#college-studies #computer IT

By: Yashraj Maher

Introduction of Operating System

1. An Operating System acts as on interface between users and computer hardware components.
2. The purpose of an Operating System is to provide an environment in which users can execute programs conveniently and efficiently.
3. An Operating System is software that manages computer hardware. The hardware must provide appropriate mechanisms to ensure correct operation of computer system and to prevent users programs from interfering with he proper operation of the system.
4. An Operating System core typically includes programs to manage these resources, such as a traffic controllers, a scheduler, memory management, I/O programs, and system utilities.

What is Program ?

- Program is set of Instructions.

What is Software ?

- Software is a set of Programs designed for specific tasks. There are basically two types of software.

System Software

- Software designed for controlling and managing the operation of computer systems is known as system software.
- Examples: Operating systems, device drivers etc.

Application Software

- Software designed for end-users to be used for specific purpose.
- Examples: MS-Office, Chrome, VS code, Canva, Whatsapp etc

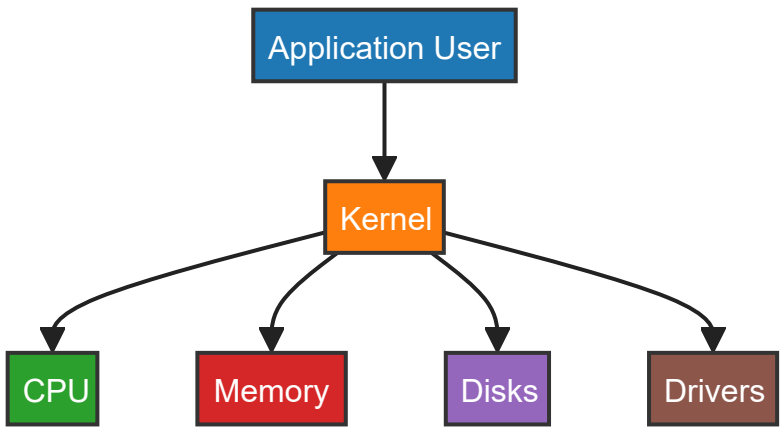
History of Operating System

Generation	Duration	Technologies	Types of O.S
First	1940s-50s	Vacuum Tubes	-
Second	1950s-60s	transistors	Batch Systems
Third	1960s-80s	Intergrated Circuits	Multi-Programming
Forth	1980s-	-	PC

- An operating system is an interface between the user and computer system. These are numbers of operating systems, e.g. windows.

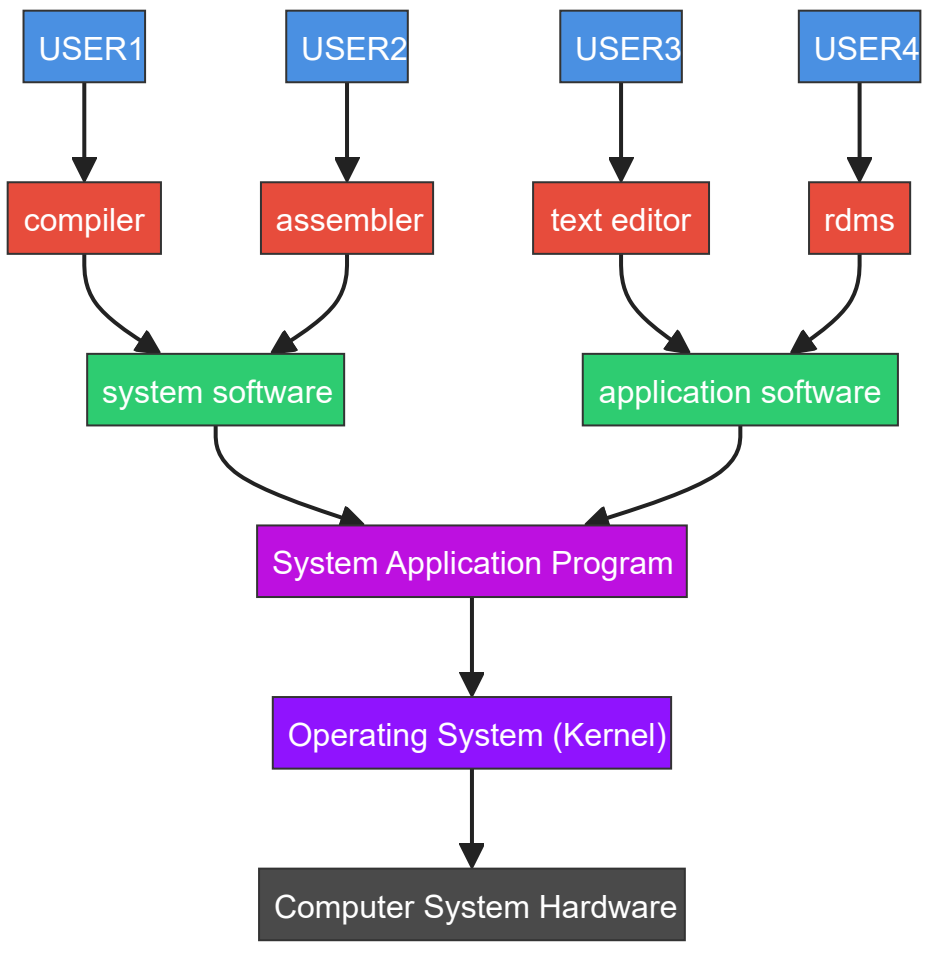
- Nowadays, windows and other operating system provide graphical user interface.
- Along with operating systems, microprocessors and operating systems have also evolved. This example of command-based operating systems which provides character-based user interfaces like CUI, GUI.

Kernel



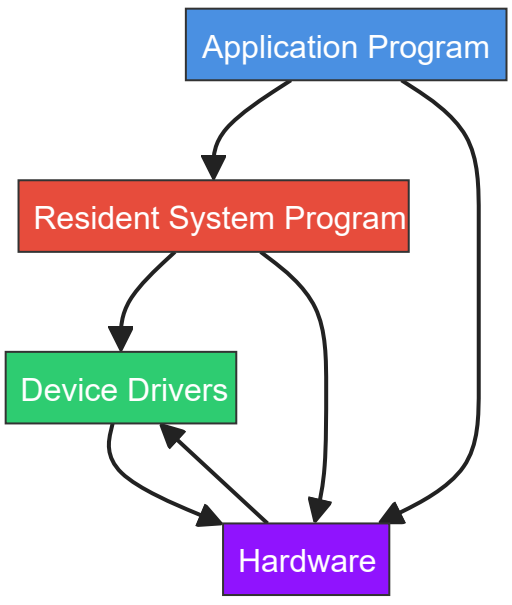
- The kernel is a computer program, at the core of operating system & generally it has complete control over the system
- Kernel prevents processes from conflicts.
- It act as a bridge between software applications and hardware components.
- The kernel manages system resources like memory, CPU and devices. Insuring everything is working smoothly and efficiently.
- It handles tasks like running different programs accessing files & connecting to device like printer, scanner etc.

Structure of Operating System



- It is the simplest structure of operating system and can be used only for small and limited system.
- It is not well-defined structure.
- Operating system acts as intermediate between user and computer system.

Simple Operating System Structure



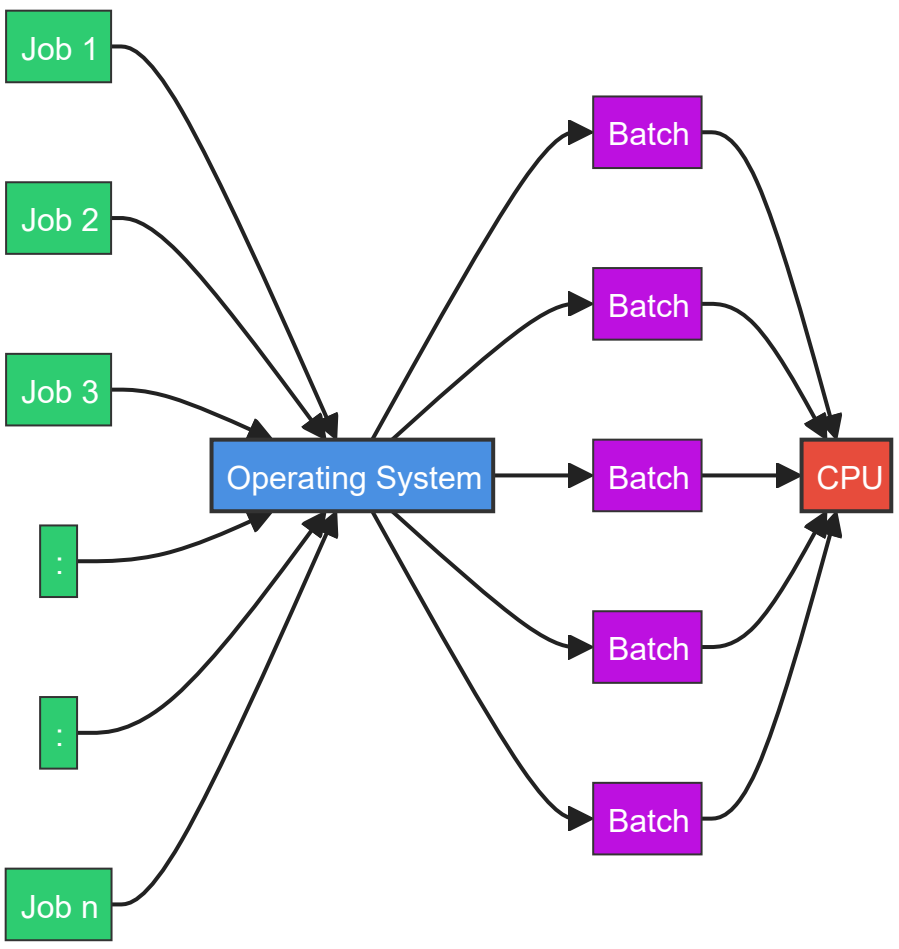
- This is a simple and not well-defined structure of operating system.

