
- **Significance:** Simplifies code and increases efficiency.



# Types of Threads

The two main types of threads are user-level threads and kernel-level threads.



# User - Level Threads

- User-level threads are implemented by users.
- The kernel is unaware of the existence of user-level threads.
- The kernel treats user-level threads as single-threaded processes.
- User-level threads are small and much faster than kernel-level threads.
- Represented by: Program counter (PC), Stack, Registers, and Small process control block
- No kernel involvement in synchronization for user-level threads.

# Kernel-level threads

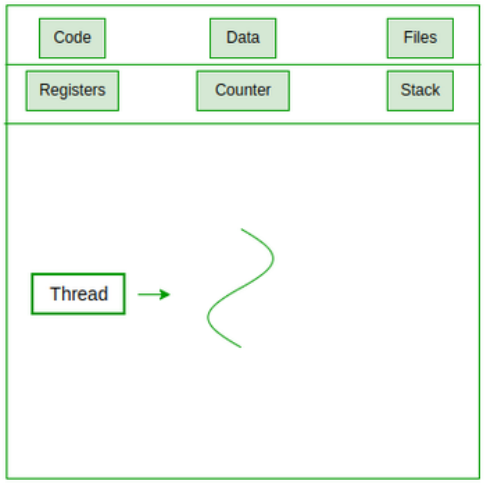
- Kernel-level threads are handled by the operating system directly.
- Thread management is done by the kernel.
- The kernel manages context information for both the process and its threads.
- Kernel-level threads are slower than user-level threads due to kernel management.

# Advantages and Disadvantages of User-level & Kernel-Level Threads

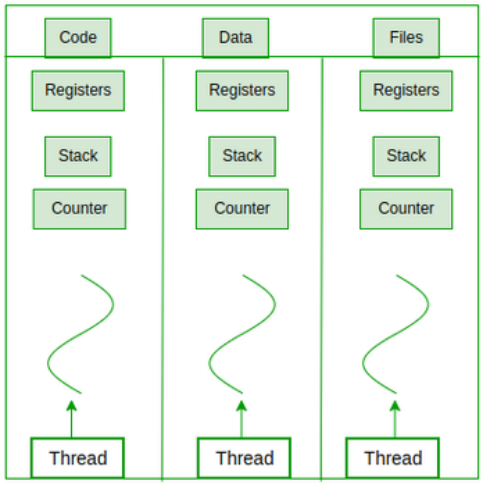
| ASPECT | USER-LEVEL THREADS | KERNEL-LEVEL THREADS |
|---|---|---|
| Advantages | - Faster | - Only blocks the calling thread |
| | - More resource-efficient | - Kernel manages scheduling and resources |
| | - Quicker context switching | - Easier synchronization |
| | - Simpler to implement | - Easier to debug |
| Disadvantages | - Blocks entire process on system call | - Slower |
| | - Kernel is unaware | - Less resource-efficient |
| | - Complex synchronization | - Slower context switching |

# Multi-Threading

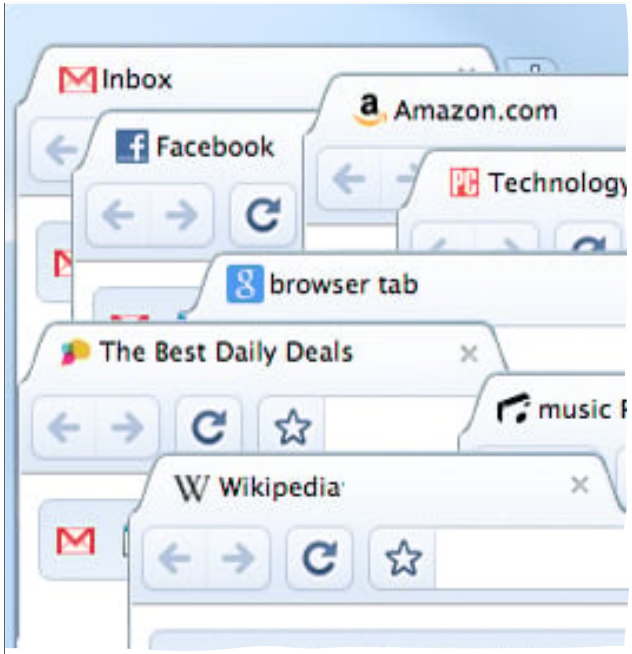
The ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer.

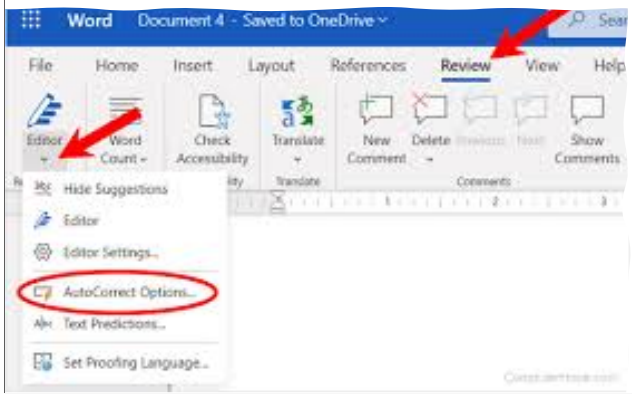



Single Threaded Process




Multi Threaded Process



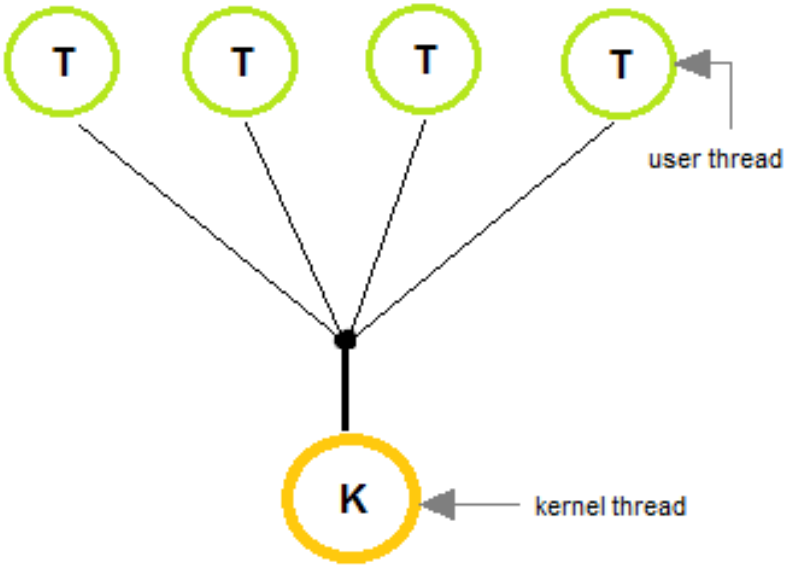




# Example of multi-threading

- In a browser, each tab can be a different thread.
- MS Word employs multiple threads: one for text formatting, another for processing inputs, etc.

# Multi-Threading Models

- **Many-to-One:** Many user-level threads mapped to a single kernel thread.
- **One-to-One:** Each user-level thread maps to a kernel thread.
- **Many-to-Many:** Many user-level threads mapped to many kernel threads.

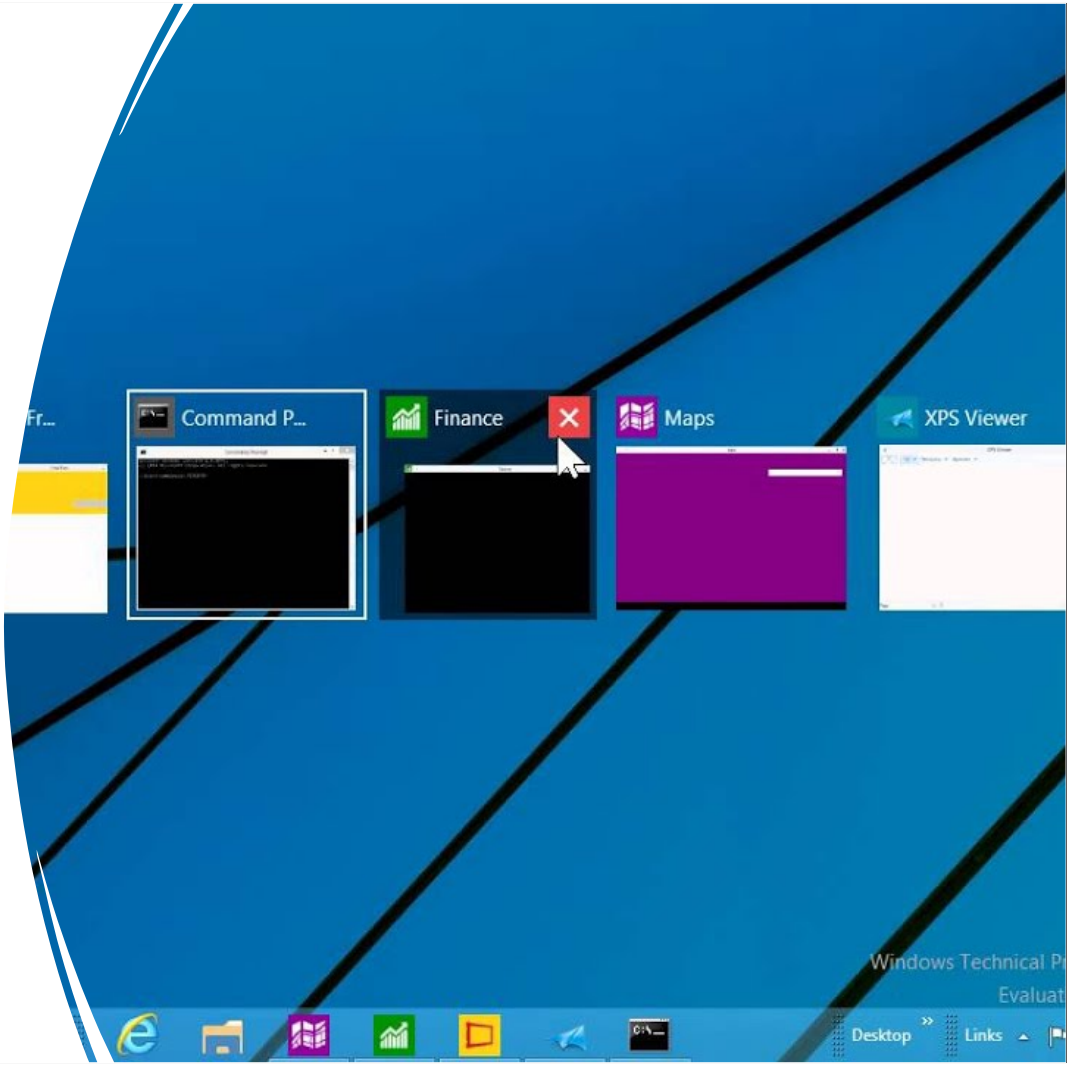


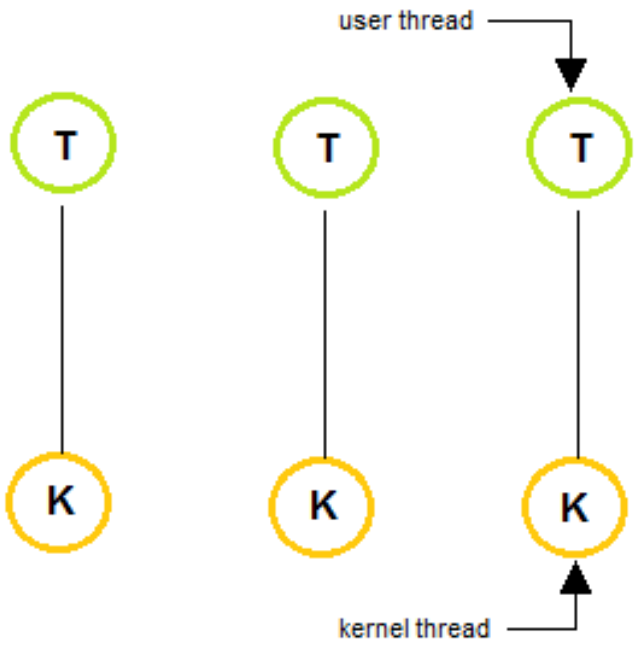

# Many-to-One Model

- **Description:** Many user-level threads mapped to a single kernel thread.
- **Advantage:** Simple to implement.
- **Disadvantage:** If the kernel thread blocks, all user-level threads are blocked.

# eXAMPLE

Imagine you are working on your computer. You can have several programs open at the same time—like a word processor, a web browser, and a music player. All of these programs are running under your user account. So, one user can have many applications running.

