Operating System

1. Basics of Operating Systems

Introduction to Operating Systems:

An **Operating System (OS)** is system software that manages computer hardware and software resources and provides services for computer programs. It serves as an intermediary between users and the computer hardware.

Main Functions of an Operating System:

- **Process Management**: Handles the execution of multiple processes.
- **Memory Management**: Manages primary memory (RAM) allocation and deallocation.
- **File System Management**: Manages files on storage devices.
- **Device Management**: Manages device communication and peripheral devices.
- **Security & Access Control**: Protects data and resources through authentication and permissions.
- **User Interface (UI)**: Provides a command-line interface (CLI) or graphical user interface (GUI) for user interaction.

Types of Operating Systems:

1. **Batch Operating Systems**:
    - Jobs are processed in batches without user interaction.
    - Example: Early IBM systems.
    - **Drawback**: No real-time interaction, as tasks are queued.
2. **Time-Sharing Operating Systems**:
    - Multiple users share the system simultaneously.
    - The OS uses **time slices** to give each process a short burst of CPU time.
    - Example: UNIX.
    - **Advantage**: Quick response time and efficient utilization of resources.
3. **Distributed Operating Systems**:
    - Manages multiple machines working together as a single system.
    - It allows resource sharing across a network of computers.
    - Example: LOCUS.
4. **Real-Time Operating Systems (RTOS)**:
    - Designed to process data as it comes, with strict timing constraints.
    - **Hard RTOS**: Guarantees tasks are completed within deadlines.
    - **Soft RTOS**: Prioritizes deadlines but may allow some flexibility.
    - Example: Systems controlling industrial robots, flight control systems.

5. **Network Operating Systems (NOS)**:
    - Provides services for networking computers, allowing resource sharing like files and printers.
    - Example: Novell NetWare, Microsoft Windows Server.
6. **Mobile Operating Systems**:
    - Designed for smartphones and tablets.
    - Example: Android, iOS.

---

2. Process Creation and Termination

What is a Process?

A **process** is a program in execution. It includes the program code, current activity (represented by the program counter), and associated resources like open files, registers, and memory.

Process States:

1. **New**: The process is being created.
2. **Ready**: The process is ready to run but is waiting for CPU allocation.
3. **Running**: The process instructions are being executed.
4. **Waiting**: The process is waiting for some event (like I/O).
5. **Terminated**: The process has finished execution.

Process Control Block (PCB):

The **PCB** is a data structure that holds process-related information such as:

- Process ID.
- Program counter.
- Registers.
- Memory limits.
- List of open files.

Process Creation:

A new process is created by the OS when:

- A **new program** is loaded into memory.
- A process **creates a child process** (using `fork()` in UNIX).
- A **user request** initiates a process (e.g., double-clicking an application).

Steps in Process Creation:

1. **Assign a Unique Process ID (PID)**.
2. **Allocate memory** for the process.
3. **Initialize process control block (PCB)**.
4. **Load program code into memory**.
5. **Set up the execution environment** (registers, stack, etc.).
6. **Place the process in the Ready Queue** for CPU scheduling.

Example in UNIX:

In UNIX, a parent process can create a child process using the `fork()` system call.

```
int pid = fork();
if (pid == 0) {
    // Child process
    execv("program");
} else {
    // Parent process
    wait(NULL); // Wait for child to terminate
}
```

Process Termination:

A process can be terminated due to:

- **Normal Completion**: The process finishes its task.
- **Error**: The process encounters an error (e.g., divide by zero).
- **Killed by another process**: The process is forcibly terminated by another process (e.g., `kill` in UNIX).
- **Resource unavailability**: The process can't continue due to a lack of resources (e.g., memory).

Process Termination in UNIX:

```
exit(0); // Normal termination
```

---

3. Pre-emptive Scheduling

Definition:

Pre-emptive scheduling allows the operating system to suspend a running process to allocate the CPU to another process. This is used in time-sharing and real-time systems, ensuring that no single process monopolizes the CPU.

How it Works:

- The CPU can switch from one process to another if the current process's time slice (quantum) expires or a higher-priority process becomes ready to execute.
- Pre-emptive scheduling introduces the concept of context switching, where the CPU's state is saved before switching to another process.

Advantages:

- Improved response time and CPU utilization.
- Suitable for time-sharing and interactive systems.
- Allows for higher-priority processes to be executed without waiting for lower-priority processes to complete.