Types of Operating System

---

1. Batch Operating System

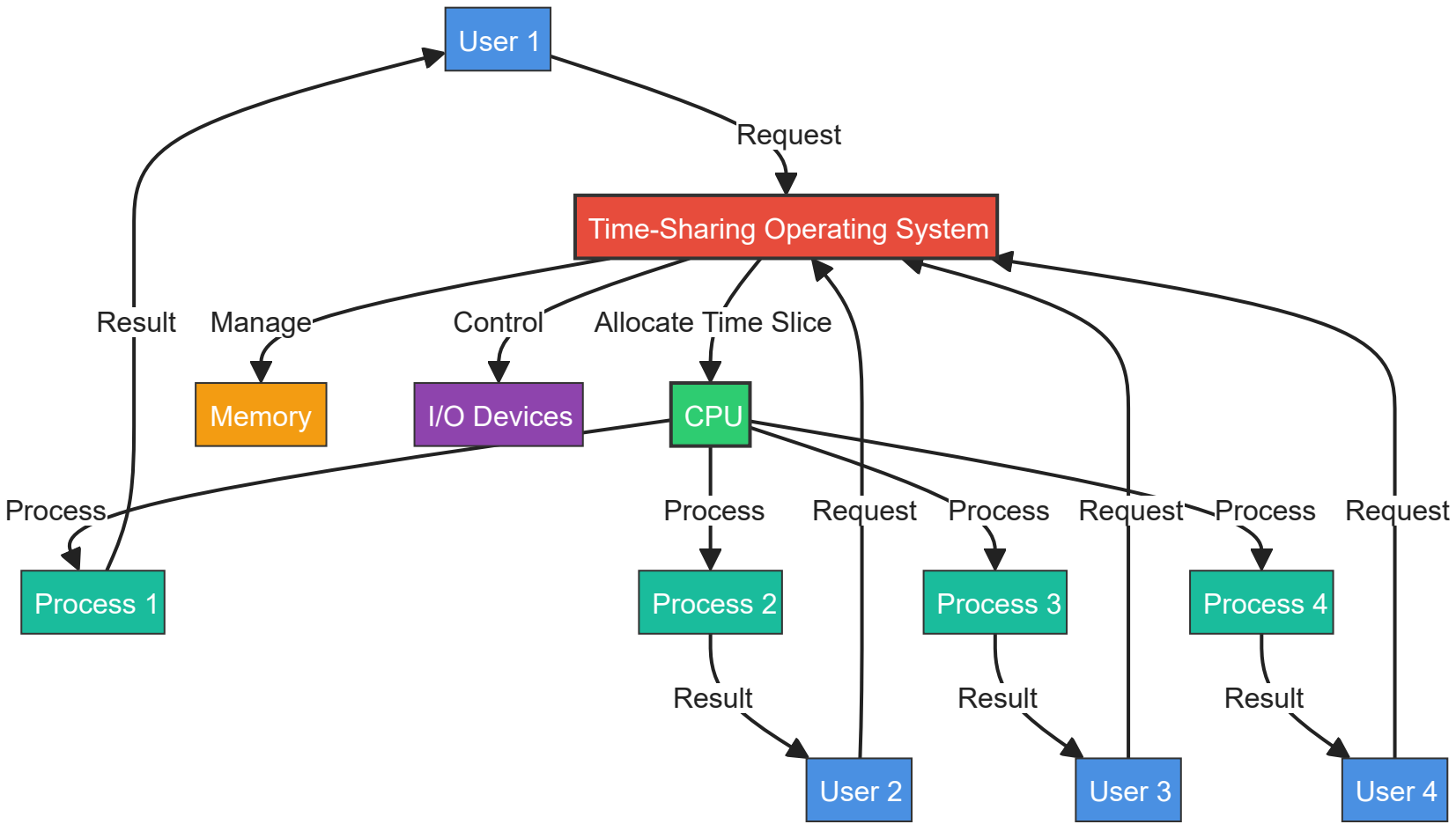
- Jobs are processed in batches without user interaction, jobs with similar needs are batched together and run through the system.
- Example: Early IBM mainframe operating system.



---

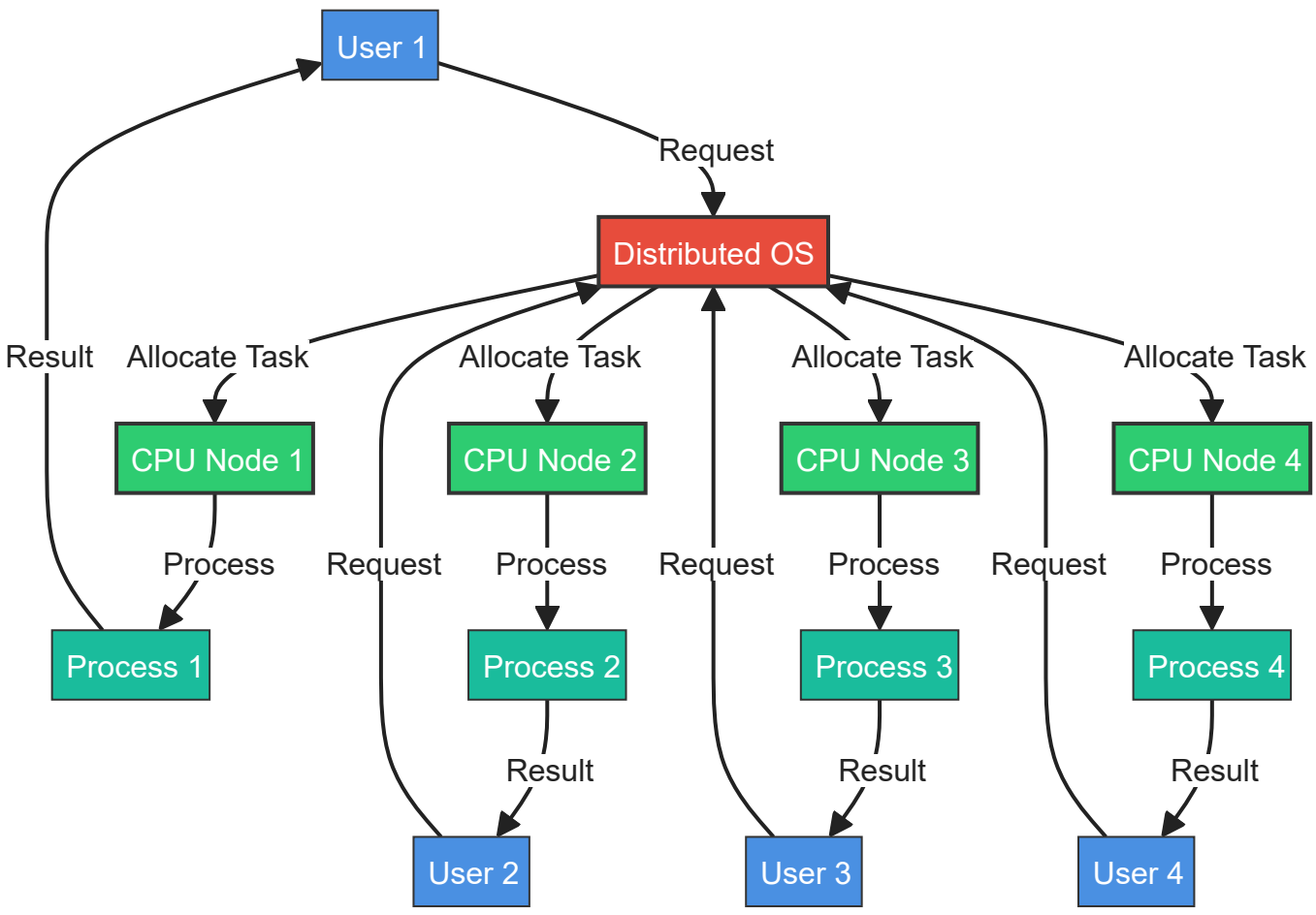
2. Time-Sharing Operating System

- Multiple Users can access the system concurrently. Time-sharing divides system time into small intervals and switches between tasks quickly, giving the illusion of simultaneous execution.
- Example: UNIX, Mulitics.