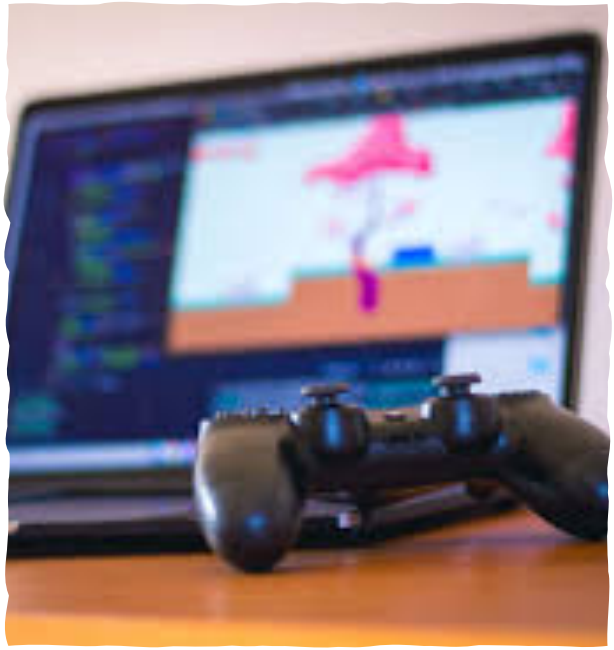




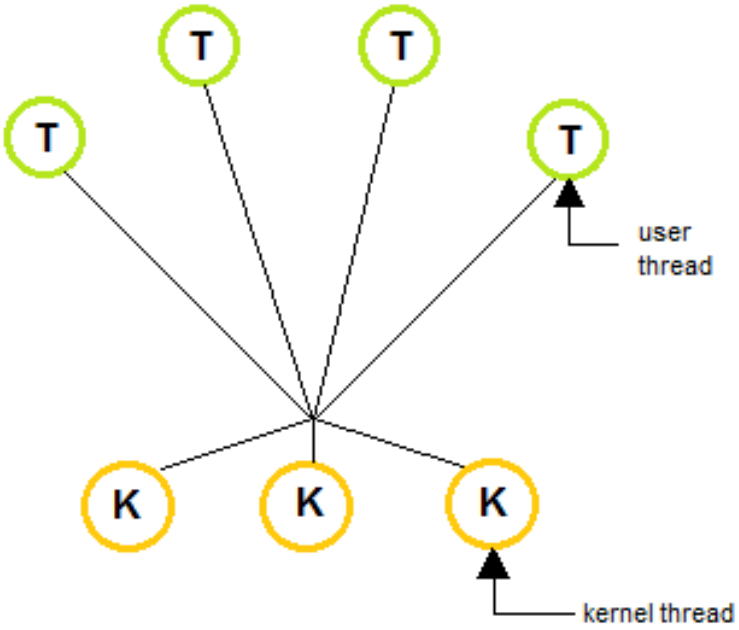

# One-to-One Model

- **Description:** Each user-level thread maps to a kernel thread.
- **Advantage:** More concurrency as other threads can run if one thread is blocked.
- **Disadvantage:** Creates overhead due to large number of kernel threads.

# EXAMPLE



Think of a process as a task or job your computer is doing, like running a game or a web browser. Each of these tasks has a unique “ID card” called a PCB that keeps track of everything about that task, like how much memory it’s using or what it’s doing right now. So, one process gets one PCB.




# Many-to-Many Model

- **Description:** Many user-level threads mapped to many kernel threads.
- **Advantage:** Combines the benefits of both models, allowing better resource utilization.
- **Disadvantage:** More complex to implement.

# EXAMPLE

Think of files on your computer (like documents or pictures). Different users can have different kinds of access to these files—some might be able to read them, some might be able to write to them, and some might not have access at all. And, each user can have different access to many files. So, many files can be accessed by many users in various ways.

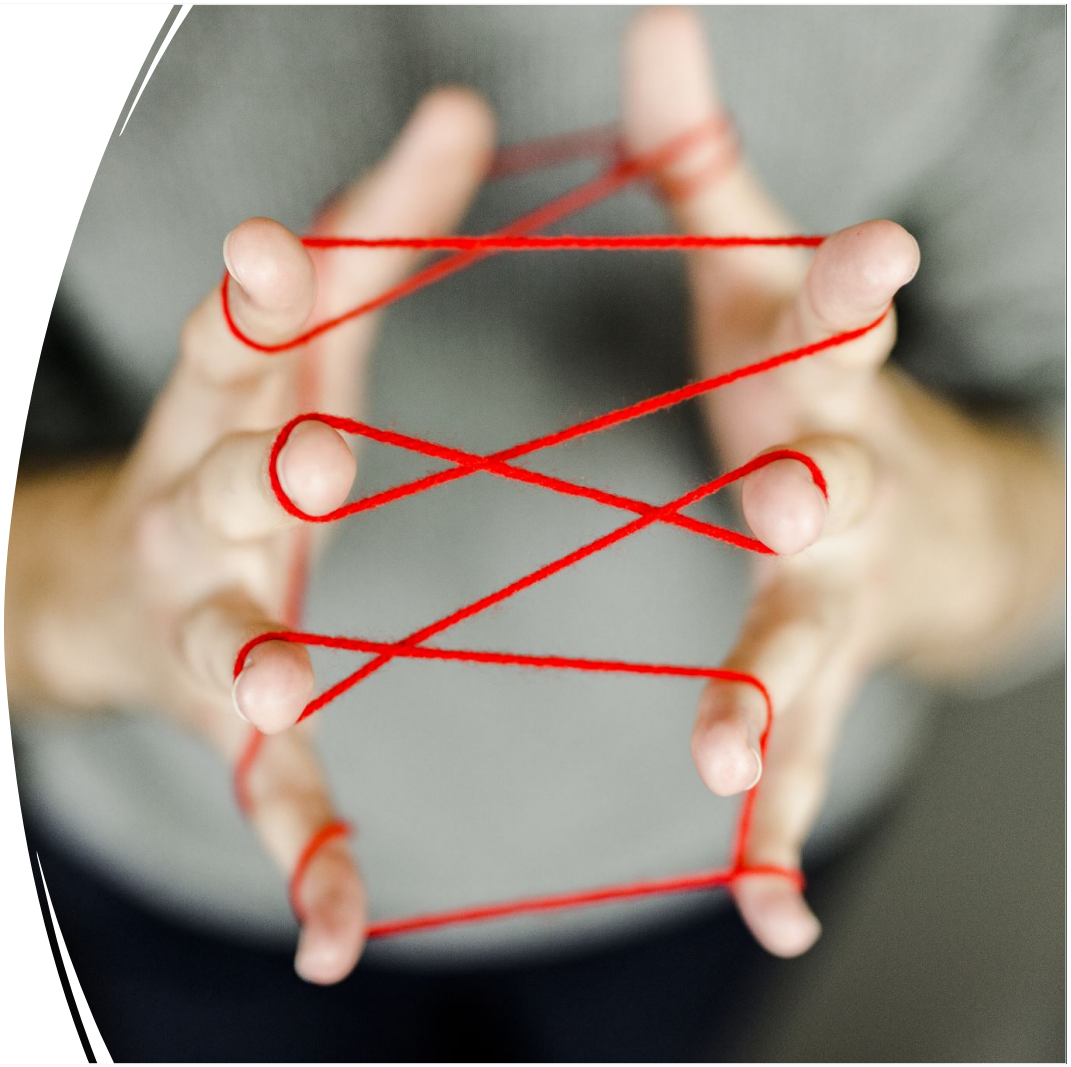
# Thread Libraries

**PURPOSE:**
PROVIDE PROGRAMMERS WITH AN API FOR CREATING AND MANAGING THREADS

**EXAMPLES:**
POSIX PTHREADS, JAVA THREADS, WINDOWS THREADS.

# Issues in Multi-Threading

**Thread cancellation, signal handling (synchronous/ asynchronous), handling thread-specific data, and scheduler activations**



# Thread Cancellation

**Definition**: The process of terminating a thread before it completes execution.

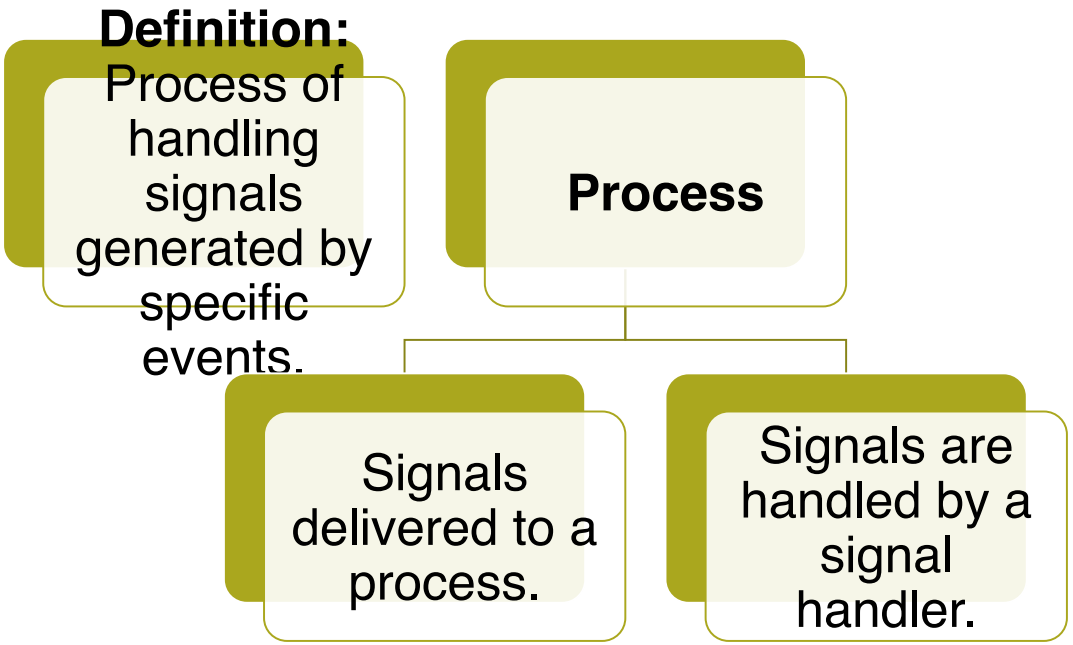
**Asynchronous Cancellation**

- **Definition**: Terminates the target thread immediately.
- **Use Case**: Required for immediate termination.

**Deferred Cancellation**

- **Definition**: Allows the target thread to periodically check if it should be canceled.
- **Use Case**: Ensures safe and orderly termination.

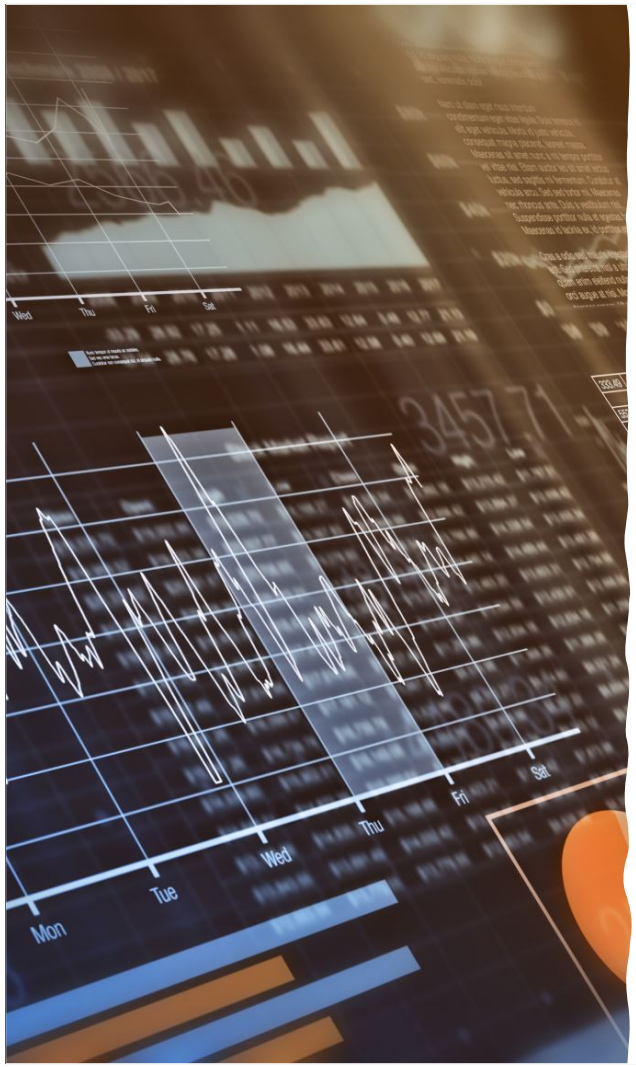
# Signal Handling



# Handling thread-specific data

- Definition: Managing data that is unique to each thread in a multi-threaded application.
- Process

1. **Creation**: Allocate data storage for each thread.
2. **Access**: Provide mechanisms for each thread to access its own data.
3. **Cleanup**: Ensure thread-specific data is cleaned up when the thread terminates.




# Scheduler Activations

- **Definition**: A mechanism that provides upcalls, enabling communication from the kernel to the thread library.
- **Purpose**: Allows the application to maintain the correct number of kernel threads.
- **Benefit**: Improves performance and resource management.