Disadvantages:

- Higher overhead due to frequent context switching.
- More complex implementation.

Examples of Pre-emptive Scheduling Algorithms:

1. Round Robin (RR):

- **Concept**: Each process gets a fixed time slice (quantum), and processes are executed in a circular order.
- **Pre-emptive**: If a process doesn’t finish within its quantum, it's pre-empted and placed at the end of the ready queue.

Example:

Let’s consider three processes with burst times:

Process	Burst Time
P1	5
P2	9
P3	6

For a time quantum of 4 units, the Gantt chart would look like:

Copy code

```
| P1 | P2 | P3 | P1 | P2 | P3 | P2 |
0    4    8    12   13   17   21   22
```

- **Average Waiting Time (AWT)** = (0 + 4 + 8) / 3 = 4 units.
- **Average Turnaround Time (ATT)** = (13 + 22 + 21) / 3 = 18.67 units.

2. Priority Scheduling (Preemptive):

- **Concept**: Processes are scheduled based on priority, and if a higher-priority process arrives, it preempts the running process.
- **Pre-emptive**: If a process with higher priority arrives, it preempts the current process.

---

4. Non-Preemptive Scheduling

Definition:

In non-preemptive scheduling, once a process is assigned the CPU, it runs to completion or until it voluntarily relinquishes the CPU (e.g., when it requires I/O). It cannot be interrupted by the operating system.

Advantages:

- Simple to implement.
- No context switching overhead during process execution.

Disadvantages:

- Poor response time for short tasks.
- Not ideal for time-sharing systems.

Examples of Non-Preemptive Scheduling Algorithms:

1. First-Come, First-Served (FCFS):

- **Concept**: Processes are executed in the order they arrive. The first process to arrive is executed first.

Example:

Consider three processes:

Process	Burst Time
P1	10
P2	5
P3	8

The execution order would be P1 → P2 → P3.

Gantt Chart:

Copy code

```
| P1 |          | P2 |          | P3 |
0    10         15         23
```

- **Average Waiting Time (AWT)** = (0 + 10 + 15) / 3 = 8.33 units.
- **Average Turnaround Time (ATT)** = (10 + 15 + 23) / 3 = 16 units.

2. Shortest Job First (SJF):

- **Concept**: The process with the shortest burst time is selected for execution.
- **Non-preemptive**: Once a process starts, it runs to completion.

Example:

Consider three processes:

Process	Burst Time
P1	6
P2	8
P3	3

Execution order: P3 → P1 → P2.

Gantt Chart:

Copy code

```
| P3 |    | P1 |          | P2 |
0    3    9          17
```

- **AWT** = (0 + 3 + 9) / 3 = 4 units.
- **ATT** = (3 + 9 + 17) / 3 = 9.67 units.

---

5. Race Conditions

Definition:

A race condition occurs when multiple processes or threads access shared resources simultaneously, and the final outcome depends on the sequence in which they execute. This can lead to inconsistent or unexpected results.

Example:

Consider two processes that update a shared variable `counter`.

Process	Arrival Time	Burst Time	Completion Time
P1	0	4	6
P2	1	3	9
P3	2	2	7

2. **Priority Scheduling**:
    - **Concept**: Each process is assigned a priority. The CPU is assigned to the process with the highest priority. Lower-priority processes are pre-empted by higher-priority ones.
    - **Types**:
        - **Pre-emptive**: If a process with a higher priority arrives, it pre-empts the currently running process.
        - **Non-pre-emptive**: The currently running process is allowed to finish even if a higher-priority process arrives.

    - **Example**: Consider three processes arriving at the same time with priorities of 1, 3, and 2. The OS will preempt a lower-priority process when a higher-priority process arrives.
3. **Shortest Remaining Time First (SRTF)**:
    - **Concept**: The process with the shortest remaining CPU time is executed next. If a new process arrives with a shorter burst time, it pre-empts the currently running process.
    - **Example**: Consider four processes arriving at different times with varying burst times. If a new process with a shorter burst time arrives, it pre-empts the current process.

Key Differences Between Pre-emptive and Non-Pre-emptive Scheduling:

Pre-emptive Scheduling	Non-Pre-emptive Scheduling
The OS can suspend a process at any time.	The process runs until completion.
Responsive to high-priority tasks.	Simpler to implement but less flexible.
Useful in time-sharing systems.	Often used in batch systems.