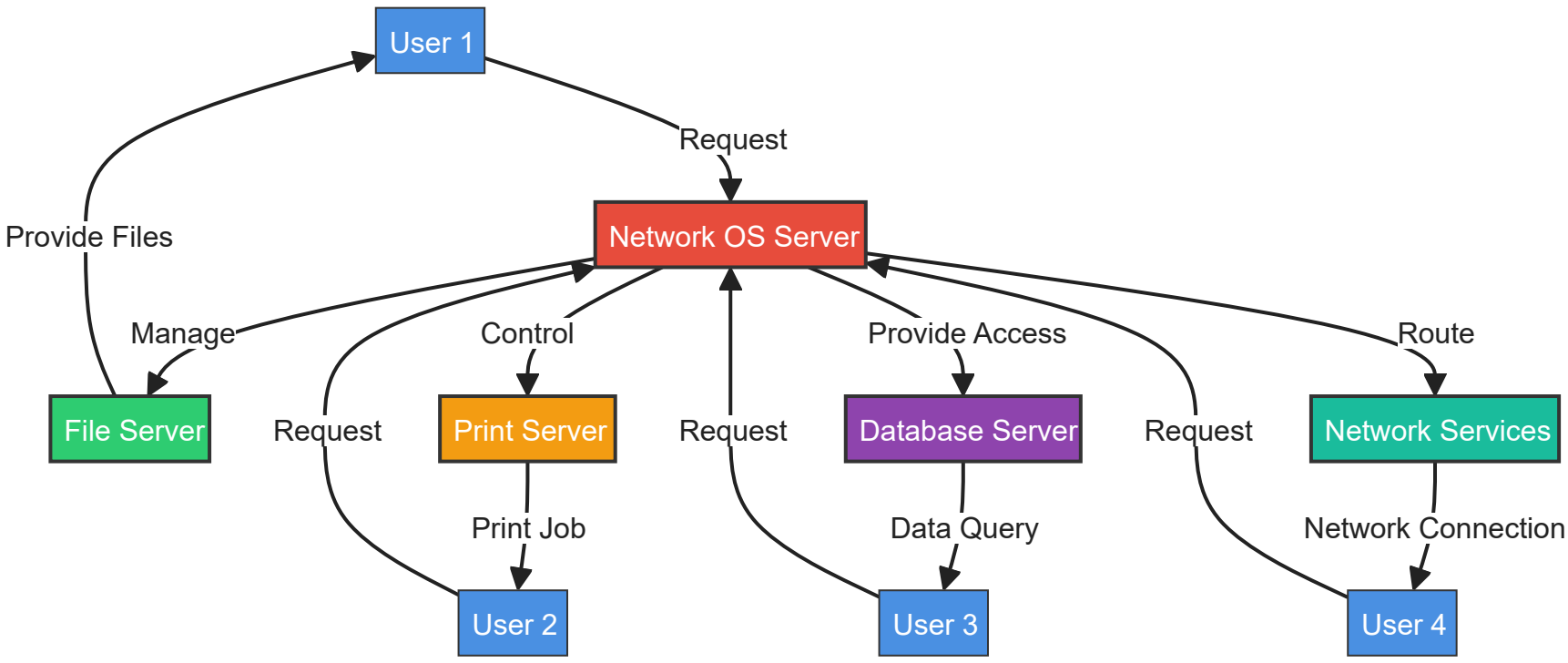
3. Distributed Operating System

- Manages a group of independent computers and makes them appear to be a single computer. It distributes computations among different machines to improve performance and efficiency.
- Example: Apache Hadoop, Amoeba.



4. Network Operating System

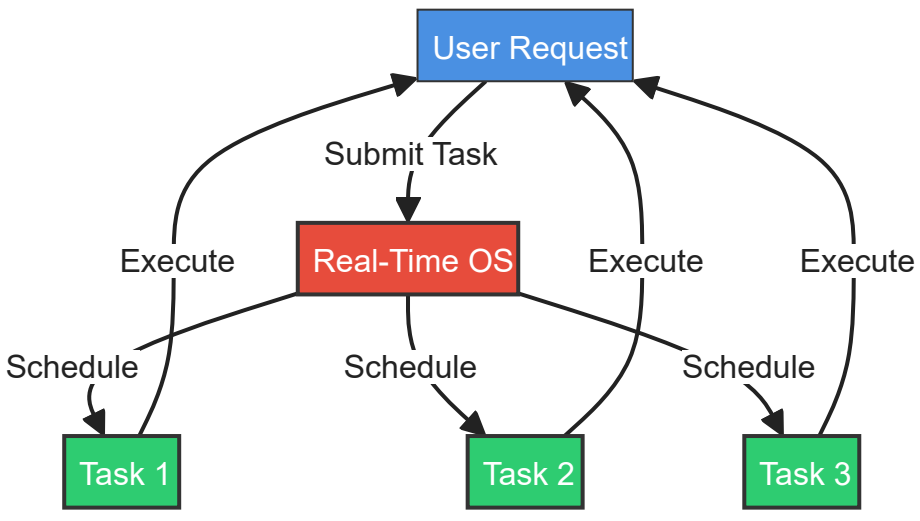
- Provides features for computers connected on network to communicate and share resources. Manages data, users, groups, security and applications over a network.
- Example; Novell Netware, Microsoft Windows Server.



---

5. Real-Time Operating System

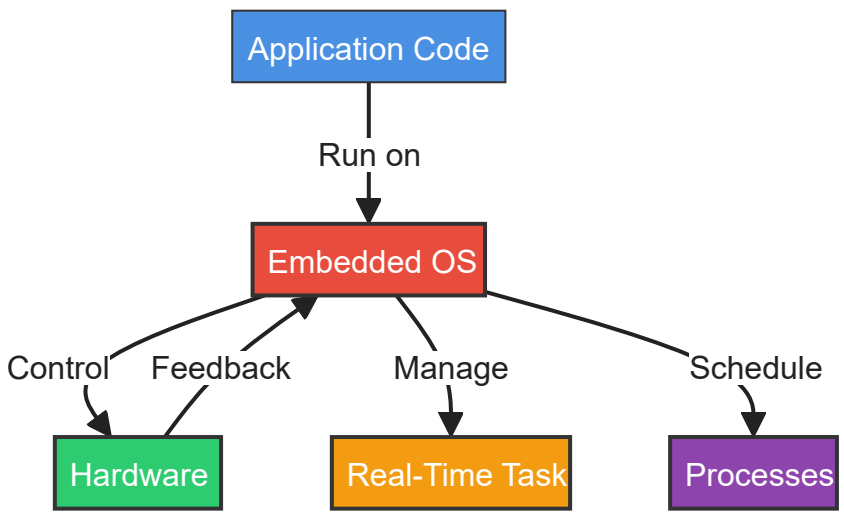
- Designed to process data as it comes in, typically without buffer delays, used in environments where time constraints are critical.
- Types:
  1. Hard Real-Time Systems:
     - Strict timing constraints, missing a deadline can result in system failure.
  2. Soft Real-Time Systems:
     - More flexible, deadlines can occasionally be missed without catastrophic consequences.
- Examples: Vxworks, Free RTOS.



---

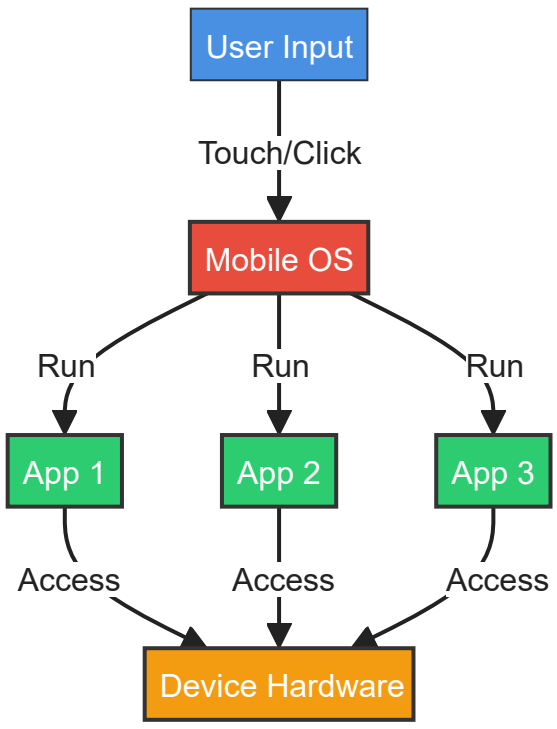
6. Embedded Operating System

- Designed to operate on small machines like PDAs with limited resources. They are typically specialized for specific tasks.
- Example: Embedded Linux, embedded windows.