# basic thread creation and execution

- **Define Functions**: Two functions, print_numbers and print_letters, are defined. Each function prints a sequence of items with a 1-second delay between prints.
- **Create Threads**: Two threads, t1 and t2, are created. t1 runs the print_numbers function, and t2 runs the print_letters function.
- **Start Threads**: Both threads are started using the start() method, which triggers the execution of the respective functions in parallel.
- **Wait for Completion**: The join() method is used to wait for both threads to finish their execution before the main program continues.

```
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in ['A', 'B', 'C', 'D', 'E']:
        print(f"Letter: {letter}")
        time.sleep(1)

# Creating threads
t1 =
threading.Thread(target=print_number
s)
t2 =
threading.Thread(target=print_letters)

# Starting threads
t1.start()
t2.start()

# Waiting for threads to finish
t1.join()
t2.join()
```

# What will the print_numbers function do when executed in a thread?

A. Print numbers from 1 to 5, each followed by a 1-second delay

B. Print letters from A to E, each followed by a 1-second delay

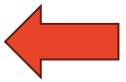
C. Print numbers from 1 to 5 without any delay

D. Print numbers from 1 to 5, but with a 1-second delay between threads

What is the purpose of the t1.join() and t2.join() calls in the code?

A. To start the threads t1 and t2

B. To wait for the threads t1 and t2 to complete before the main program exits



C. To ensure that t1 and t2 run concurrently

D. To terminate the threads t1 and t2 immediately

What will happen if time.sleep(1) is removed from both functions?

A. The output will be printed immediately without any delay

B. The threads will not start at all

C. The threads will wait for 1 second before printing the output

D. The program will hang indefinitely

# Activity

- Scan this QR code or click the link below to read the article titled 'A Comparison of Process and Thread Creation' by Martin Sysel.
- Then, be prepared to answer multiple-choice questions that the professor will ask in class.

What is the primary difference between processes and threads according to the article?

A. Processes deal with resource ownership, while threads are units of execution.

B. Processes are lighter than threads.

C. Threads can exist without processes.

D. Processes cannot create threads.

In Linux, which system call is primarily responsible for creating both processes and threads?

A. exec()

B. fork()

C. clone() 

D. pthread_create()

How does the fork() function differ from pthread_create() in terms of resource sharing?

A. fork() shares more resources than pthread_create().

B. pthread_create() shares more resources than fork().

C. Both share the same amount of resources.

D. Neither shares any resources.

Which of the following is a key advantage of threads over processes?

A. Threads are safer and more secure.

B. Threads have faster inter-thread communication.

C. Threads have their own address space.

D. Threads do not require synchronization.

In the context of this article, which statement is true about processes and threads in Linux?

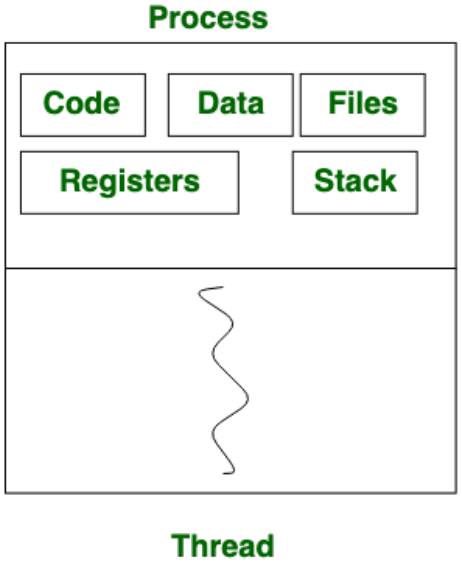
A. Processes are generally more lightweight than threads.

B. Threads and processes are both created using the exec() system call.

C. Processes and threads are both considered tasks and are similarly scheduled.

D. Threads cannot share virtual memory with their parent process.

# Process vs. Thread




**Process**

- **Memory**: Separate.
- **Overhead**: Higher.
- **Communication**: Uses IPC.
- **Creation**: Slower and more expensive.
- **Isolation**: Isolated from other processes.

**Thread**

- **Memory**: Shared.
- **Overhead**: Lower.
- **Communication**: Easy within process.
- **Creation**: Faster and cheaper.
- **Isolation**: Not isolated; affects other threads.

# Example of a process

Applications like word processors or web browsers.

# Example of a thread

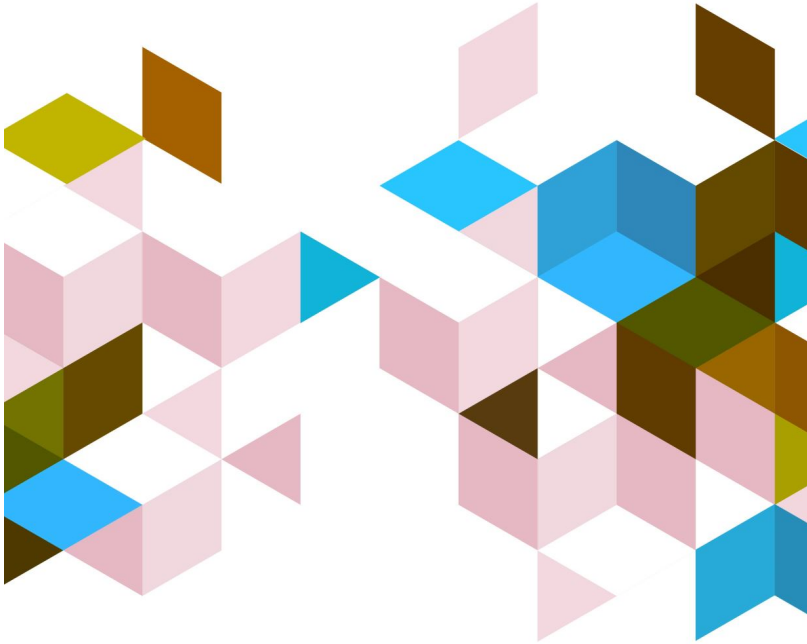
Background tasks or operations within an application.

# When to choose between using processes and threads

- Processes: Better isolation, security, and fault tolerance. Use when tasks are independent and require separate memory spaces. Examples: Web server handling each request in a separate process, Microservices architecture with separate processes for different services, and Database systems with each client connection in a separate process.

- Threads: Faster communication and resource sharing. Use when tasks are interdependent and can benefit from parallel execution. Example: Real-time data processing in a stock trading application, GUI applications with the main thread for UI and background threads for tasks, Parallel computation in scientific applications, and Web server with thread pools for handling multiple connections.

# Understanding Process Components and Management



# What is a process?

A process is a program in execution.

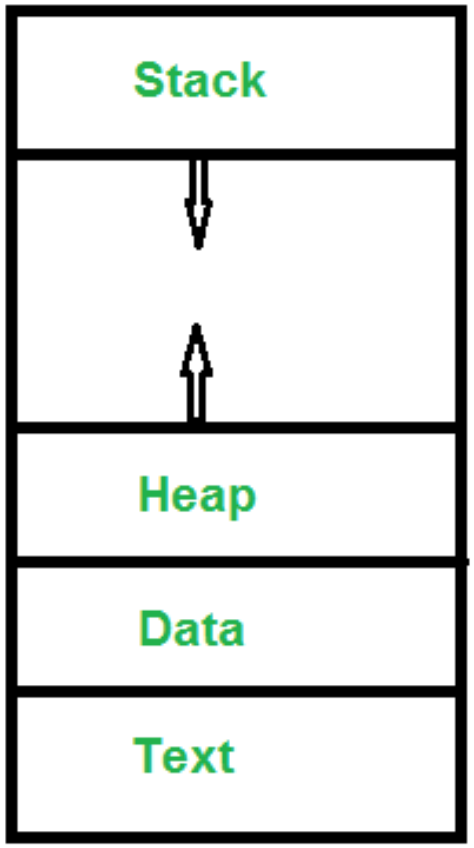
Ex. When we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we run the binary code, it becomes a process.

# What is Process Management?

- **Definition**: Techniques and strategies used to design, monitor, and control business processes for efficient and effective goal achievement.
- **Importance in Operating Systems**:
  - Critical for multi-user environments.
  - Tracks and schedules processes.
  - Dispatches processes sequentially.
  - Ensures users feel they have full control of the CPU.

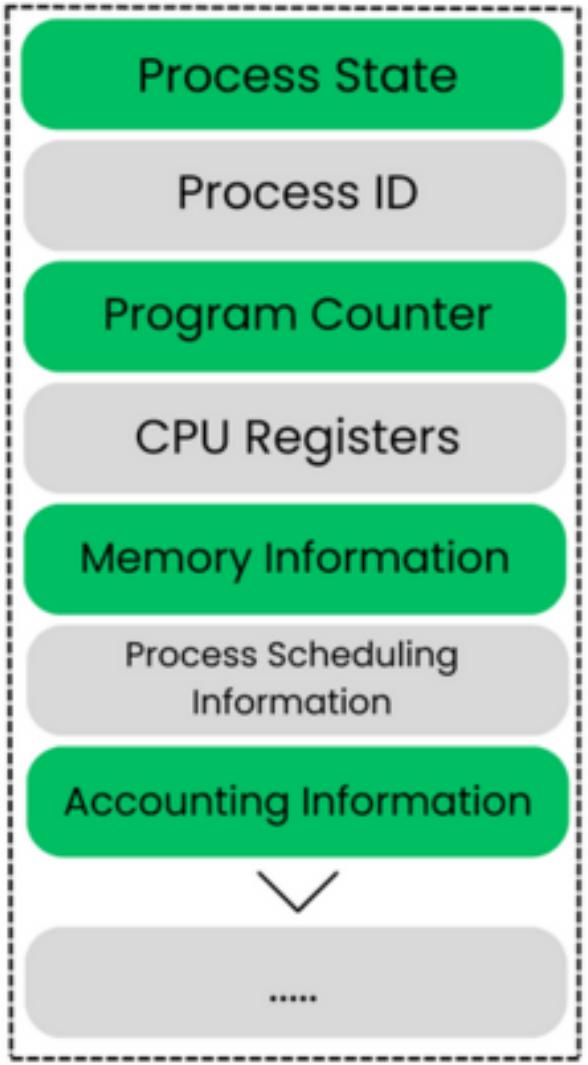
# Benefits

- **Operational Efficiency**: Streamlines task completion steps and resource usage.
- **Cost Reduction**: Identifies and eliminates inefficiencies.
- **Customer Satisfaction**: Improves service delivery and responsiveness.
- **Compliance**: Ensures adherence to regulatory standards.




# Process Components

- **Text Section:** Contains the code and current activity (Program Counter).
- **Stack:** Holds temporary data like function parameters and local variables.
- **Data Section:** Stores global variables.
- **Heap Section:** Manages dynamic memory allocation.

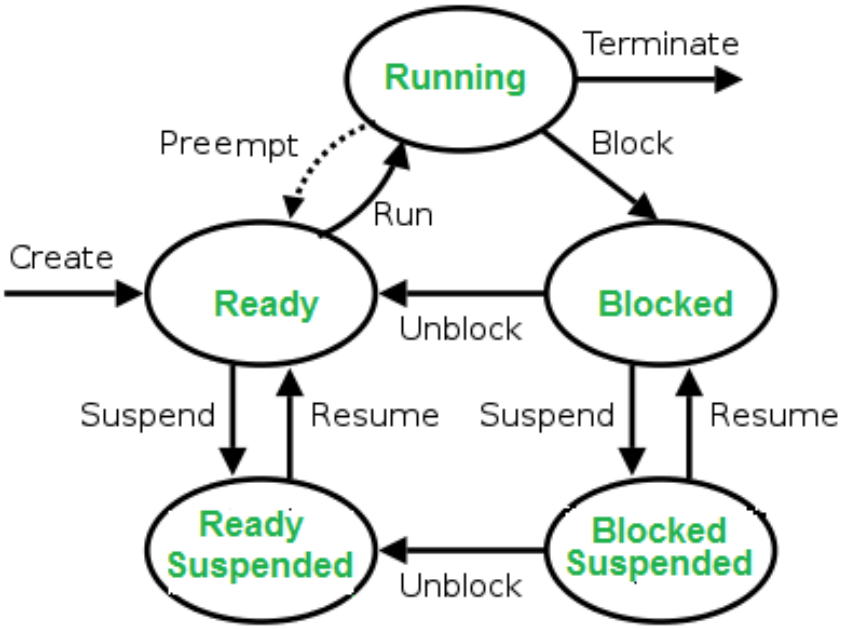



# Characteristics of a Process

---

- **Process Id:** A unique identifier assigned by the operating system.
- **Process State:** Can be ready, running, etc.
- **CPU Registers:** Like the Program Counter (CPU registers must be saved and restored when a process is swapped in and out of the CPU).
- **Accounts Information:** Amount of CPU used for process execution, time limits, execution ID, etc.
- **I/O Status Information:** For example, devices allocated to the process, open files, etc.
- **CPU Scheduling Information:** For example, Priority (Different processes may have different priorities, for example, a shorter process assigned high priority in the shortest job first scheduling).