6. Threads

A **thread** is the smallest unit of execution within a process. It represents a single sequence of instructions that can be executed independently by the CPU. Each thread within a process shares the same resources, such as memory and open files, but operates independently of other threads, making threads a key concept in **concurrent programming**.

6.1. Key Concepts of a Thread

- **Process vs. Thread**:
    - A **process** is a running instance of a program. It contains the program's code, data, and other resources.
    - A **thread** is a lightweight, smaller part of a process that performs a specific task. A process can have multiple threads running concurrently, sharing the same resources like memory and file handles but having their own execution state (like registers, stack, and program counter).
- **Single-threaded vs. Multi-threaded**:
    - A **single-threaded** application has only one thread, which means it performs one task at a time.

- A **multi-threaded** application has multiple threads, allowing it to perform multiple tasks simultaneously or concurrently. Each thread can run different parts of the program in parallel.

6.2. Characteristics of Threads

- **Independent Execution**: Each thread runs independently of other threads in the same process. It has its own execution flow but shares process-level resources (memory, data, etc.) with other threads.
- **Shared Memory**: Since threads belong to the same process, they share the same address space, allowing them to access common variables and data. This is both an advantage (efficient communication) and a challenge (potential for race conditions).
- **Lighter than Processes**: Threads are more lightweight compared to processes. Creating a new thread is faster and uses fewer resources than creating a new process, making multi-threading more efficient than multi-processing for tasks that can be performed in parallel.
- **Context Switching**: When switching between threads, the CPU performs a context switch, which involves saving the state (registers, program counter, etc.) of the current thread and loading the state of the new thread. Since threads share the same process resources, context switching between threads is faster than between processes.

6.3. Why Use Threads?

1. **Concurrency**: Threads allow a program to perform multiple tasks simultaneously. For example, in a web server, one thread can handle a client request while another thread handles a database operation.
2. **Parallelism**: On multi-core CPUs, different threads can be executed on different CPU cores at the same time, leading to true parallelism and faster execution of tasks.
3. **Improved Performance**: By dividing a program into multiple threads, some tasks can run concurrently, improving the responsiveness and performance of the program, especially in I/O-bound tasks (e.g., reading from a file, network requests).
4. **Efficient Resource Sharing**: Since threads in a process share memory and resources, communication between threads is easier and faster compared to processes, where communication typically requires more complex inter-process communication (IPC).

6.4. Components of a Thread

1. **Thread ID**: A unique identifier for each thread.
2. **Program Counter (PC)**: Stores the address of the next instruction to be executed.
3. **Registers**: Store intermediate data used by the thread.
4. **Stack**: Each thread has its own stack that stores local variables, function parameters, and the return address for function calls.
5. **Thread State**: The current state of the thread (running, waiting, terminated, etc.).

6.5. Thread Life Cycle (States)

A thread can be in one of several states during its life cycle:

1. **New**: The thread is created but not yet started.
2. **Runnable**: The thread is ready to run and waiting for CPU time.
3. **Running**: The thread is actively being executed by the CPU.
4. **Blocked/Waiting**: The thread is waiting for an external event (e.g., I/O completion or another thread).
5. **Terminated**: The thread has finished executing.

6.6. Types of Threads

1. **User-Level Threads**: These are managed by the user application, not the operating system. Thread management (creation, scheduling, and termination) is done in user space.
    - **Advantages**: Faster thread management since there's no system call overhead.
    - **Disadvantages**: The OS kernel is unaware of these threads, which can make handling certain system-level operations (like I/O) inefficient.

2. **Kernel-Level Threads**: These are managed by the OS kernel. The kernel is aware of and handles thread creation, scheduling, and termination.
    - **Advantages**: The kernel can optimize thread management, especially for multi-core processors.
    - **Disadvantages**: More overhead than user-level threads, as every thread operation requires interaction with the kernel.

There are several models that describe how **user threads** are mapped to **kernel threads**. These models manage how user-level threads (those created and managed by user programs) are mapped to the underlying kernel-level threads (which the operating system manages) for execution. These models are important in operating systems to manage the concurrency and efficiency of multithreaded applications.