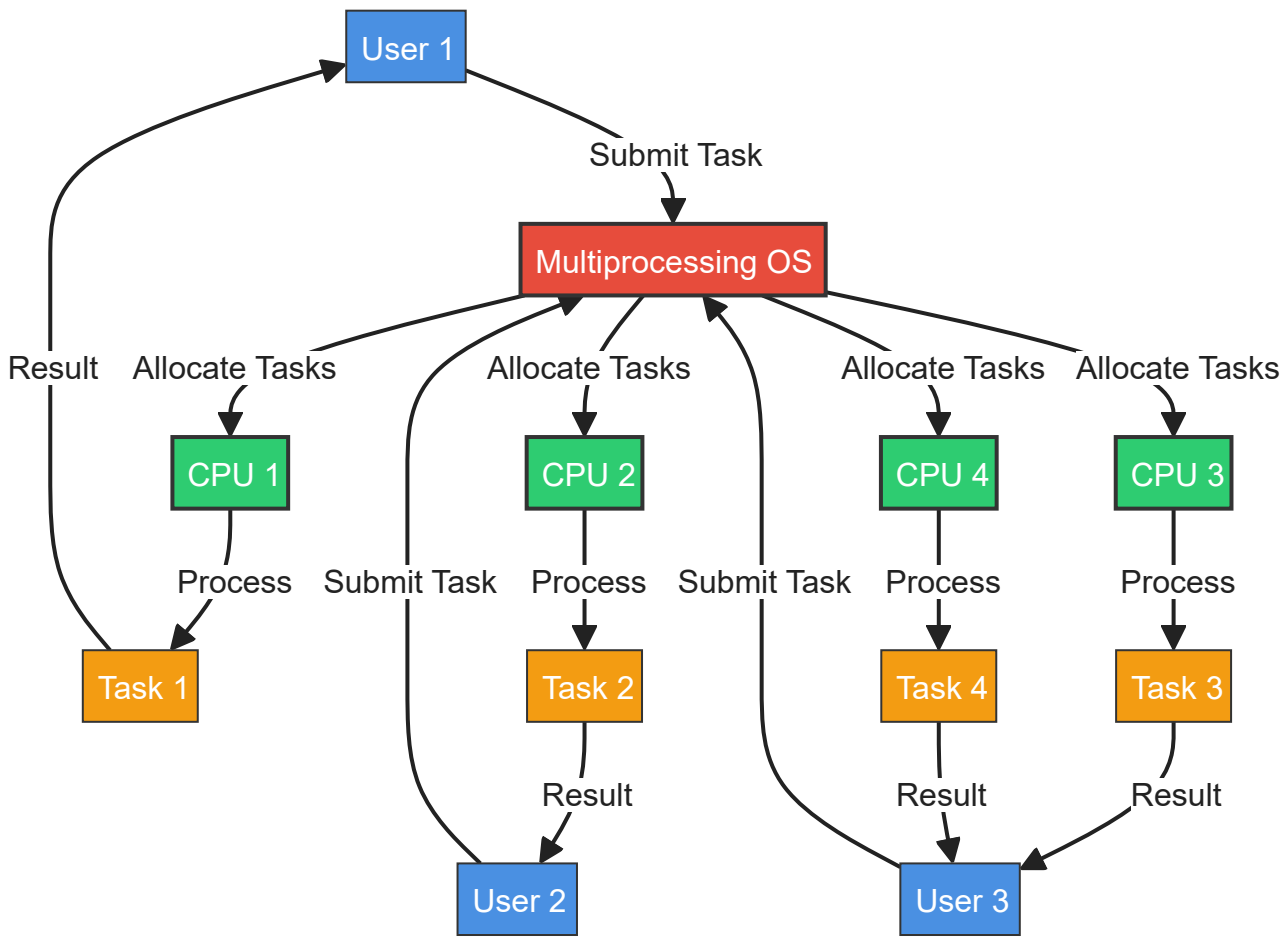
7. Mobile Operating System

- Designed specifically for mobile devices such as smartphones and tablets. These OSs are optimized for mobile specific features like touch interfaces and low power consumption.
- Example: Android, iOS, Linux.



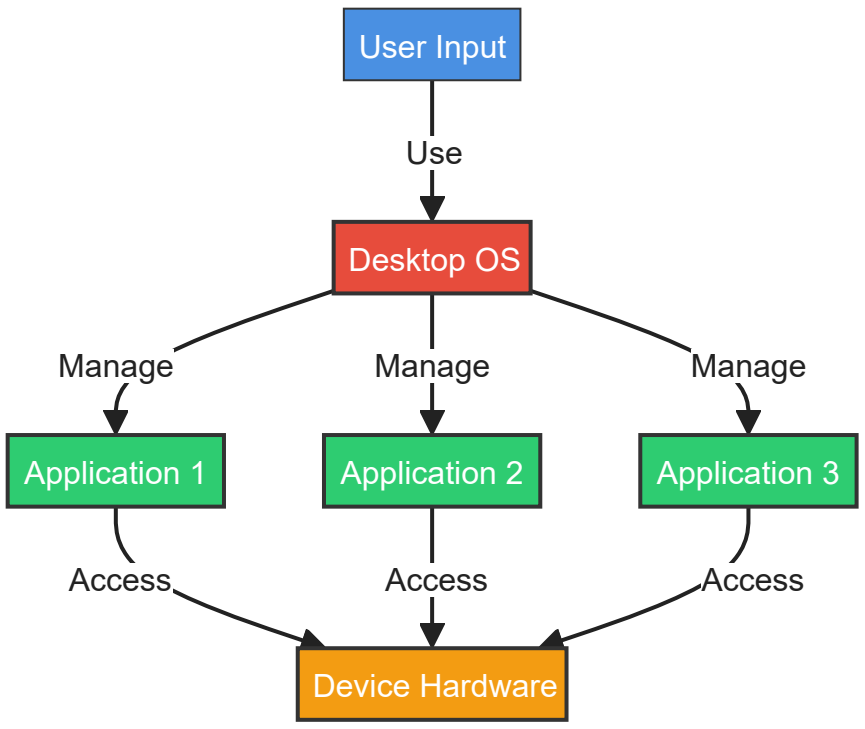
8. Multiprocessing Operating System

- Supports the use of more than one processor at the same time. These systems can execute multiple processes simultaneously.
- Example: Linux, UNIX.



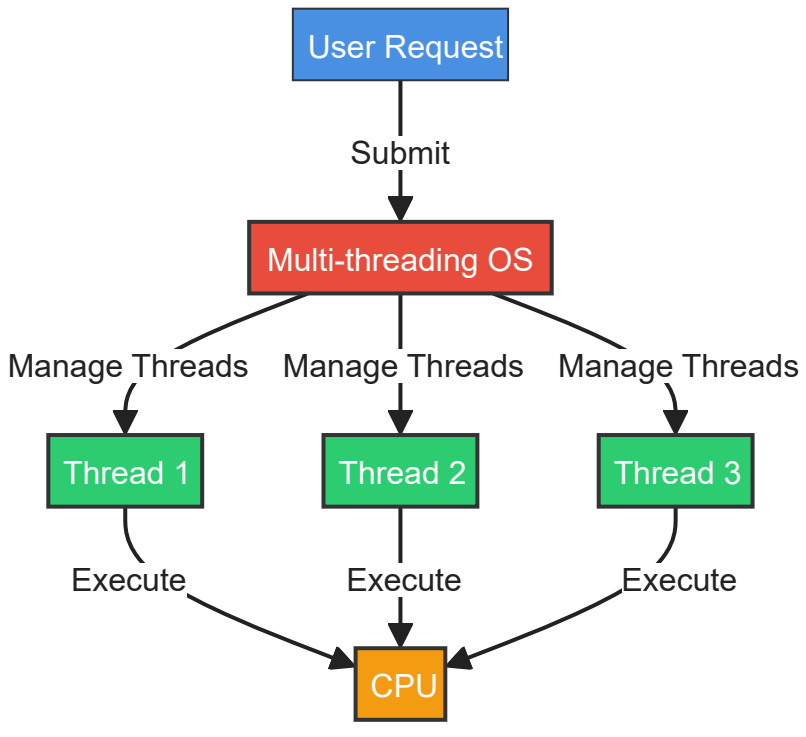
9. Desktop Operating System

- Intended for use on personal computers, these OSs support a wide range of applications and peripherals.
- Example: Microsoft Windows, macOS, Linux distributions.