**Note:** Diagram shows a typical structure of a **Process Control Block (PCB)** containing these attributes.




# States of Process

- **New:** Newly Created Process (or) being-created process.
- **Ready:** After creation, process moves to Ready state, i.e., ready for execution.
- **Run:** Currently running process in CPU (only one process at a time can be under execution in a single processor).
- **Wait (or Block):** When a process requests I/O access.
- **Complete (or Terminated):** The process completed its execution.
- **Suspended Ready:** When the ready queue becomes full, some processes are moved to a suspended ready state.
- **Suspended Block:** When the waiting queue becomes full.

# Real-Life Scenario: Cooking a Meal in a Kitchen

Imagine you're cooking a meal in your kitchen. You have several tasks to complete, like chopping vegetables, boiling water, and frying ingredients. Your kitchen has limited resources—only so many burners on the stove, a single cutting board, and one set of knives.

- **Processes:** Cooking a dish, boiling water, chopping vegetables.
- **Resources:** Stove, pots, knives, ingredients.
- **States:**
  - **Ready:** Ingredients are prepped, but not yet cooking.
  - **Running:** A dish is being cooked on the stove.
  - **Waiting:** Waiting for the water to boil.
  - **Terminated:** The dish is cooked and ready to be served.

# Activity

**Case Scenario - The Café Analogy**

Imagine you are the manager of a busy café. The café has a limited number of tables, baristas, and kitchen space to serve customers. Each customer who enters needs to be seated, place an order, and have their food or drinks prepared by the baristas.

# Questions

1. In the café scenario, who or what represents a process? Explain how this analogy helps you understand what a process is in an operating system.
2. In the café scenario, who or what represents a CPU in a computer system? How does this help you understand the "running" state of a process?
3. In the café scenario, who or what represents a system resource? How does this help you understand the "waiting" or "blocked" state of a process?
4. What happens when multiple customers arrive at the café at the same time? How does this scenario relate to processes in an operating system?
5. Consider a scenario where a customer must leave suddenly before finishing their coffee. How would you relate this to the concept of process?
6. How does the café handle peak hours when many customers arrive simultaneously? Relate this to how an operating system manages multiple processes that need CPU time and other resources.

In the café scenario, who or what represents a process? Explain how this analogy helps you understand what a process is in an operating system.

---

Each customer represents a process. Just like a process in an OS, a customer performs tasks (ordering, eating) and requires resources (table, barista). This helps us see a process as an active entity that moves through different stages.

In the café scenario, who or what represents a CPU in a computer system? How does this help you understand the "running" state of a process?

---

The barista is like the CPU, executing tasks (making coffee) for customers (processes). When a barista is working on an order, the process is in the "running" state, similar to the CPU executing a process.

In the café scenario, who or what represents a system resource? How does this help you understand the "waiting" or "blocked" state of a process?

---

Tables represent system resources like memory. If all tables are occupied, customers must wait, similar to how a process waits when needed resources are unavailable.

What happens when multiple customers arrive at the café at the same time? How does this scenario relate to processes in an operating system?

---

When multiple customers arrive, the baristas must decide who to serve first, similar to how an OS schedules processes. The order of service reflects process scheduling.

Consider a scenario where a customer must leave suddenly before finishing their coffee. How would you relate this to the concept of process?

---

If a customer leaves suddenly, it's like a process being terminated early, freeing up resources (table, barista time) for others. This is similar to closing or crashing a program.

How does the café handle peak hours when many customers arrive simultaneously? Relate this to how an operating system manages multiple processes that need CPU time and other resources.

---

During peak hours, the café manages a high load by prioritizing orders and optimizing service. This is like an OS managing multiple processes efficiently under heavy load.

# Types of Processes

- **I/O Bound**:
  - Spends more time doing I/O than computations
  - Many short CPU bursts
- **CPU Bound**:
  - Spends more time doing computations
  - Few very long CPU bursts

* CPU burst is a period when the CPU is fully engaged with a process's tasks

# Ex. I/O-Bound Processes

A text editor that frequently reads from and writes to files is I/O-bound. It spends more time waiting for file I/O operations than it does processing text.

# Ex. CPU-Bound Processes

A video encoding application or a scientific simulation is CPU-bound. It spends most of its time performing complex calculations, utilizing the CPU heavily, with minimal I/O operations.

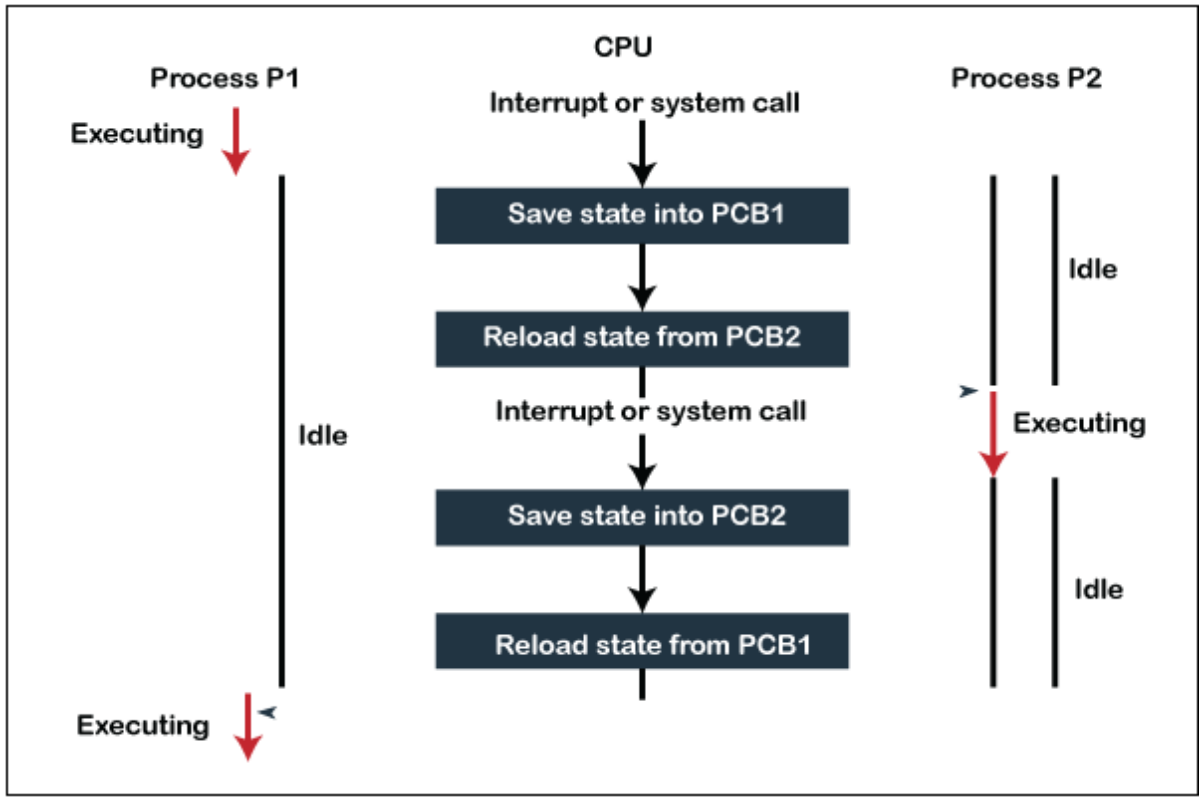
# Importance in Scheduling

- Understanding whether a process is I/O-bound or CPU-bound helps the OS's scheduler to allocate CPU time more effectively.

For example, Scheduling algorithms may prioritize I/O-bound processes to keep the system responsive. Since I/O-bound processes often spend time waiting for I/O and quickly free up the CPU, this allows CPU-bound processes to run during those idle periods, making the system more efficient.

# Context Switching

- **When CPU switches to another process:**
  - Save the state of the old process (to PCB)
  - Load the saved state (from PCB) for the new process
- **Context Switch Time**: Dependent on hardware

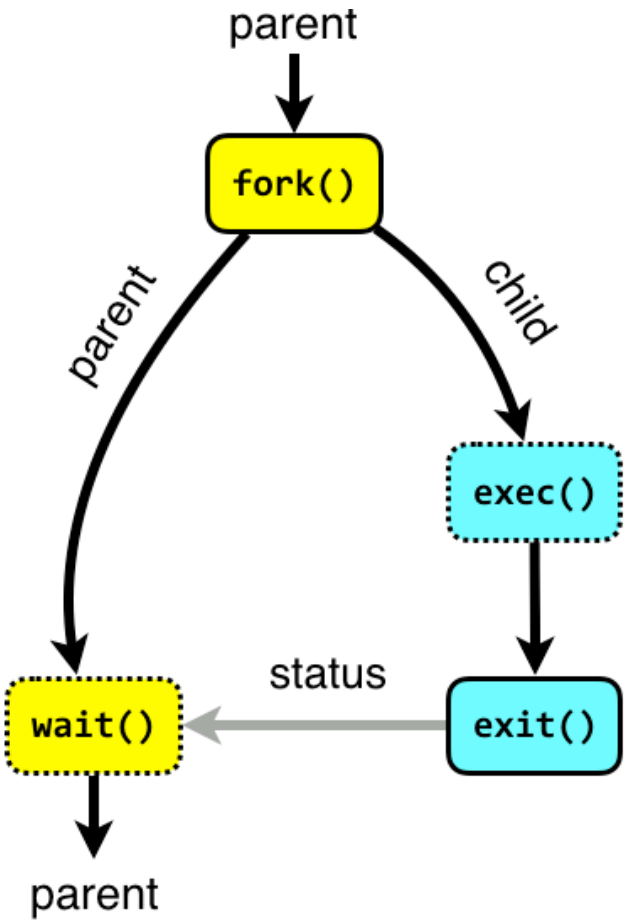


# Process Creation

- **Parent Processes Create Children Processes** (form a tree)
- **Process Management**: Using PID
- **Resource Sharing**: All, some, or none
- **Execution**:
  - Concurrently with children
  - Wait until children terminate

# System Calls

- **fork()**: Creates a new process
- **exec()**: Replaces the process's memory space with a new program



# Understanding Forking with Illustrative Example

- Imagine one participant draws a simple picture or scene on their paper. This drawing represents the **"parent process"** in an operating system.
- Now, another participant is asked to replicate this drawing exactly on their own sheet of paper. This second drawing represents the **"child process"** that is created when a process is forked.
- Both participants work independently. Changes made to one drawing do not affect the other. This illustrates how **each process has its own space and does not automatically share changes with the other.**

# Reflection

- **Process Independence:** The drawings show how processes run separately and do not affect each other.
- **Initial Copy, Independent Evolution:** The child process starts as a copy with the same state but can develop differently, similar to the drawings.
- **Separation of Tasks:** Changes in one process do not impact the other, demonstrating effective multitasking.

# Modification Impact

- Changes made to shared resources by either the parent or child process will affect both.

**Example:** If the parent process modifies a file descriptor, the child process will see the change too.

# Undesirable Changes

- Unintended modifications can lead to unexpected behavior in either process.

**Example:** If one process updates a shared variable without coordination, the other process might operate on outdated or incorrect data.

# To address Modification Impact and Undesirable Changes, you should:

- Implement synchronization mechanisms.
- Manage file descriptors carefully to avoid unintended modifications.
- Use atomic operations for shared variables.
- Employ IPC mechanisms and locking strategies.
- Design consistent access patterns for shared resources.

# Alternatives to fork

- **posix_spawn:** Simplifies process creation with reduced overhead and fewer pitfalls.
- **vfork:** Creates a new process sharing the parent's address space until exec is called.
- **clone:** Provides fine-grained control over shared resources between parent and child.
- **CreateProcess:** Windows API for creating and initializing new processes.
- **Microkernel APIs:** Offers modular process creation and mangement without fork.

# Despite its limitations, fork is still commonly used because:

- **Legacy Systems:** Many existing applications depend on fork, and changing them would be complex.
- **Simplicity:** fork offers a straightforward way to create processes, especially in traditional Unix applications.
- **Copy-on-Write:** Modern fork implementations use copy-on-write to efficiently manage memory.
- **Historical Precedent:** fork is a long-standing standard in Unix-like systems, maintaining consistency with past practices.
- **Application Design:** Some applications are built around fork, making it challenging to switch.
- **Limited Alternatives:** Alternatives like posix_spawn and vfork don't fully replace all of fork's use cases or functionality.

# Activity

Step 1: Read the Code

Read through the provided code snippet to understand its structure and what it does.

Step 2: Identify Fork Calls

Identify the locations in the code where the fork() function is called. Each fork() call creates a new child process.

Step 3: Count Forks

Count the number of fork() calls in the code. For each fork() call, determine how many additional processes it creates.