The three primary models are:

1. **Many-to-One Model**
2. **One-to-One Model**
3. **Many-to-Many Model**

Let’s dive into the details of each:

---

i). Many-to-One Model

In the **many-to-one model**, multiple user threads are mapped to a single kernel thread. The thread management (creation, scheduling, synchronization) is done entirely in user space, and the kernel only knows about the single kernel thread on which the user threads are running.

How it Works:

- The process manages multiple user threads but the OS manages only one kernel thread.
- All user threads are mapped to this one kernel thread, meaning they must share the CPU time of the single kernel thread.

Advantages:

- **Fast thread management**: Since thread creation and context switching happen in user space, they are very efficient and don't require kernel intervention.
- **No kernel-level overhead**: Since the kernel only needs to manage one thread, there is minimal overhead.

Disadvantages:

- **No true parallelism**: Even on multiprocessor systems, user threads can only run one at a time because there is only one kernel thread. This means there's no true concurrency or parallel execution.
- **Blocking**: If one user thread performs a blocking operation (e.g., I/O), the entire process is blocked because the kernel-level thread blocks, stopping all user threads associated with it.

Example Systems:

- **Green Threads in Java** (early versions): A Java implementation of user threads, where multiple user threads were managed by one kernel thread.
- **GNU Portable Threads (Pth)**: A user-level thread library for UNIX-based systems.

---

ii). One-to-One Model

In the **one-to-one model**, each user thread is mapped to a unique kernel thread. For every user thread that is created, a corresponding kernel thread is also created.

How it Works:

- Each user thread has a corresponding kernel thread, allowing multiple threads to run in parallel on multiprocessor systems.

- Thread management (e.g., creation, termination, and context switching) requires system calls to the kernel, as the OS is aware of each thread.

Advantages:

- **True parallelism**: Multiple threads can run simultaneously on different processors or cores, leveraging multi-core systems effectively.
- **Blocking**: If one thread blocks, other threads can continue executing because they each have their own kernel thread.

Disadvantages:

- **High overhead**: Since each user thread requires a corresponding kernel thread, creating and managing many threads can lead to significant overhead. Each thread needs its own kernel resources (memory, registers, etc.).
- **System limits**: The number of threads in the system is limited by the OS's ability to handle kernel threads. Creating thousands of threads may exceed system limits.

Example Systems:

- **Windows NT/XP/7**: Each user thread is mapped to a kernel thread.
- **Linux**: Modern Linux threading libraries such as POSIX threads (pthreads) also use this model.

---

iii). Many-to-Many Model

The **many-to-many model** is a hybrid approach, allowing multiple user threads to be mapped to a smaller or equal number of kernel threads. This model allows the OS to create a flexible balance between user-level thread management and kernel-level parallelism.

How it Works:

- The OS can map **many** user threads to a **smaller number of kernel threads**.
- The number of kernel threads is configurable based on system resources.
- User threads are scheduled onto the available kernel threads. The mapping can be dynamic, meaning the system can create more kernel threads if needed or reduce the number to optimize performance.

Advantages:

- **Flexibility**: This model offers the benefits of both the many-to-one and one-to-one models. It can handle multiple user threads without the overhead of a one-to-one model, while still allowing true parallel execution on multi-core systems.

- **Efficient resource usage**: It avoids the overhead of creating too many kernel threads while allowing for some degree of concurrency.

Disadvantages:

- **Complex implementation**: The mapping between user and kernel threads is more complex and requires advanced scheduling techniques.

Example Systems:

- **Solaris** (prior to Solaris 9): Used a many-to-many threading model.
- Some modern UNIX implementations support many-to-many through **pthreads** (POSIX threads), which can be configured to use a similar model.

7. Race Conditions in Operating Systems

A **race condition** occurs in a concurrent system (such as an operating system) when the **behavior or outcome of a process depends on the timing or sequence of uncontrollable events, such as the execution order of threads or processes**. Essentially, two or more processes or threads are racing to access shared resources (like memory, files, or variables), and the final result can differ depending on which thread wins the race to execute a particular part of the code.

Why Do Race Conditions Occur?

Race conditions occur when:

1. **Multiple processes or threads** are executing simultaneously in a system.
2. These processes or threads **share common resources** such as memory or files.
3. The processes or threads **access or modify these resources without proper synchronization mechanisms**, leading to unpredictable behavior based on timing.

For example, in the banking example, both threads access and modify the same resource (the bank balance) without coordinating with each other.