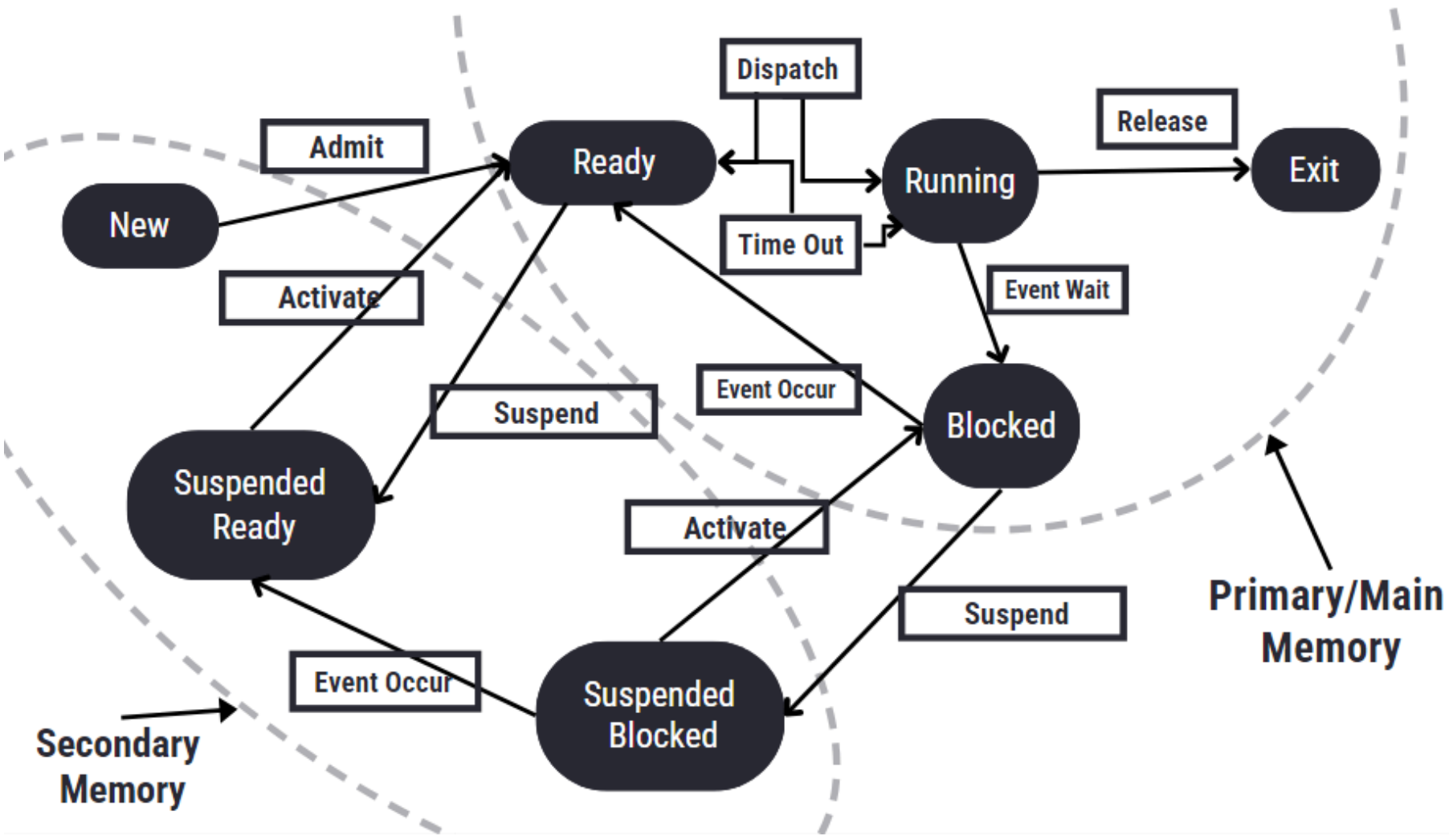


10. Multi-threading Operating System

- Allows different parts of a single programs to run concurrently, multi-threading improves the performance of applications by splitting tasks into smaller threads.
- Examples: Modern versions of windows, Linux.



Process state transition diagram



Scheduling

- Process is a program in execution, one process can have multiple threads(light weight processes) scheduling is important in many computer environments.
- One of the most important areas of scheduling is which program will work on the CPU. This is handled by the operating system. There are many different ways in which we can choose to configure program.

Process Schedulers

- Are the fundamental components of operating system, responsible for deciding the order in which processes are executed by the CPU.
- In simple terms they manage how the CPU allocates it's time among the multiple processes. That are competing for it's attention.

Process Scheduling

- Is the activity of process manager. That Handles the removal of running process from the CPU and the selection of another process based on a particular strategy.
- Process scheduling is an essential part of multi-programming operating system, such O.S allow more than one process to be loaded into executable memory at that time and the loaded process shares the CPU using time multi-plexin.

Categories

- **Primitive**
  - In this case operating system assigns resources to process for a pre-determined period. The process switches from running state to ready state or from waiting state to ready state during resource allocation this switching happens

because CPU may give other process priority and substitute the currently active process for the higher priority process.

- **Non-primitive**
  - In this case process's resource can't be taken before the process has finished running. When a running process finishes and transition to a wait state resources are utilized.

Types of Schedulers

Long-Term Schedulers

- **Purpose**: Decides which processes are admitted into the system for execution.
- **Function**: Controls the admission of new processes from a queue into the system's main memory (RAM). It helps in managing the process mix and balancing the load in the system.
- **Example**: When you start a new application on your computer, the long-term scheduler decides when and if that application should be loaded into memory for execution.

Medium-Term Schedulers

- **Purpose**: Manages processes that are in the main memory and can be swapped in and out of the main memory to the disk.
- **Function**: Controls which processes should be kept in memory and which should be temporarily moved to secondary storage (disk) to free up memory. It helps in balancing between processes that are running and those that are suspended.
- **Example**: If your computer is running low on RAM, the medium-term scheduler might move some processes to disk storage to free up memory for other processes.

Short-Term Schedulers

- **Purpose**: Decides which of the processes in memory should be executed next by the CPU.
- **Function**: Handles the process switching at a much faster rate compared to the other schedulers. It ensures that the CPU is always busy by selecting processes that are ready to run and allocating CPU time to them.
- **Example**: When you have multiple applications open, the short-term scheduler decides which application's process gets CPU time next.

Some Other Types of Schedulers

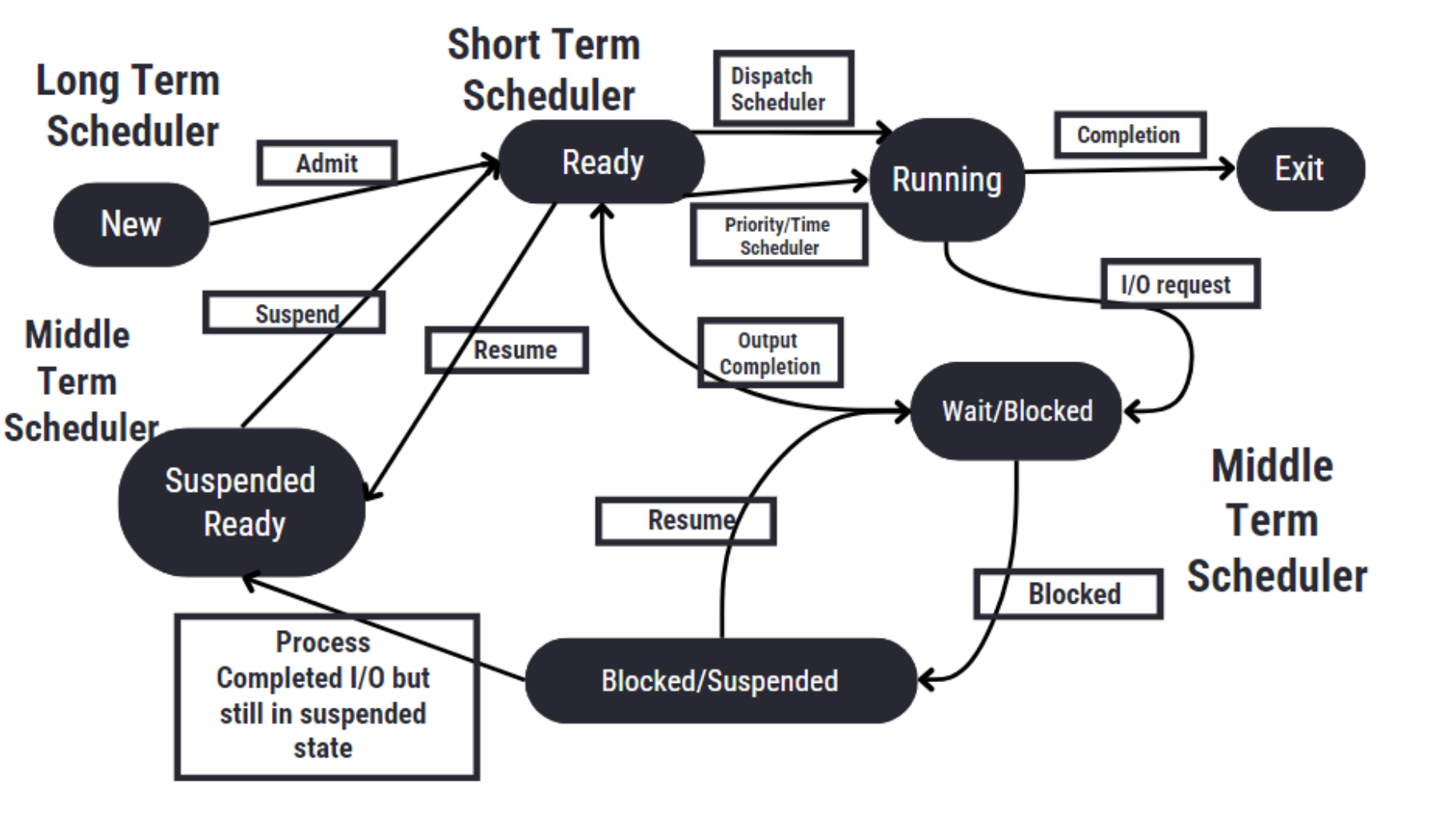
I/O Schedulers

- **Purpose**: Manages the order of I/O operations (such as reading from or writing to disk).
- **Function**: Optimizes the performance of I/O operations by deciding the order in which I/O requests should be processed. It helps in reducing waiting times and improving system efficiency.
- **Example**: If multiple programs are requesting to read from a disk, the I/O scheduler decides the sequence in which these requests are handled to minimize delays and maximize throughput.

Real-Time Schedulers

- **Purpose**: Ensures that real-time tasks are completed within specified time constraints.
- **Function**: Manages tasks that require timely and predictable execution, often used in systems where missing deadlines could lead to significant issues (e.g., embedded systems, control systems).
- **Example**: In an autonomous vehicle, the real-time scheduler ensures that the system's response to sensor inputs and control commands happens within strict time limits to maintain safe operation.