Step 4: Calculate Total Forks

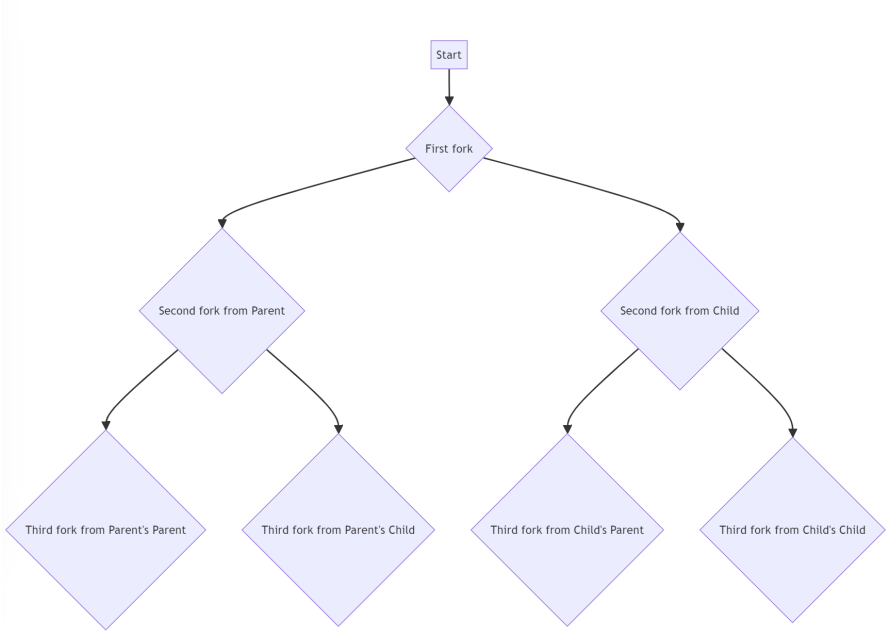
Add up the number of additional processes each fork() call creates to find the total number of forks created in the code.

Step 5: Final Answer

Share your findings. How many forks are created in the provided code snippet?

# Code Snippet

```
#include <stdio.h>
#include <unistd.h>

int main() {
    fork();
    fork();
    fork();
    return 0;
}
```



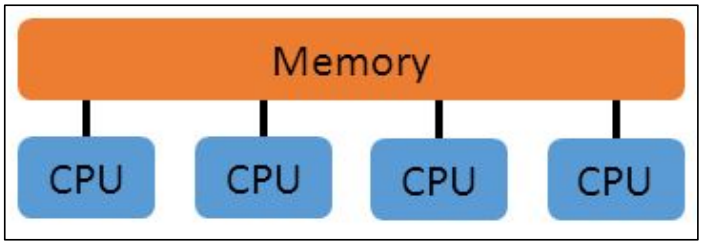

# Answer

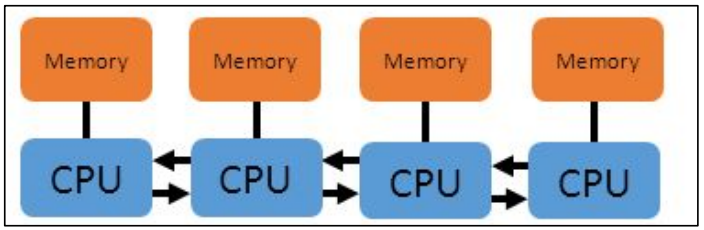
Let's count the number of forks created in the code:

- The first fork() creates one additional process.
- The second fork() is encountered after the first fork, so it creates another two processes (one for each existing process).
- The third fork() is encountered after the second fork, so it creates another four processes (one for each existing process).

So, in total, there are 1 (first fork) + 2 (second fork) + 4 (third fork) = 7 forks created.

# Interprocess Communication (IPC)




- Shared memory: A method where multiple processes share a common memory space to communicate and exchange data.
- Message passing: A method where processes communicate by sending and receiving messages, usually through a communication channel or a messaging queue.

# Message Passing

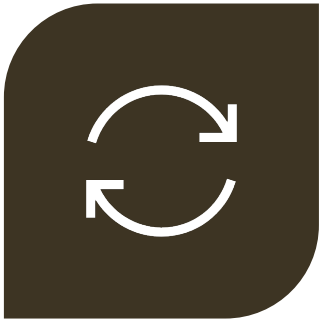
**Blocking (Synchronous):**

- Blocking send: Sender waits until the message is received.
- Blocking receive: Receiver waits until a message is available.

**Non-blocking (Asynchronous):**

- Non-blocking send: Sender sends and proceeds without waiting.
- Non-blocking receive: Receiver checks for a message; continues if none is found.

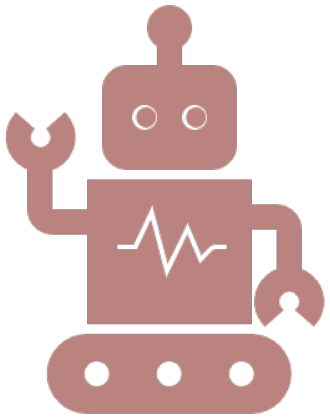
# Current Challenges in OS Processes



**CONCURRENCY ISSUES:** MANAGING MULTIPLE PROCESSES EFFICIENTLY, ESPECIALLY IN MULTI-CORE SYSTEMS.

**SECURITY IN PROCESSES:** ENSURING THAT PROCESSES DO NOT INTERFERE WITH EACH OTHER AND PROTECTING SYSTEM INTEGRITY.

**RESOURCE ALLOCATION:** FAIR AND EFFICIENT DISTRIBUTION OF CPU, MEMORY, AND I/O RESOURCES AMONG COMPETING PROCESSES.



With AI and Large Language Models rising, data sovereignty, user control, identity, and the proprietary vs. open-source debate are back. Should we rethink yesterday's OS Processes for tomorrow?

# The Need for Advanced OS Processes

- **Dynamic Resource Management:** Adaptive algorithms that adjust resource allocation based on real-time needs.
- **AI-Assisted Process Scheduling:** Utilizing AI to predict and optimize task scheduling for better performance.
- **Enhanced Security Protocols:** Advanced techniques to isolate and protect processes from malicious interference.

# Core Processes of a Future OS

- **Modular Process Management:** A system where processes can be independently updated and optimized.
- **Context-Aware Processing:** Processes that adapt based on the current environment, device, and user behavior.
- **Scalable Process Handling:** Efficient process management across diverse devices, from IoT to high-performance computing.

# Potential Features in Process Management

- **AI-Driven Resource Allocation:** AI optimizes how resources are distributed to processes based on usage patterns.
- **Seamless Cross-Platform Processes:** Processes that maintain state and functionality across different devices.
- **User-Controlled Process Prioritization:** Allowing users to prioritize which processes should receive more system resources.

# How can an operating system balance process efficiency with user experience?

An OS can balance efficiency and user experience by implementing adaptive process management. For instance, the OS could **prioritize foreground applications** (those actively used by the user) while optimizing background tasks to minimize resource use. AI-driven predictions can enhance responsiveness, ensuring **critical tasks receive more CPU and memory without causing delays**. Additionally, providing users with customizable settings allows them to decide how much system performance is devoted to specific tasks, enhancing overall satisfaction while maintaining efficiency.

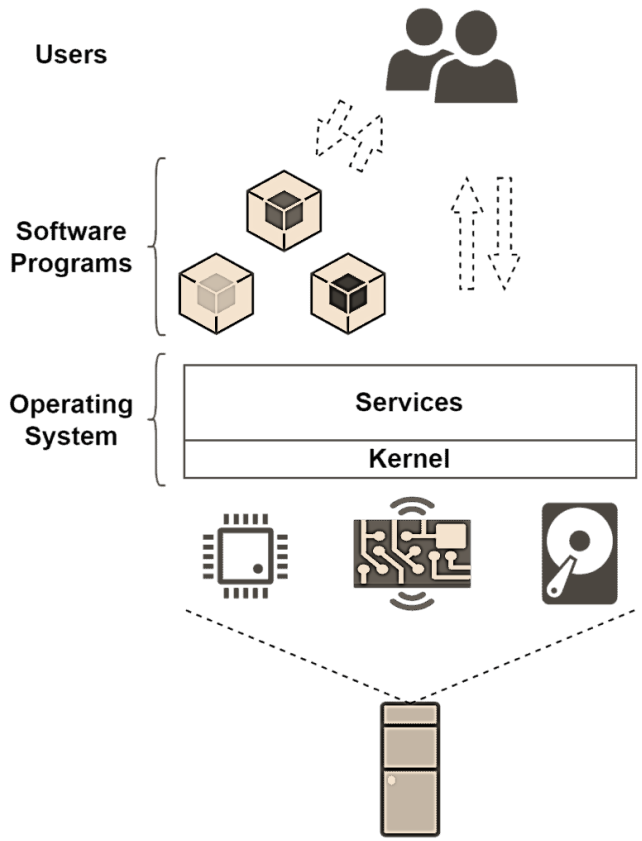
# Understanding OS Structures

# User Interface (UI)

- Can be Command-Line (CLI), Graphics User Interface (GUI), or Batch
- Allows user interaction with system services via system calls (typically written in C/C++)

# System Services for Users

- Program execution
- I/O operations
- File-system manipulation
- Communications
- Error detection



# Services for Efficient OS Operation

- Resource allocation
- Accounting
- Protection and security

# System Calls and APIs

- Accessed via APIs such as Win32, POSIX, Java
- Each system call has an associated number
- System call interface maintains a table indexed by these numbers

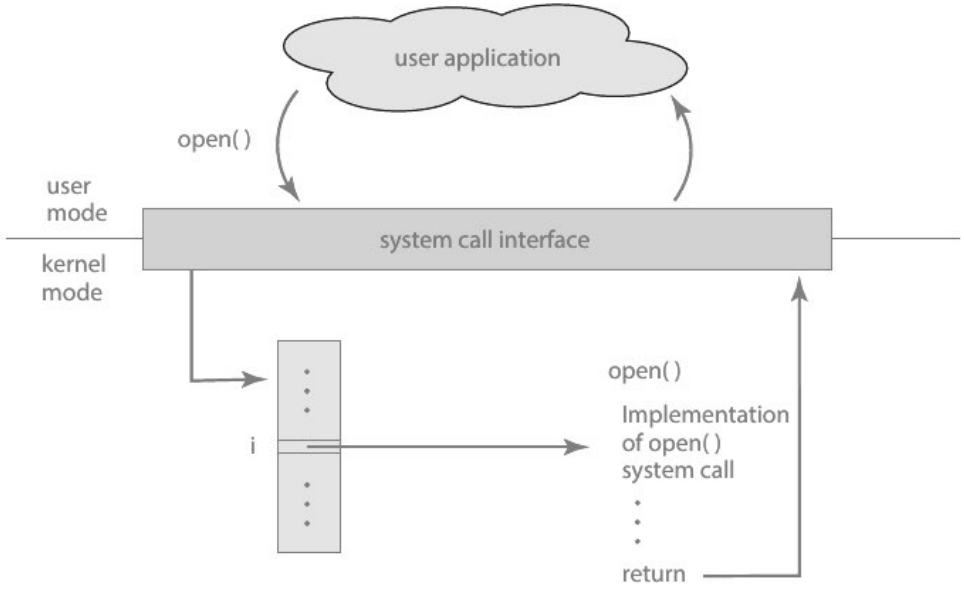


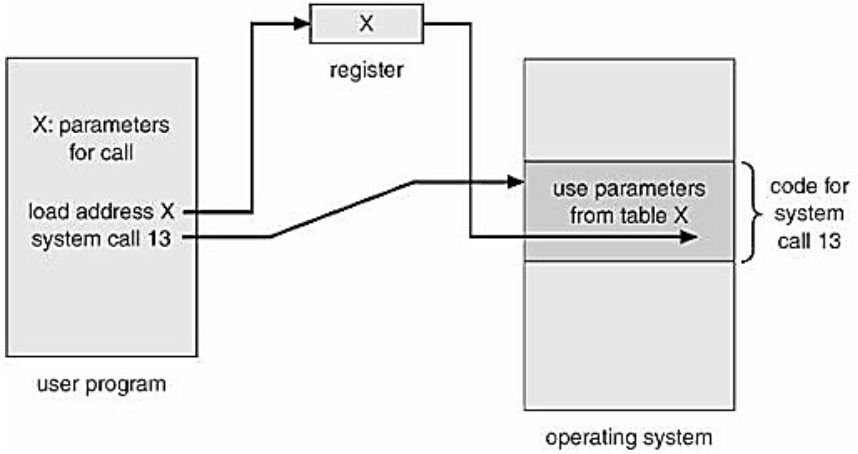

**Figure 2.6** The handling of a user application invoking the `open()` system call.

# Passing Parameters in System Calls

•Methods:

- Passing in registers
- Address of parameter stored in a block
- Pushed onto the stack by the program, popped off by the OS

•Block and stack methods allow for flexibility in the number and length of parameters