Critical Section Problem

A race condition usually involves a **critical section**, a part of the code where shared resources (such as variables, data structures, or files) are accessed. The goal of process synchronization is to ensure that only one thread or process is executing the critical section at any given time, preventing race conditions.

Critical Section Requirements:

1. **Mutual Exclusion**: Only one process should be allowed to execute the critical section at a time.
2. **Progress**: If no process is executing the critical section and a process wants to enter, it should be allowed to enter.
3. **Bounded Waiting**: No process should wait indefinitely to enter the critical section.

---

8. Process Synchronization Techniques

To avoid race conditions, we need to implement proper synchronization mechanisms that control the access to shared resources. The goal of process synchronization is to ensure that **multiple processes or threads can operate concurrently without causing inconsistencies or errors** due to shared resource access.

1. Locks and Mutexes

- **Locks** (or **Mutexes** — short for mutual exclusion) are the simplest form of synchronization. They are used to enforce **mutual exclusion** on a critical section.
- When a thread or process wants to access a shared resource, it must first acquire the lock. If another thread holds the lock, the requesting thread must wait until the lock is released.

2. Semaphores

- A **semaphore** is a more general synchronization tool. It uses two operations: **wait (P)** and **signal (V)** to control access to a shared resource.
- Semaphores can be used for mutual exclusion (binary semaphores) or to control access to a limited number of resources (counting semaphores).
- A **binary semaphore** can only have two values, 0 and 1, and works similarly to a mutex.
- A **counting semaphore** can have any integer value and is used to allow multiple processes to access the resource simultaneously, up to a defined limit.

Here's a breakdown of when **UP (signal)** and **DOWN (wait)** semaphores get triggered:

---

1. DOWN Operation (P operation / wait / acquire)

The **DOWN** operation is triggered when a process or thread wants to access a shared resource. It decrements the semaphore value and checks if the resource is available:

- **When DOWN is triggered**:

    - **Semaphore Value > 0**: If the semaphore value is greater than 0, it means that the resource is available. The thread or process decrements the semaphore and proceeds to access the resource.
    - **Semaphore Value == 0**: If the semaphore value is 0 or less, it means that no resources are currently available, and the thread or process is forced to wait until another process releases the resource (via the UP operation). The process is **blocked** until it can successfully decrement the semaphore.

Use Case of DOWN:

- A **DOWN operation** is triggered when a thread wants exclusive access to a critical section (for mutual exclusion) or shared resource. It ensures that the number of threads accessing the shared resource never exceeds the number of available units.

---

2. UP Operation (V operation / signal / release)

The **UP** operation is triggered when a process or thread releases a shared resource. It increments the semaphore value, signaling that the resource is now available for other threads or processes waiting on it.

- **When UP is triggered**:
    - A thread or process calls the **UP operation** after it has finished using the resource. This increments the semaphore value by 1.
    - If there are any processes or threads waiting (i.e., blocked due to the semaphore value being 0), one of them will be allowed to proceed.
    - If no threads are waiting, the semaphore simply increments, allowing future threads to access the resource without blocking.

Use Case of UP:

- The **UP operation** is triggered when a thread has finished using the resource and wants to make it available to other threads. It increments the semaphore and wakes up any waiting threads.

Types of Semaphores and How They Use UP and DOWN:

1. **Binary Semaphores (Mutex)**:
    - Can have values of 0 or 1, representing a single shared resource (like a mutex).
    - **DOWN** operation is triggered when a thread tries to acquire the lock, and if the lock is unavailable (value 0), the thread blocks.
    - **UP** operation is triggered when a thread releases the lock, signaling to other threads that the lock is available.

```
DOWN(Semaphore s){
    if(s.value == 1){
        s.value = 0;
    }
    else{
        // Block this process & place in suspended list
        // slup();
    }
}
```

```
UP(Semaphore s){
    if(/* suspended list is empty */){
        s.value = 1;
    }
    else{
        // Select a process from suspended list
        // and wake up();
    }
}
```

2. **Counting Semaphores**:
    - Can have values greater than 1, representing multiple instances of a resource (e.g., a pool of network connections).
    - **DOWN** operation is triggered when a thread wants to acquire one of the available resources.
    - **UP** operation is triggered when a thread returns a resource, signaling availability.

```
Down(Semaphore s){
    s.value--;
    if(s.value <= 0){
        // Put process in suspended list
    }
    else
        return;
}
```

```
UP(Semaphore s){
    s.value++;
    if(s.value <= 0){
        // Select a process from suspended list
        // and wake up();
    }
    else
        return;
}
```

9. Deadlocks

A **deadlock** occurs in an operating system when a set of processes is **stuck** because each process is waiting for a resource that is held by another process in the same set. None of the processes can continue because they are all waiting for each other, resulting in a system deadlock.