Diagram



Criteria of Scheduling

1. CPU Utilization

- **Description**: Measures the percentage of time the CPU is actively working on tasks.
- **Goal**: Maximize CPU utilization to ensure the CPU is being used effectively and not idle.

2. Throughput

- **Description**: The number of processes completed per unit of time.
- **Goal**: Maximize throughput to increase the number of processes finished in a given time frame.

3. Turnaround Time

- **Description**: The total time taken from the submission of a process to its completion.
- **Goal**: Minimize turnaround time to improve the overall efficiency and responsiveness of the system.

---

4. Waiting Time

- **Description**: The total time a process spends waiting in the ready queue before being executed.
- **Goal**: Minimize waiting time to reduce the time processes spend waiting for CPU access.

---

5. Response Time

- **Description**: The time from the submission of a request until the first response is produced (not the output).
- **Goal**: Minimize response time to improve the interactive experience for users.

---

6. Fairness

- **Description**: Ensures that each process receives a fair share of the CPU and other resources.
- **Goal**: Provide equitable access to resources for all processes, preventing starvation or unfair delays.

---

7. Priority

- **Description**: The ability to prioritize processes based on their importance or urgency.
- **Goal**: Ensure that high-priority processes receive more immediate attention and resources.

---

8. Predictability

- **Description**: The ability to predict the execution time of processes based on the scheduling algorithm.
- **Goal**: Achieve predictable performance to ensure that processes meet their deadlines, especially in real-time systems.

---

9. Overhead

- **Description**: The extra time and resources spent on managing the scheduling process, such as context switching.
- **Goal**: Minimize scheduling overhead to ensure efficient use of system resources.

---

10. Scalability

- **Description**: The ability of the scheduling algorithm to handle an increasing number of processes without significant performance degradation.

- **Goal**: Ensure the scheduler remains effective as the number of processes or the system size grows.

---

11. Resource Utilization

- **Description**: Efficient use of various system resources (CPU, memory, I/O).
- **Goal**: Maximize the utilization of all available resources to improve overall system performance.

Concurrency - Introduction

Concurrency is the ability of an operating system to execute multiple tasks simultaneously, allowing for efficient utilization of resources and improved performance. In today's computing environment, with multicore processors and high-speed networking, concurrent processing has become increasingly important for operating systems to meet user demands.

Definition of Concurrent Processes

Concurrent processing, also known as concurrent execution, refers to the ability of an operating system to execute multiple tasks or processes at the same time. It involves the parallel execution of tasks, with the operating system managing and coordinating these tasks to ensure they do not interfere with each other.

Concurrency is typically achieved through techniques such as process scheduling, multithreading, and parallel processing. It is a critical technology in modern systems, enabling them to provide the performance, scalability, and responsiveness required in today’s computing environments.

Importance of Concurrent Processing in Modern Operating Systems

---

Concurrent processing plays a significant role in modern operating systems due to the following reasons:

1. **Improved Performance**: With the advent of multicore processors, modern operating systems can execute multiple threads or processes simultaneously, leading to improved system performance. Concurrent processing allows the operating system to make optimal use of available resources, thereby maximizing system throughput.
2. **Resource Utilization**: Concurrent processing allows for better utilization of system resources, such as CPU, memory, and I/O devices. By executing multiple threads or processes concurrently, the operating system can utilize idle resources effectively, leading to enhanced resource utilization.
3. **Enhanced Responsiveness**: Concurrent processing improves system responsiveness by enabling the operating system to handle multiple tasks simultaneously. This is especially important in interactive applications, where quick responses to user inputs are critical.

Concurrency Processing in Operating Systems

Concurrency is the ability of an operating system to execute multiple tasks seemingly simultaneously, optimizing resource use and boosting performance. This is essential in modern computing due to multicore processors and high-speed networking.

Types of Concurrency Processing:

- **Process Scheduling:** This fundamental form sequentially executes multiple processes, with each receiving a time slice before the next takes over. Imagine this as a juggling act, where the operating system rapidly switches between processes to give the illusion of simultaneous execution.
- **Multithreading:** A process is divided into multiple threads that share the same memory space and resources. This division of labor within a process leads to enhanced performance and efficiency. Consider it similar to a team working on different aspects of a project concurrently.
- **Parallel Processing:** Multiple processors or cores execute tasks simultaneously, distributing the workload for significant performance gains. This is analogous to having multiple workers tackling different parts of a task simultaneously, speeding up completion.
- **Distributed Processing:** Tasks are distributed across multiple interconnected computers or nodes, enhancing scalability and fault tolerance. This is similar to a large project being divided into smaller tasks, each handled by a different team, leading to faster completion and resilience to individual team failures.

Independent Processes

Independent processes, as the name suggests, operate without sharing resources or memory. They interact through inter-process communication mechanisms, ensuring a high level of isolation and security.

Think of them as individuals working on separate projects in their own workspaces. They don't interfere with each other's work or share resources.

Dependent Processes

In contrast to independent processes, **dependent processes** rely on shared resources or data. This interdependency, while potentially increasing efficiency, introduces the risk of data inconsistency and race conditions if not managed correctly.

Imagine multiple chefs sharing the same kitchen and ingredients. While they can potentially prepare a meal faster by collaborating, they need to coordinate their actions to avoid conflicts and ensure the dish is prepared correctly.

Introduction to Process Synchronization

Process synchronization is crucial in managing dependent processes. It ensures that processes access shared resources in a controlled manner, preventing conflicts and maintaining data consistency.

This coordination is typically achieved using mechanisms like semaphores, critical sections, and monitors. These tools act like traffic signals, ensuring that only one process can access a shared resource at a time, thus preventing accidents and ensuring smooth operation.

To illustrate, consider a shared bank account accessed by multiple users (processes). Without synchronization, if two users try to withdraw money simultaneously, it could lead to an incorrect balance. Process synchronization mechanisms prevent such conflicts by ensuring that only one user can access and modify the account balance at any given time.

Process Synchronization in Operating Systems

- **Process synchronization** is the coordination of multiple processes in a multitasking operating system to ensure that they do not interfere with each other and produce inconsistent data.
- The main objective of process synchronization is to ensure that multiple processes do not interfere with each other's execution to prevent inconsistent data.

- To achieve this, various synchronization techniques, such as **semaphores**, **monitors**, and **critical sections**, are used. These techniques help in ensuring data consistency and integrity to avoid synchronization problems.
- Process synchronization is a crucial aspect of modern OSs as it directly impacts the speed and efficient functioning of multitasking systems.

What is a Process?

- A process is a program in execution. It is also called a running or active program. It includes the program's code, data, and the activity it intends to perform.
- The CPU, memory, and other resources are used by the process.

Types of Processes

Based on the level of synchronization, processes are categorized into two types:

1. **Independent Process:** The execution of one process doesn't affect the execution of other processes.
2. **Co-operative Process:** Processes that can affect or be affected by other processes executing in the system.

- Process synchronization problems mainly arise in the case of co-operative processes because resources are shared in co-operative processes.

Race Condition

- A **race condition** is a situation that may occur when more than one process is trying to access and modify the same shared data concurrently. This can lead to unpredictable and incorrect outcomes, depending on the order in which the processes access and modify the shared data.
- If different processes access the same memory location concurrently, then the outcome depends on the particular order in which they access.
- Race conditions usually occur inside a critical section.
- A critical section is a code segment where shared resources are accessed and modified. Only one process is allowed to execute within the critical section at any given time to avoid race conditions.
- If the critical section is treated as an atomic instruction, race conditions can be avoided. Atomic instructions are executed as a single, indivisible unit, ensuring that no other process can interfere with the execution.
- Using locks or atomic variables can prevent race conditions.

Producer-Consumer Problem

- The producer-consumer problem, a classic example of a synchronization problem, arises in situations where one or more processes (producers) generate data and other processes (consumers) consume that data.
- Both the producer and consumer share a common, fixed-size buffer. The producer puts data into the buffer, and the consumer takes it out.
- The fixed-size buffer creates a constraint. Producers cannot add data to a full buffer, and consumers cannot remove data from an empty buffer.

Solution for Producer-Consumer Problem

- The solution involves synchronizing the producer and consumer processes using semaphores or other synchronization primitives.
- The producer checks if the buffer is full. If the buffer is full, the producer goes to sleep. Otherwise, the producer produces data and puts it into the buffer. Then, the producer wakes up any sleeping consumer.
- The consumer checks if the buffer is empty. If it is empty, the consumer goes to sleep. Otherwise, the consumer consumes data/removes it from the buffer and then wakes up the sleeping producer.

Introduction to Semaphore in OS

- A semaphore is a hardware or software-based synchronization tool used to solve the critical section problem in concurrent programming. It controls access to shared resources by multiple processes.
- It can be thought of as a synchronization variable that can be accessed and modified atomically by different processes or threads to ensure that only one of them is in the critical section at any given time.

What is the Problem of Critical Section?

- The critical section problem is a classic synchronization problem that arises when multiple processes try to access and modify shared resources concurrently.
- A critical section is a code segment where shared resources are accessed. Only one process should be allowed to execute in the critical section at a time.
- This is because if multiple processes are allowed to access the critical section simultaneously, it can lead to data inconsistency and race conditions.