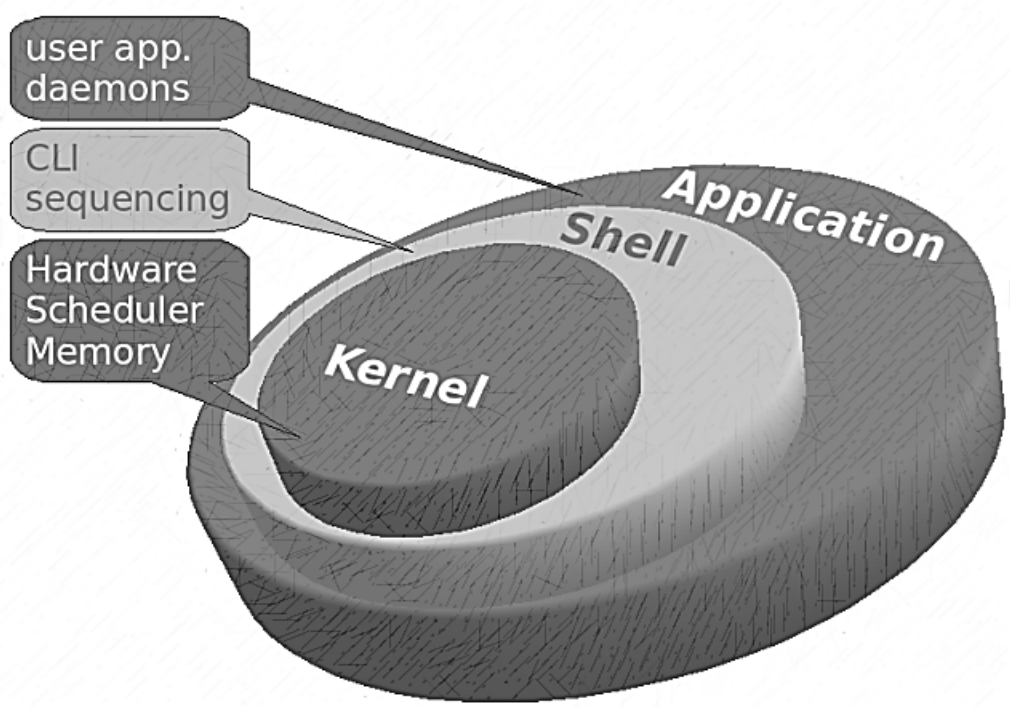


# Types of System Calls

- **Process control:** end, abort, load, execute, create/terminate process, wait, allocate/free memory
- **File management:** create/delete file, open/close file, read, write, get/set attributes
- **Device management:** request/release device, read, write, logically attach/ detach devices
- **Information maintenance:** get/set time, get/set system data, get/set process/file/device attributes
- **Communications:** create/delete communication connection, send/ receive, transfer status information

# Understanding Operating System Components

- Operating systems facilitate user interaction with computer hardware and resource management.
- Two fundamental components: Shell and Kernel.



# Shell

Definition: User interface for interacting with the OS.

Functionality: Accepts user commands, translates them for the kernel.

Features: Command history, tab completion, scripting.

# Shell Commands in C++ (Windows vs. Unix)

| Function | Windows Command | Unix Command |
|---|---|---|
| List Directory Contents | system("dir") | system("ls -l") |
| Create Directory | system("mkdir new_dir") | system("mkdir new_dir") |
| Remove Directory | system("rmdir new_dir") | system("rmdir new_dir") |
| Copy File | system("copy source.txt destination.txt") | system("cp source.txt destination.txt") |
| Move File | system("move source.txt destination.txt") | system("mv source.txt destination.txt") |
| Delete File | system("del file.txt") | system("rm file.txt") |
| Print Working Directory | system("cd") | system("pwd") |
| Display File Content | system("type file.txt") | system("cat file.txt") |

# Kernel

Definition: Core component managing system resources.

Responsibilities: Memory management, process scheduling, device management.

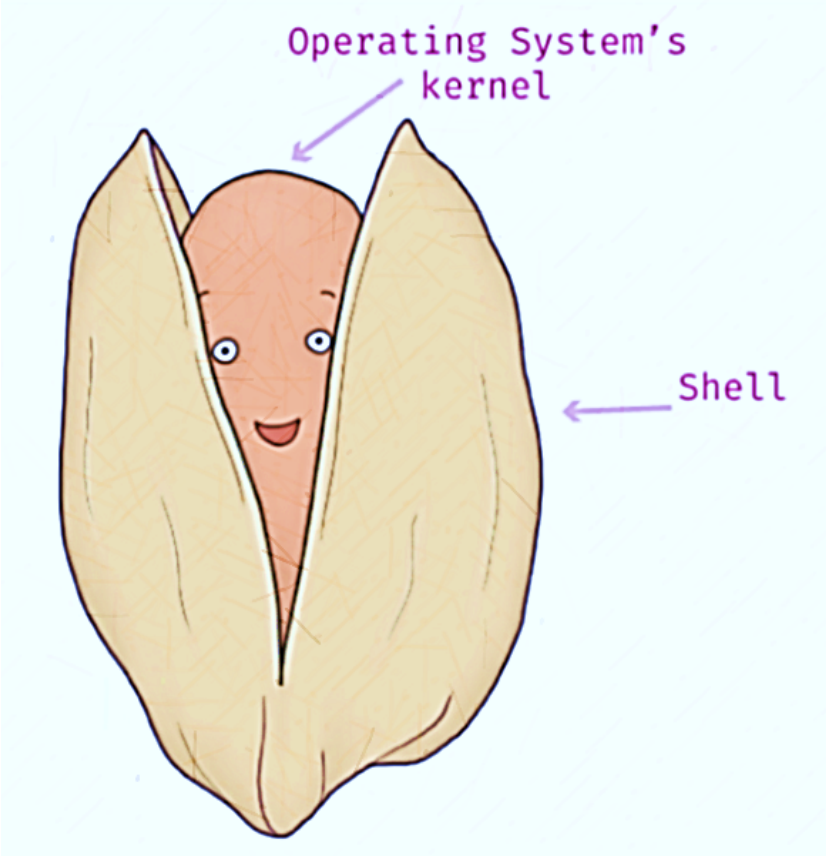
Interaction: Directly communicates with hardware.

# System Calls

| System Call | Description |
| --- | --- |
| fork() | Creates a new process by duplicating the calling process. |
| exec() | Replaces the current process image with a new process image. |
| wait() | Makes the calling process wait until one of its child processes terminates. |
| exit() | Terminates the calling process and returns a status to the parent process. |
| open() | Opens a file or device and returns a file descriptor. |
| read() | Reads data from a file descriptor into a buffer. |
| write() | Writes data from a buffer to a file descriptor. |

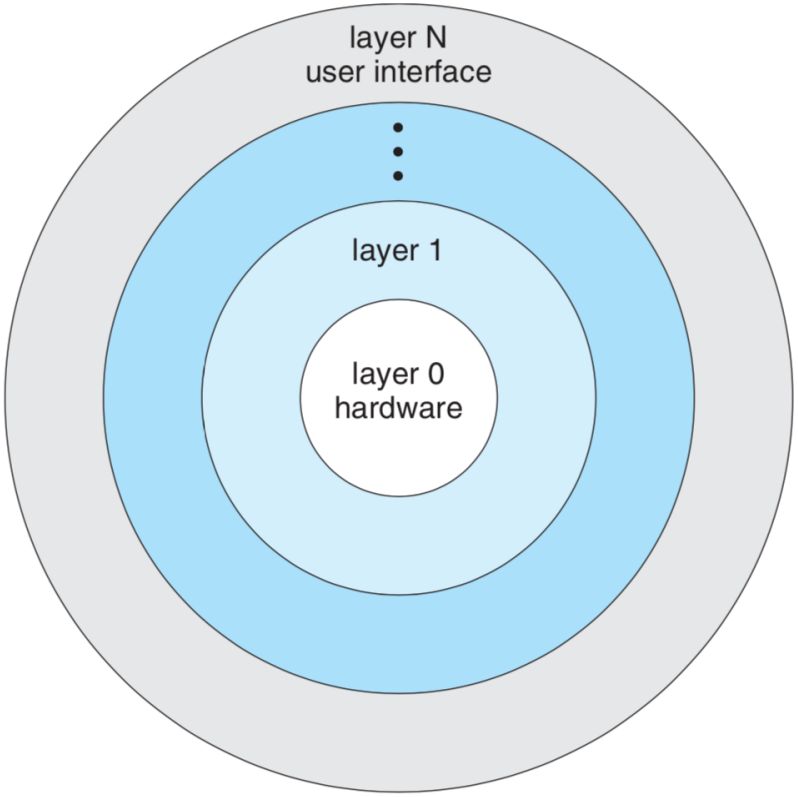
# Difference between Shell and Kernel

- **Shell**: User interface, interprets user commands.
- **Kernel**: Core system component, manages hardware and resources.



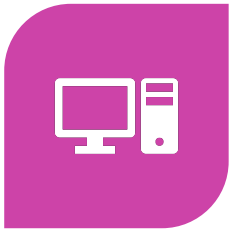
# OS Layered Approach

- Divided into layers (levels), with hardware at the bottom (layer 0) and the user interface at the top (layer N)
- Each layer uses functions and services of lower layers



# Virtual Machines

USES A LAYERED APPROACH



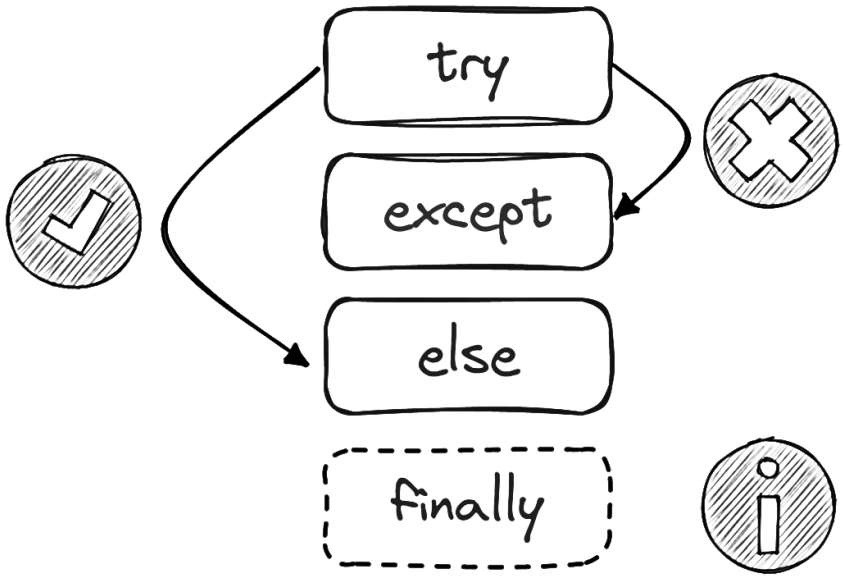
TREATS HARDWARE AND OS KERNEL AS HARDWARE

HOST CREATES THE ILLUSION OF A DEDICATED PROCESSOR AND MEMORY FOR EACH PROCESS

EACH GUEST HAS A 'VIRTUAL' COPY OF THE COMPUTER

# Error Handling

- Application failures generate core dump files capturing the memory of the process
- OS failures generate crash dump files containing kernel memory



# Activity

- Write a program that executes five different shell commands. For example:
  - **dir** (List contents of current directory)
  - **echo "Hello, World!"** (Print "Hello, World!")
  - **mkdir test_directory** (Create a new directory named "test_directory")
  - **type example.txt** (Display contents of a text file named "example.txt")
  - **cd** (print the current working directory)
- Experiment with different commands and observe the output.
- Comment your code and provide explanations for each command you execute.
- Share your experience and any challenges you faced.
- Discuss the importance of using shell commands in programming and real-life applications.
- Highlight any security concerns or best practices when executing shell commands from a program.

# Sample Answer

**1.Experience & Challenges:**

1. Experience: Seamless interaction with the OS.
2. Challenges: Ensuring cross-platform compatibility.

**2.Importance:**

1. Vital for file manipulation, process management, and network operations.
2. Streamline workflows and automate tasks in real-life applications.