9.1. Deadlock Basics

Definition:

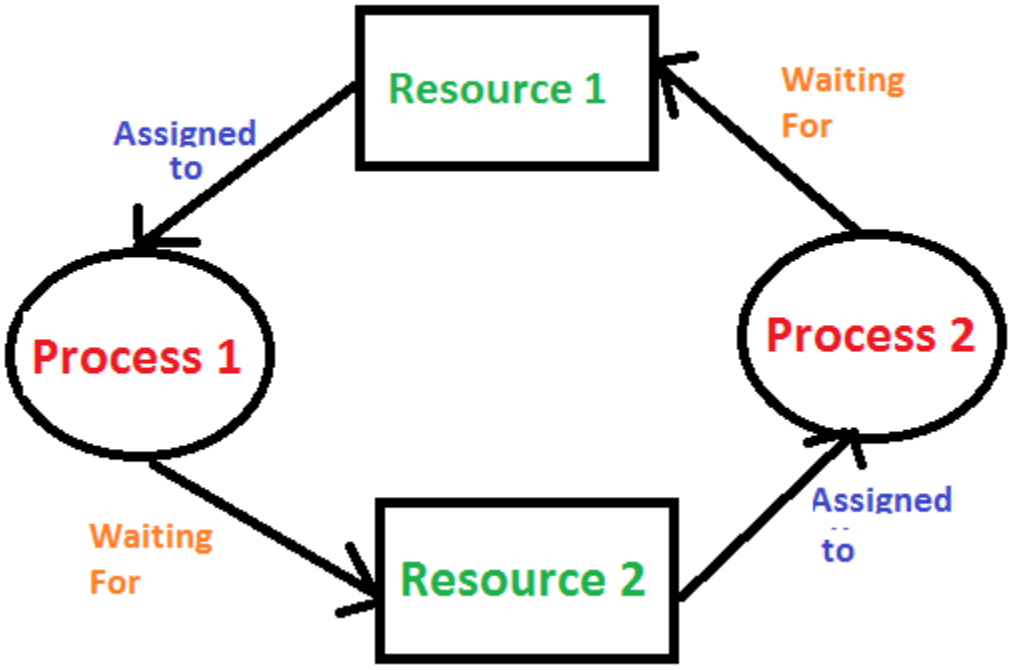
A **deadlock** is a situation where two or more processes are unable to proceed because each process is waiting for the other to release a resource. It is a circular chain of dependencies among the processes.

Example:

Consider two processes, P1 and P2, and two resources, R1 and R2.

- P1 holds R1 and waits for R2.
- P2 holds R2 and waits for R1.

In this case, neither P1 nor P2 can proceed, causing a **deadlock**.



9.2. Necessary Conditions for Deadlock

For a deadlock to occur, the following **four conditions** must be true **simultaneously**:

1. **Mutual Exclusion**:
    - At least one resource must be held in a **non-sharable mode** (i.e., only one process can use the resource at a time).
2. **Hold and Wait**:
    - A process is holding at least one resource and is waiting for additional resources held by other processes.
3. **No Preemption**:
    - Resources cannot be forcibly taken away from a process; they must be released voluntarily by the process holding them.
4. **Circular Wait**:

    - A set of processes are waiting for each other in a circular chain, where each process holds a resource that the next process in the chain is waiting for.

9.3. Deadlock Representation

The deadlock situation can be represented by a **Resource Allocation Graph (RAG)**. In this graph:

- **Processes** are represented by circles.
- **Resources** are represented by squares.
- An **edge from a process to a resource** indicates that the process is requesting that resource.
- An **edge from a resource to a process** indicates that the resource is allocated to the process.

A **cycle** in the graph indicates a **deadlock**.

9.4. Methods for Handling Deadlock

There are several strategies for handling deadlocks in an operating system:

9.4.1. Deadlock Prevention

Deadlock prevention ensures that **one of the necessary conditions** for deadlock does not occur. This can be done by altering system behavior.