How Semaphore Solves the Critical Section Problem

- A semaphore is a non-negative integer variable. Its least value is 0. It can't be negative.
- A semaphore uses two atomic operations to solve the critical section problem: **wait()** and **signal()**.
- The **wait()** operation, also known as P(), is used for deciding the entry condition for a process into the critical section. If the semaphore value is greater than 0, it decrements the semaphore value by 1 and allows the process to enter the critical section. However, if the semaphore value is 0, the process will be put to sleep until the value becomes greater than 0.
- The **signal()**, also known as V(), operation signals the completion of a process in the critical section. It increments the semaphore value by 1. If there are processes waiting to enter, one of them is woken up.

By using semaphores, the critical section problem can be effectively solved, ensuring data consistency and preventing race conditions.

Got it! I'll clarify the explanations and mention the types of graphs or diagrams typically associated with each concept so you can explore further or find suitable images:

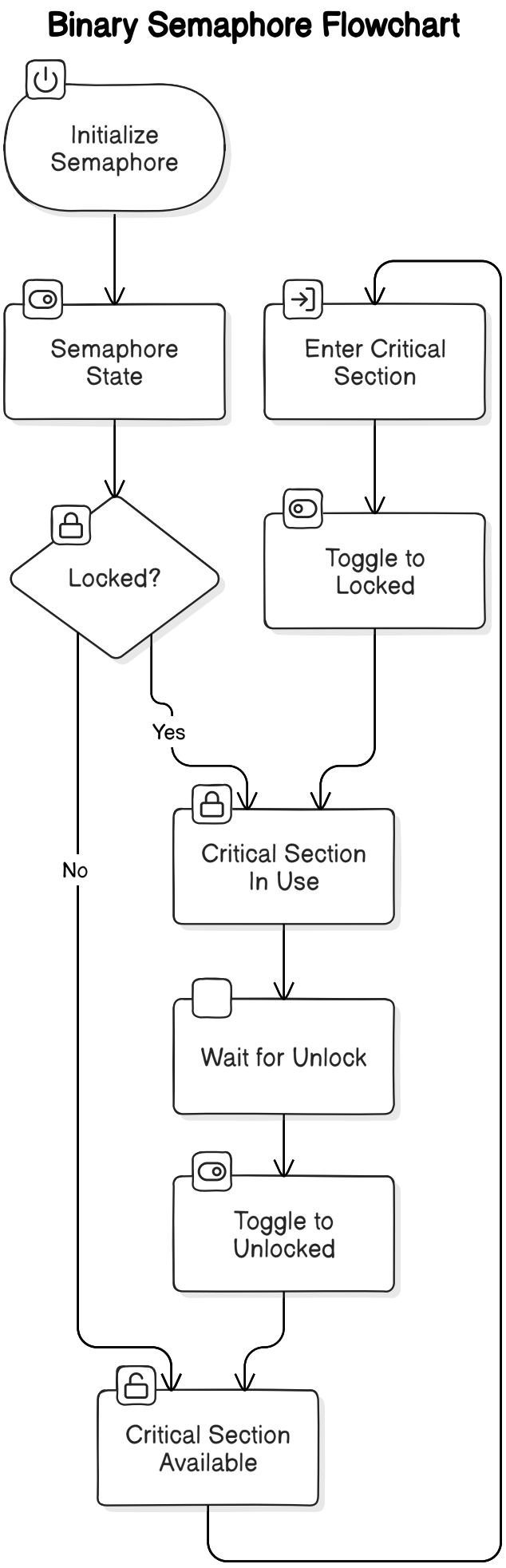
Types of Semaphores

---

1. Binary Semaphore (Mutex)

- A binary semaphore, or mutex, is often represented using simple diagrams showing only two states (locked and unlocked). It has only two values, 0 or 1, and is used for mutual exclusion. When a process accesses a critical section, it "locks" the

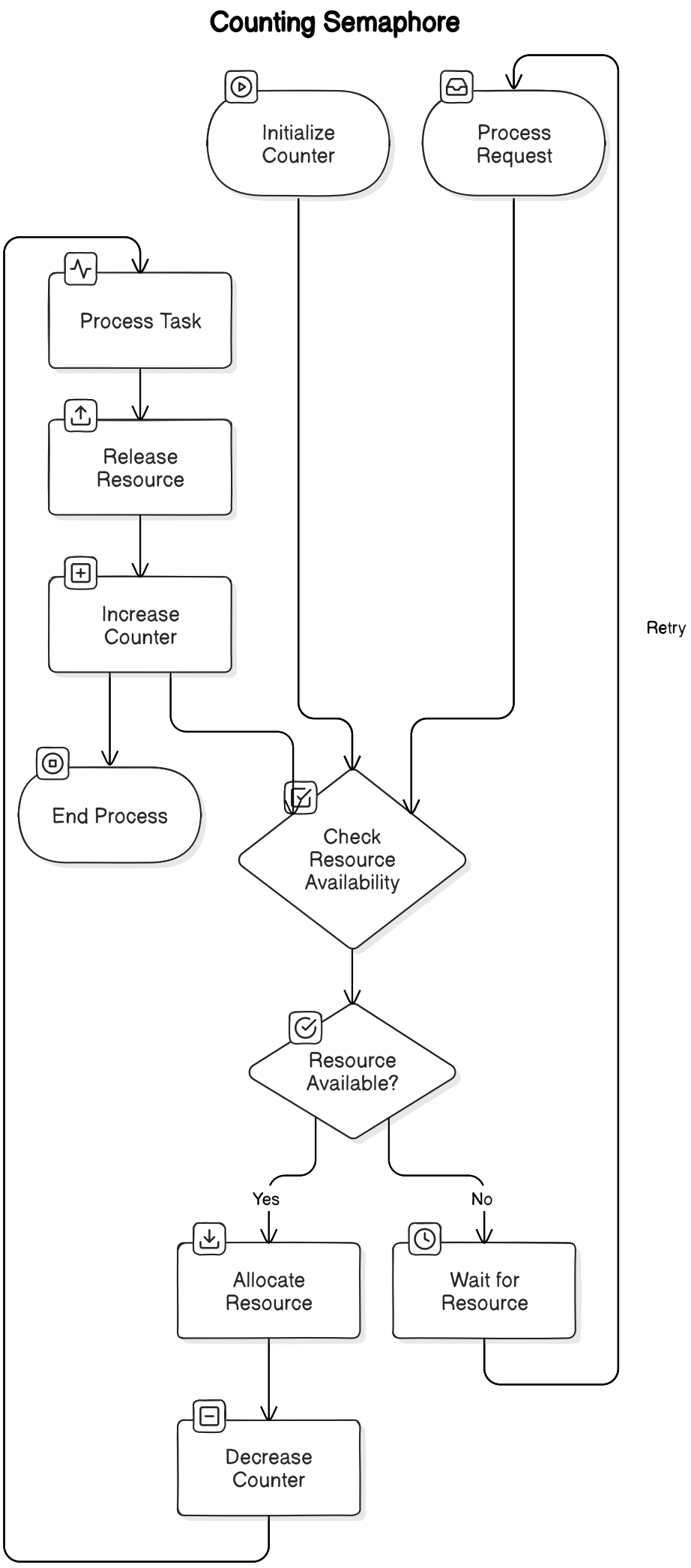
semaphore by setting it to 0. When the process finishes, it "unlocks" the semaphore by setting it back to 1.



2. **Counting Semaphore**:

- A counting semaphore keeps track of multiple identical resources. It is initialized to the number of available resources, and each time a process acquires a resource, the semaphore decrements. When a process releases the resource, it increments

the semaphore.



Deadlocks in Operating Systems

What is Deadlock?

In computing, a **deadlock** is a scenario where two or more processes cannot proceed because each process is waiting for another to release a resource. It's like two people trying to pass each other in a narrow hallway but neither moves aside, causing a standstill. In an operating system, deadlocks cause processes to become stuck in a state where none can proceed without external intervention.