**3.Security & Best Practices:**

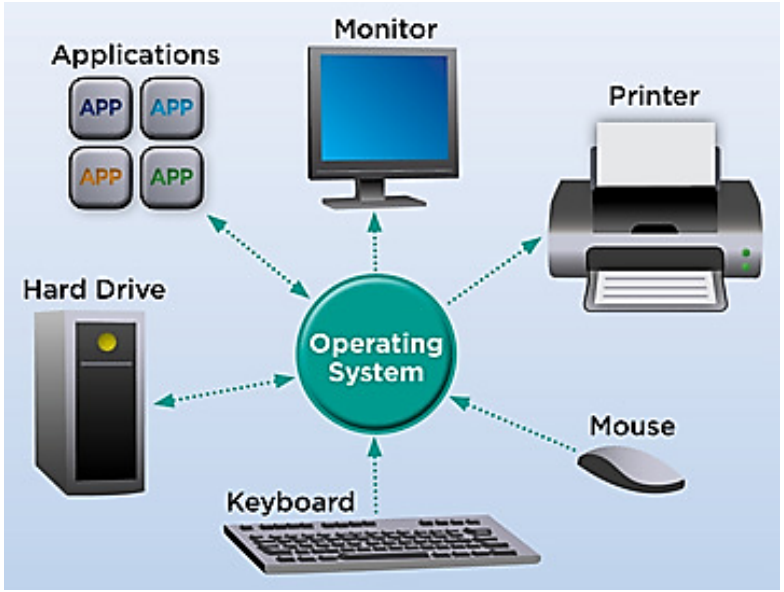
1. Guard against command injection attacks.
2. Validate user inputs and restrict access to sensitive resources.
3. Thoroughly test commands before deployment.

```
#include <cstdlib>
#include <iostream>

int main() {
    // 1. List contents of current directory
    std::cout << "Listing contents of current directory:\n";
    system("dir");

    // 2. Print "Hello, World!"
    std::cout << "\nPrinting 'Hello, World!':\n";
    system("echo Hello, World!");

    // 3. Create a new directory named "test_directory"
    std::cout << "\nCreating directory 'test_directory':\n";
    system("mkdir test_directory");

    // 4. Display contents of a text file named "example.txt"
    std::cout << "\nDisplaying contents of 'example.txt':\n";
    system("type example.txt");

    // 5. Change directory to "test_directory"
    std::cout << "\nChanging directory to 'test_directory':\n";
    system("cd test_directory");

    // Optional: Print current working directory
    std::cout << "\nCurrent working directory:\n";
    system("cd");

    return 0;
}
```

# Introduction to Operating Systems

# What is an Operating System?

- **Definition**: An OS is a program that acts as an intermediary between a user of a computer and the computer hardware.
- **Goals**:
  - Execute user programs
  - Make the computer system easy to use
  - Utilize hardware efficiently



# Evolution and History of Operating Systems

- **Early Days:** Computers had no operating systems; each program required different code, making data processing complex and time-consuming.
- **1956:** General Motors developed the first OS for a single IBM computer.
- **1960s:** IBM began installing OS in their devices.
- **UNIX:** First version launched in the 1960s, written in C.
- **Microsoft:** Developed OS at IBM's request.
- **Today:** All major computer devices have OS, performing similar functions with unique features.

# Common Operating Systems

# Fundamental Goals of an Operating System

1

**Efficient Use:** Optimize computer resource utilization

2

**User Convenience:** Provide easy and convenient system usage

3

**Non-Interference:** Prevent user activity interference

# Advantages and Disadvantages of Operating Systems

- **Advantages:**

✓**Memory Management:** Manages data in the device efficiently.

✓**Hardware Utilization:** Optimizes use of computer hardware.

✓**Security:** Maintains device security.

✓**Application Efficiency:** Helps run different applications smoothly.

- **Disadvantages:**

✓**Usability:** This can be difficult for some users.

✓**Cost and Maintenance:** Some are expensive and need heavy maintenance.

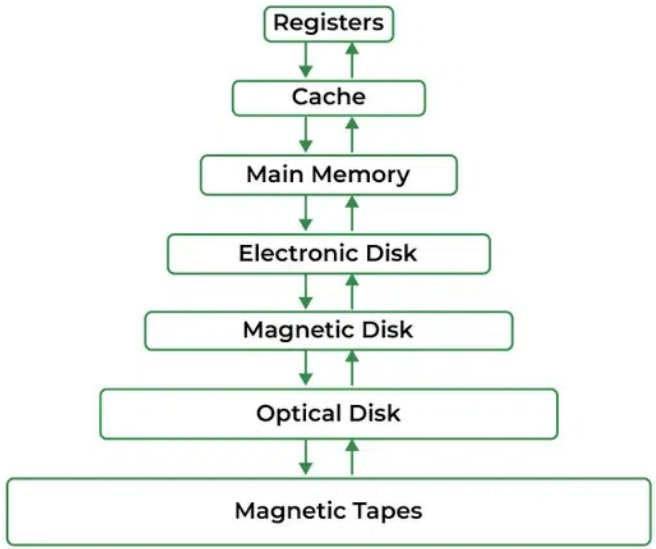
✓**Security Risks:** Vulnerable to hacking threats.

# Computer System Structure

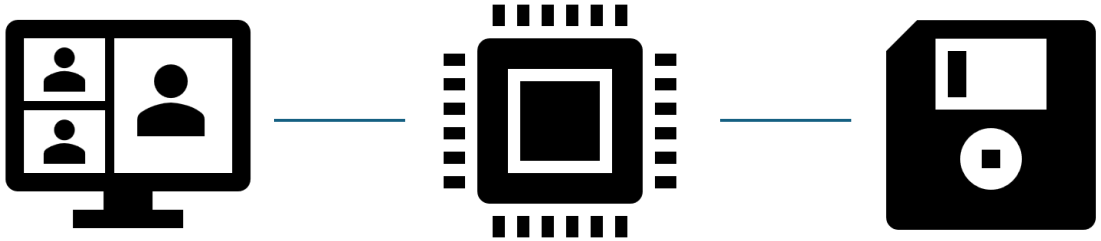
- **Hardware**: Physical components
- **OS**: Manages hardware, provides services for applications
- **Applications**: Software performing specific tasks
- **Users**: People interacting with the computer

# Functions of an OS

- **Resource Allocator**: Decides between conflicting requests for efficient and fair resource use.
- **Control Program**: Controls execution of programs to prevent errors and improper use of the computer.



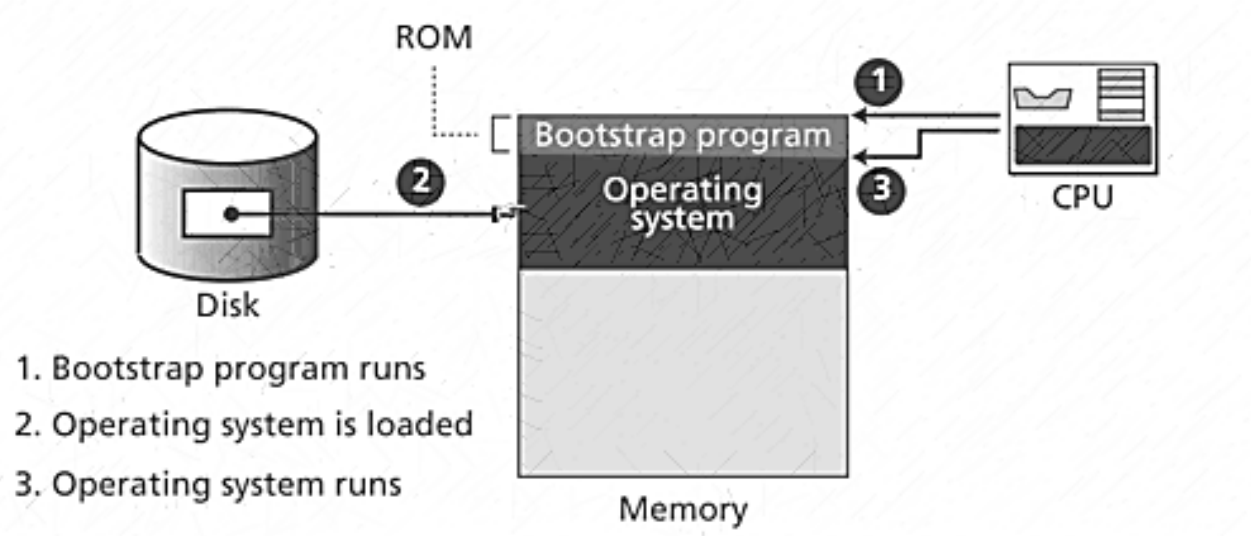
# The Kernel



- **Definition**: The one program running at all times on the computer.
- **Role**: Manages system resources and communication between hardware and software.

# Bootstrap Program

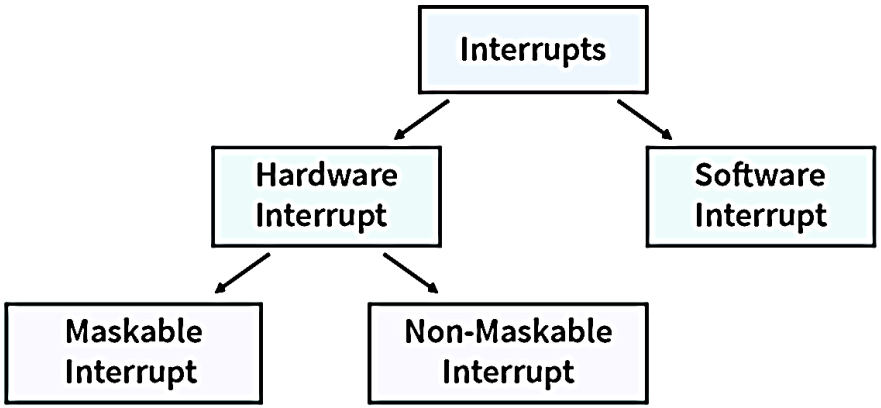
- **Definition**: Loaded at power-up or reboot.
- **Storage**: Stored in ROM or EPROM (firmware).
- **Function**: Initializes system aspects, loads OS kernel, and starts execution.



# Concurrency and Interrupts

- **Concurrency**: I/O and CPU can execute concurrently. Example: A user can type on a keyboard while a file is being downloaded.
- **Interrupts**:
  - Device controllers inform CPU of operation completion via an interrupt.
  - Control is transferred to the interrupt service routine through the interrupt vector.
  - Example: A printer controller signals the CPU that printing is complete.

# Types of Interrupts




- **Interrupts**: Incoming interrupts are disabled while another is processed.
- **Trap**: A software-generated interrupt caused by an error or user request.
- **OS Role**: Determines interrupt type via polling or vectored interrupt system.

# System Calls and Device-Status Table

- **System Call**: Request to the OS for I/O completion.
- **Device-Status Table**: Entry for each I/O device indicating its type, address, and state.

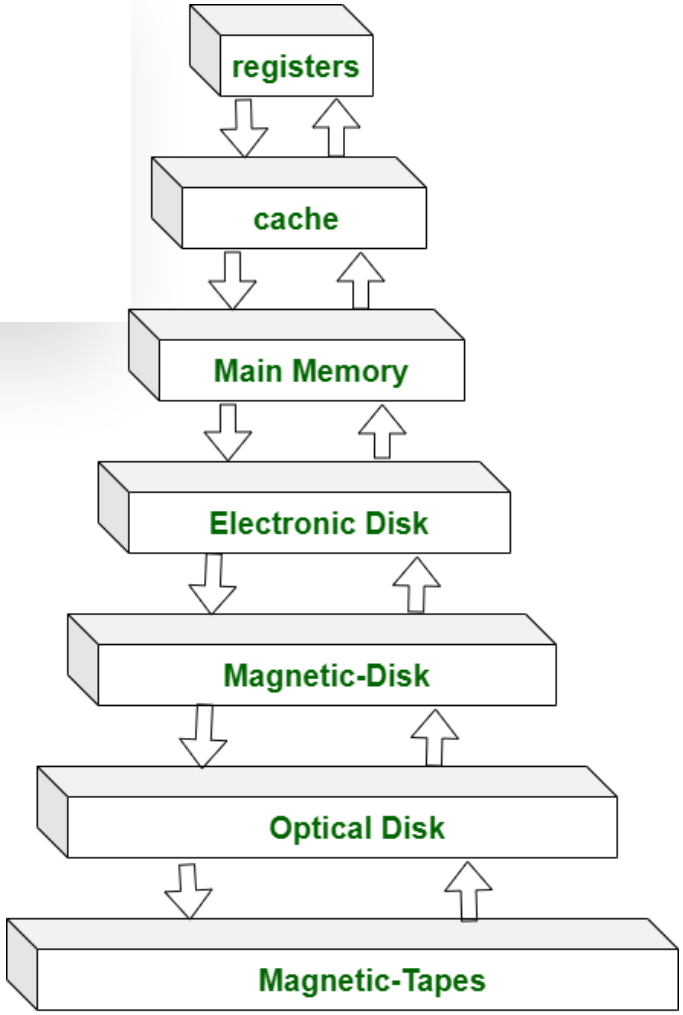
*EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

The following illustrates various equivalent system calls for Windows an UNIX operating systems.