1. Break Mutual Exclusion:

- Use **spooling** or **shareable resources** so that processes can share resources without causing deadlock.
- Example: Printer spooling allows multiple processes to send print jobs to a shared buffer (spool) instead of directly holding the printer resource.

2. Remove Hold and Wait:

- Ensure that a process **requests all its required resources at once** and holds none while waiting. This can be inefficient due to resource underutilization.

- Example: A process waits for both the printer and scanner at the start rather than holding one and waiting for the other.

3. Allow Preemption:

- If a process holding some resources requests additional resources that are unavailable, **preempt the resources** it currently holds.
- Example: Preempt CPU time from a lower-priority process when a higher-priority process needs CPU.

4. Avoid Circular Wait:

- Impose a **total ordering** on the resources. Each process can only request resources in a predefined order.
- Example: If processes must acquire resources in the order `R1`, `R2`, `R3`, then deadlocks due to circular waiting are prevented.

9.4.2. Deadlock Avoidance

In **deadlock avoidance**, the system ensures that it **never enters a deadlock state** by checking resource allocation at runtime. This is done using algorithms that consider future resource needs and only grant resources if they won't lead to deadlock.

1. Banker's Algorithm:

- The most famous deadlock avoidance algorithm, the **Banker's Algorithm**, works by simulating resource allocation to see if it leads to a safe state. The system only grants a resource request if it leaves the system in a **safe state**.
- **Safe State**: A state in which there exists a sequence of processes that can finish without leading to deadlock.
- **Example**:
    - Let's say there are `3` processes and `5` units of a resource. The Banker's Algorithm ensures that even if a process requests more units, there will still be enough for the others to finish and avoid deadlock.

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

9.4.3. Deadlock Detection and Recovery

In this strategy, the system allows deadlocks to occur but provides mechanisms to **detect** and **recover** from them.

1. Detection:

- The system continuously monitors the state of resource allocation and looks for cycles in the **resource allocation graph** (RAG). If a cycle is detected, a deadlock has occurred.

2. Recovery:

- **Terminate Processes**: Abort one or more processes to break the cycle and release resources.
    - **Example**: Abort the process that is causing the deadlock or the one with the least priority.
- **Preempt Resources**: Preempt resources from one or more deadlocked processes and give them to other processes.
    - **Example**: Reallocate resources from a deadlocked process and restart it later.

---

5. Real-World Examples

Example 1: Printer and File Deadlock

- Two processes need to use a printer and a file in a specific order.
    - **Process 1** holds the printer and waits for the file.
    - **Process 2** holds the file and waits for the printer.

- Both are deadlocked, as each waits for the other to release a resource.

Example 2: Database Deadlock

- Consider a banking system where **Transaction A** locks **Account X** and **Transaction B** locks **Account Y**.
    - **Transaction A** needs **Account Y** and **Transaction B** needs **Account X**.
- A deadlock occurs as both transactions wait indefinitely for each other to release the required account.

10. Memory Management in Operating Systems (OS)

Memory management in an operating system is the process of managing the computer's primary memory (RAM). It involves handling the allocation and deallocation of memory blocks to various programs and processes running on the system. Proper memory management ensures efficient use of memory, better performance, and system stability.

CPU Utilization Formula

The overall **CPU utilization** in a system can be modeled as a function of the **probability that a process is waiting for I/O**.

CPU Utilization Formula:

$$\textit{CPU Utilization} = 1 - P^n$$

Where:

- **P** = fraction of time a process spends waiting for I/O (i.e., the probability that the CPU is idle during a process).
- **n** = number of processes (degree of multiprogramming).

Explanation:

- When only one process is running, the CPU is idle **P** fraction of the time, and the CPU utilization is **1 - P**.
- When **n** processes are running, the CPU is only idle if **all processes** are waiting for I/O, which happens with probability **P^n**. Thus, the CPU utilization increases as **n** (degree of multiprogramming) increases.

Key Responsibilities of Memory Management

1. **Allocation of Memory**:
    - Allocate memory to programs when they request it.
    - Ensure that memory is properly deallocated when it's no longer needed.
2. **Tracking Memory**:
    - Keep track of each memory location, whether it is free or allocated, and how much memory is available.
3. **Swapping and Paging**:
    - Manage the movement of processes between main memory and disk when there is insufficient memory.
4. **Protection and Isolation**:
    - Prevent processes from accessing memory allocated to other processes, ensuring data security and integrity.