Example

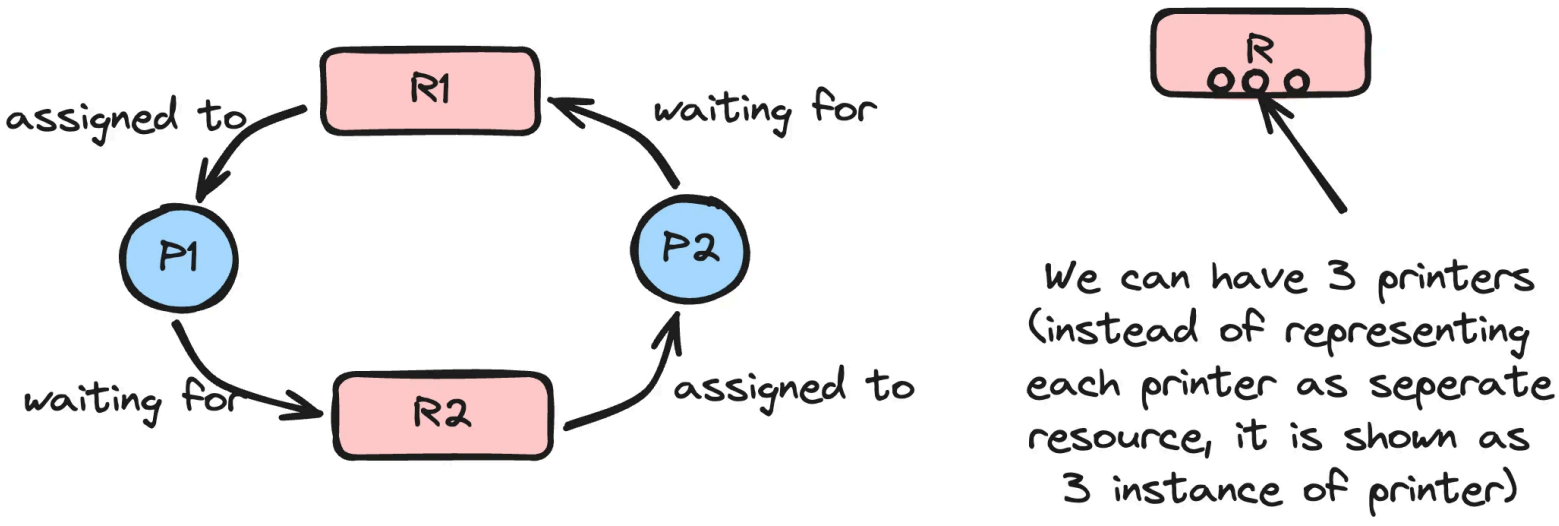
Imagine two trains on a single track, both heading toward each other. If they meet in the middle, neither can move forward unless the other reverses — but both are waiting for the other to do so. This is the essence of a deadlock.

How Deadlock Occurs in Operating Systems

A process in an operating system uses resources in the following sequence:

1. **Requests** a resource.
2. **Uses** the resource.
3. **Releases** the resource.

Deadlocks occur when each process holds a resource and waits for another process to release a different one. Let’s break it down with the following diagram:



Here, Process 1 holds Resource 1 and is waiting for Resource 2, which is held by Process 2. Process 2 is waiting for Resource 1, forming a cycle that results in deadlock.

Necessary Conditions for Deadlock

For a deadlock to occur, four conditions must be satisfied simultaneously:

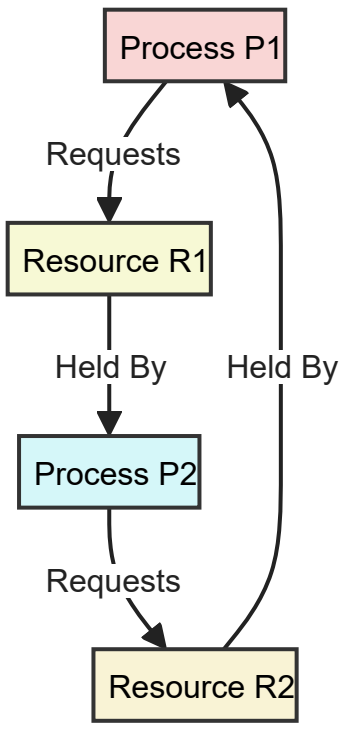
1. **Mutual Exclusion**: At least one resource is non-shareable. For instance, only one process can use a printer at a time.
2. **Hold and Wait**: A process is holding at least one resource and waiting to acquire additional resources.
3. **No Preemption**: Resources cannot be forcibly removed from a process holding them.
4. **Circular Wait**: A circular chain of processes exists, where each process is waiting for a resource held by the next process in the chain.

Deadlock Detection

Deadlock Detection involves the operating system determining whether a deadlock exists by checking for circular waits or other indicators of deadlock. Algorithms used for detection include:

- **Resource Allocation Graph (RAG) Algorithm**: Detects deadlocks by examining the resource allocation graph for cycles.

Here's a visual representation of a deadlock situation:



This cycle shows that no process can proceed without a resolution.

Methods for Handling Deadlocks

There are three main approaches to handle deadlocks:

1. Deadlock Prevention

In **deadlock prevention**, we try to avoid one of the necessary conditions for deadlock. This can be done in several ways:

- **Mutual Exclusion**: For shared resources, don’t lock them. However, this is only possible for resources that can be shared (e.g., reading a file).
- **Hold and Wait**: Ensure that a process requests all necessary resources at once before it starts. This can lead to inefficiency as processes may hold onto resources longer than needed.
- **No Preemption**: Preempt resources from lower-priority processes. This may lead to **livelock**, where processes continually release and request resources without making progress.
- **Circular Wait**: Impose an ordering on the resources. For example, always request resources in a fixed sequence to avoid circular waits.

---

2. Deadlock Avoidance

Deadlock avoidance is more strategic. Before a process starts, the system ensures that granting a resource will not lead to a deadlock. The **Banker's Algorithm** is often used here.

Banker's Algorithm

In the Banker's Algorithm, each process declares its maximum resource needs upfront. The system then checks whether it can allocate resources to a process without entering a deadlock. The algorithm essentially "plays it safe" and avoids unsafe states.

Let's break it down with an example:

Step-by-Step Example of Banker's Algorithm

Given the available resources:

- Total resources: A = 10, B = 5, C = 7
- Processes: P1, P2, P3
- Allocated and needed resources are represented as follows:

Process	Allocation (A, B, C)	Max Need (A, B, C)	Remaining Need (A, B, C)
P1	1, 0, 0	7, 5, 3	6, 5, 3
P2	2, 1, 1	3, 2, 2	1, 1, 1
P3	3, 2, 2	9, 0, 2	6, 0, 0

The system checks if it can allocate resources to each process and whether the process can finish execution without causing a deadlock.

Here’s the **Banker's Algorithm** implemented in C :

Code: Banker's Algorithm in C

```c
#include <stdio.h>

int n = 5, m = 3;  // Number of processes and resource types
int alloc[5][3] = { {0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2} };
int max[5][3] = { {7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3} };
int avail[3] = {3, 3, 2};
int need[5][3];
int finished[5], safeSequence[5];

void calculateNeed() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
}

int isSafe() {
    int work[3];
    for (int i = 0; i < m; i++) {
        work[i] = avail[i];  // Copy the available resources into work[]
    }

    // Initialize all processes as unfinished
    for (int i = 0; i < n; i++) {
        finished[i] = 0;
    }

    int idx = 0;
    for (int count = 0; count < n; count++) {
        int found = 0;
        for (int i = 0; i < n; i++) {
            if (!finished[i]) {
                int j;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > work[j]) {
                        break;
                    }
                }

                if (j == m) {  // If all need[i][j] <= work[j]
                    for (int k = 0; k < m; k++) {
                        work[k] += alloc[i][k];  // Add allocated resources of Pi to work[]
                    }
                    safeSequence[idx++] = i;
                    finished[i] = 1;  // Mark this process as finished
```

```
                    found = 1;
                }
            }
        }

        if (!found) {
            printf("System is in an unsafe state.\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence: ");
    for (int i = 0; i < n; i++) {
        printf("P%d ", safeSequence[i]);
    }
    printf("\n");

    return 1;
}

int main() {
    calculateNeed();  // Calculate the need matrix

    if (isSafe()) {
        printf("Safe sequence exists. No deadlock.\n");
    } else {
        printf("Deadlock detected. No safe sequence.\n");
    }

    return 0;
}
```

Output:

```
System is in a safe state.
Safe sequence: P1 P3 P4 P0 P2
Safe sequence exists. No deadlock.
```

How It Works:

- **Initialization**: The predefined values represent 5 processes (P0 to P4) and 3 types of resources. The allocation matrix, max matrix, and available resources are hard-coded.
- **Need Matrix**: The `need` matrix is calculated as `need[i][j] = max[i][j] - alloc[i][j]` for each process.
- **Safety Check**: The algorithm checks if there exists a sequence in which all processes can execute without causing a deadlock. If such a sequence exists, the system is safe, and it prints the safe sequence.

In this example, the safe sequence is `P1, P3, P4, P0, P2`.

---

3. Deadlock Recovery

If prevention and avoidance are not used, deadlocks can still be detected and then resolved through recovery methods:

- **Manual Intervention**: A system administrator manually kills processes or reallocates resources.
- **Automatic Recovery**: Processes are terminated or resources are preempted by the system automatically.
- **Process Termination**: In the most extreme cases, all deadlocked processes are aborted, or they are aborted one by one to break the deadlock.
- **Resource Preemption**: Resources held by a deadlocked process are forcibly taken away, and the process is rolled back to a safe state.

Deadlock vs. Starvation

While **deadlock** implies that processes are stuck waiting for each other indefinitely, **starvation** occurs when a process waits too long because resources are consistently allocated to higher-priority processes. The difference is subtle but important:

Aspect	Deadlock	Starvation
Definition	Processes are stuck waiting for each other.	A process is perpetually denied resources.
Cause	Circular dependency on resources.	Unfair resource allocation policies.
Resolution	Requires external intervention.	Can be mitigated by adjusting priority levels.

Conclusion

Deadlocks can severely impact system performance and reliability, especially in large-scale, multi-threaded environments. Various techniques, from prevention and avoidance to detection and recovery, are used to handle deadlocks. Understanding these methods is key to ensuring efficient system design and operation.