| | Windows | Unix |
|---|---|---|
| Process control | CreateProcess() | fork() |
| | ExitProcess() | exit() |
| | WaitForSingleObject() | wait() |
| File management | CreateFile() | open() |
| | ReadFile() | read() |
| | WriteFile() | write() |
| | CloseHandle() | close() |
| Device management | SetConsoleMode() | ioctl() |
| | ReadConsole() | read() |
| | WriteConsole() | write() |
| Information maintenance | GetCurrentProcessID() | getpid() |
| | SetTimer() | alarm() |
| | Sleep() | sleep() |
| Communications | CreatePipe() | pipe() |
| | CreateFileMapping() | shm_open |
| | MapViewOfFile() | mmap() |
| Protection | SetFileSecurity() | chmod() |
| | InitlializeSecurityDescriptor() | umask() |
| | SetSecurityDescriptorGroup() | chown() |

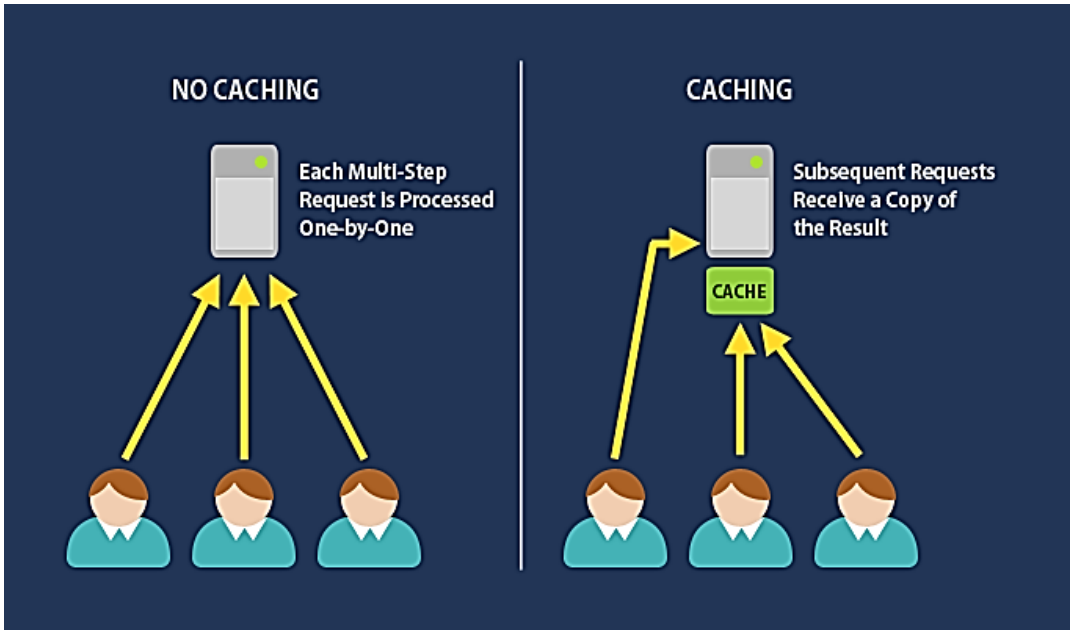
# Storage Structure

- **Main Memory**: Random access, volatile. Example: RAM, used for currently running programs.
- **Secondary Storage**: Large, non-volatile extension of main memory. Example: Hard drives, SSDs, for long-term data storage.
- **Disk**: Divided into tracks and sectors, managed by disk controller. Example: A hard disk is divided into tracks and sectors.



# Caching

Copying information into faster storage systems to speed up access.

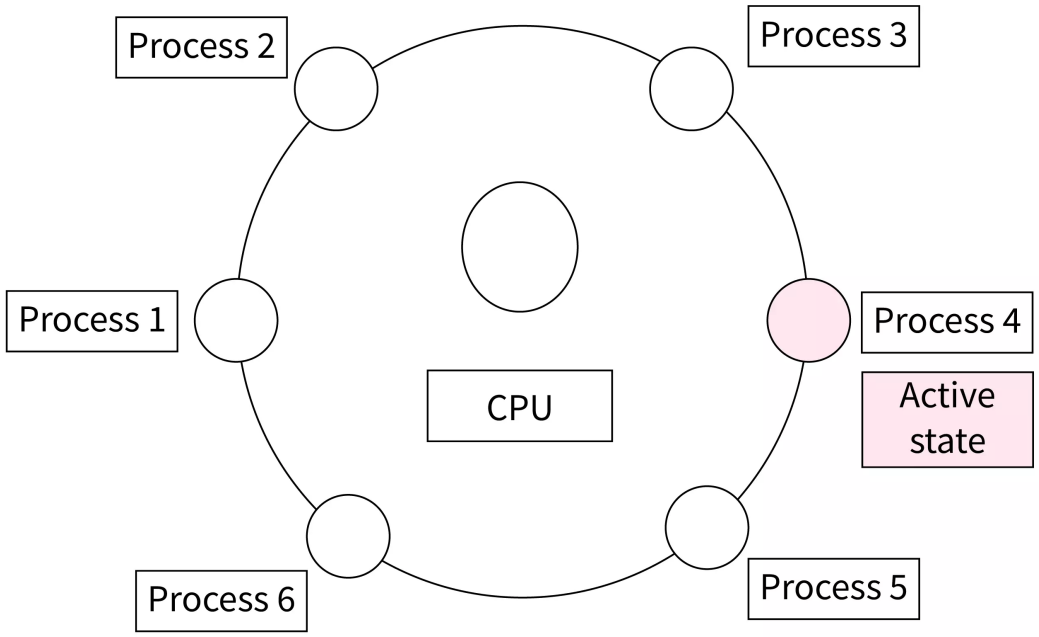


# Multiprocessor Systems

- **Benefits**: Increased throughput, economy of scale, increased reliability.
- **Types**:
  - Asymmetric
  - Symmetric
  - Clustered systems: Linked multiprocessor systems.

# Multiprogramming and Timesharing

- **Multiprogramming**: Provides efficiency via job scheduling.
- **Timesharing**: CPU switches jobs frequently for interactive computing.

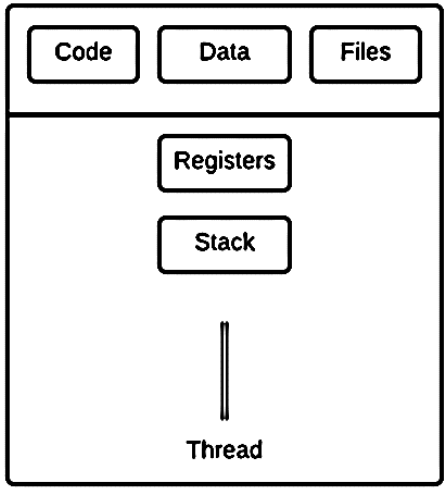


# Dual-Mode Operation

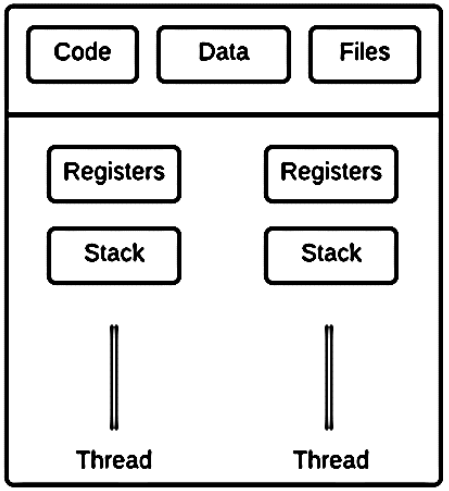
- **Definition**: Allows OS to protect itself and other system components.
- **Modes**:
  - User Mode
  - Kernel Mode
- **Privileged Instructions**: Executable only in kernel mode.

# Processes and Threads

- **Single-threaded Processes**: One program counter.
- **Multi-threaded Processes**: One program counter per thread.



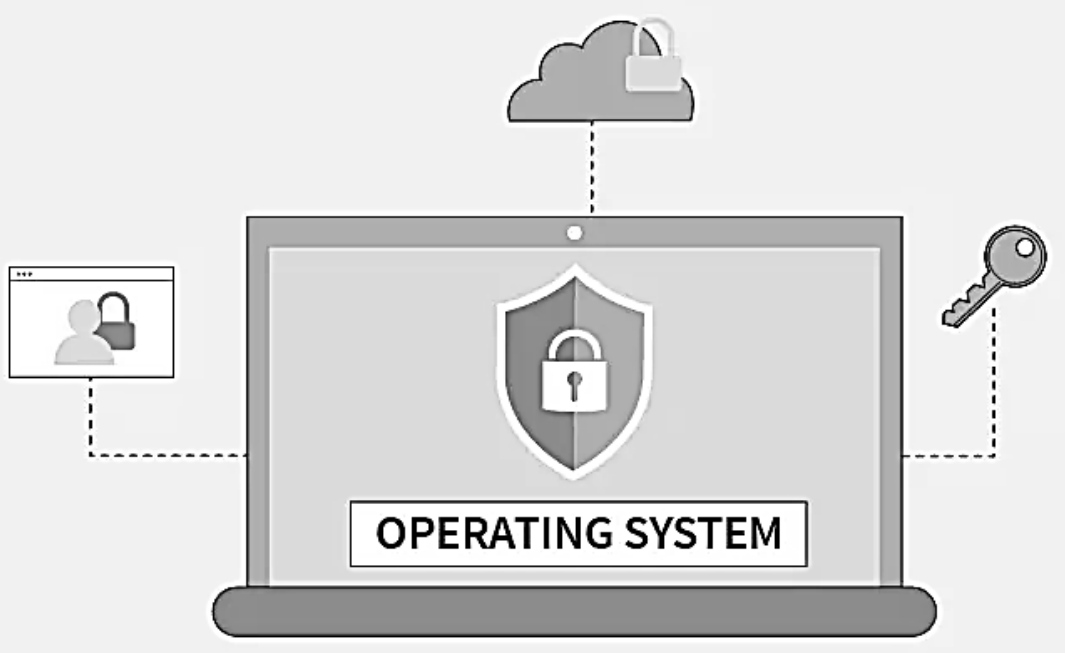

Single-threaded Process




Multi-threaded Process

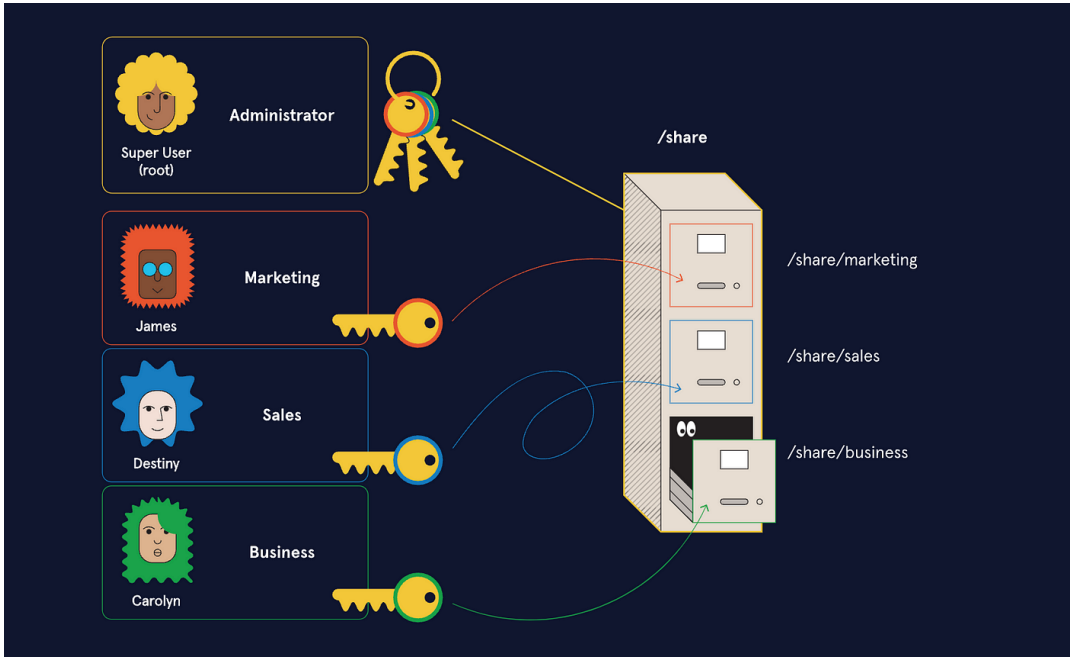
# Protection and Security

- **Protection**: Mechanism for controlling access to resources.
- **Security**: Defense against attacks.



# User and Group IDs

- **User IDs (UID)**: One per user.
- **Group IDs**: Determine privileges for users and groups.



# Activity

- **Choose an operating system such as Windows, macOS, or Linux.**
- **Answer these questions about the operating system you chose:**

1. What are the key features?
2. How does the user interface look and feel?
3. What security features are included?
4. What are the common use cases?

# Sample Answer

- macOS is known for its sleek and intuitive user interface, featuring the Dock and Mission Control for easy navigation. Key features include seamless integration with other Apple devices, built-in applications like Safari and Photos, and robust performance. Security features include FileVault for disk encryption, Gatekeeper for app security, and regular software updates. Common use cases for macOS include graphic design, video editing, and general productivity due to its stable performance and user-friendly interface.