10.1. Memory Allocation Techniques

10.1.1. Contiguous Memory Allocation

In contiguous memory allocation, each process is allocated a single contiguous block of memory. This method is simple but can lead to **fragmentation**.

Types of Contiguous Allocation:

1. Fixed Partitioning

Definition:

- In **fixed partitioning**, memory is divided into fixed-sized partitions.
- Each partition can contain exactly one process, and the partition size is determined at system startup.

Key Points:

- **Internal Fragmentation**: If a process is smaller than the partition size, the leftover space within the partition is wasted, leading to **internal fragmentation**.
- **Simple Implementation**: Easy to implement, but not flexible since partition sizes are fixed.

Example:

- Assume 100 MB memory is divided into five fixed partitions of 20 MB each. If a process requires only 10 MB, 10 MB of memory in that partition is wasted (internal fragmentation).

---

2. Variable-Size Partitioning

Definition:

- In **variable-size partitioning**, memory is divided into partitions of varying sizes based on the needs of processes.
- The OS allocates exactly as much memory as the process requires.

Key Points:

- **External Fragmentation**: Over time, small holes of free memory can form between allocated blocks, leading to **external fragmentation**.
- **Memory Compaction**: One way to handle external fragmentation is to perform **memory compaction**, where free memory blocks are merged together.

Example:

- Suppose you have 100 MB of memory, and three processes request 20 MB, 30 MB, and 40 MB respectively. Memory is divided dynamically into these sizes, but gaps of free memory can appear between processes over time.

Algorithms for Variable Partitioning in Memory Management

In **variable partitioning**, memory is dynamically divided into **variable-sized blocks** to accommodate processes based on their specific memory requirements. Unlike fixed partitioning, where partitions have predefined sizes, variable partitioning allocates exactly the required amount of memory to each process. Over time, this leads to **fragmentation** (both internal and external), so algorithms are needed to manage memory allocation efficiently.

1. First Fit Algorithm

Description:

- **First Fit** allocates the first block of memory that is **large enough** to satisfy the process's request.

- It **searches memory from the beginning** and assigns the process to the first suitable free block it finds.

Steps:

1. Traverse the memory list from the start.
2. Allocate the process to the **first block** that has enough free space.
3. If the block is larger than the process's needs, **split** the block and leave the remaining space as a new free block.

Advantages:

- **Simple** to implement.
- Faster allocation since it stops searching once a suitable block is found.

Disadvantages:

- Can cause **fragmentation** near the beginning of memory, as small gaps of unusable free space might be left over.

Example:

Consider memory blocks of sizes: **100 KB, 500 KB, 200 KB, 300 KB, 600 KB**.

- A process requiring **212 KB** arrives.
- **First Fit** will allocate the process to the **500 KB block** (the first block that can accommodate it). The remaining **288 KB** will be left as a free block.

---

2. Best Fit Algorithm

Description:

- **Best Fit** allocates the **smallest free block** that is large enough to satisfy the process's request.
- It **searches the entire memory list** to find the best-fitting block (i.e., the block with the least leftover space after allocation).

Steps:

1. Traverse the memory list and find the **smallest block** that can hold the process.
2. Allocate the process to that block.
3. If there's leftover space, create a new free block.

Advantages:

- **Minimizes leftover space**: By finding the smallest block, it tries to use memory more efficiently.

Disadvantages:

- **Slower**: The entire list must be searched to find the best fit.
- Can lead to **external fragmentation** as smaller leftover spaces are scattered across memory, making it difficult to allocate larger processes later.

Example:

Consider memory blocks of sizes: **100 KB, 500 KB, 200 KB, 300 KB, 600 KB**.

- A process requiring **212 KB** arrives.
- **Best Fit** will allocate the process to the **300 KB block** (the smallest block that can fit the process). The remaining **88 KB** will be left as a free block.

---

3. Worst Fit Algorithm

Description:

- **Worst Fit** allocates the **largest free block** available.
- It searches for the block that will leave the most leftover space after allocation, attempting to **minimize fragmentation** by keeping larger chunks of free space intact.

Steps:

1. Traverse the memory list and find the **largest block** that can hold the process.
2. Allocate the process to that block.
3. If there's leftover space, create a new free block.

Advantages:

- **Reduces small leftover spaces**, potentially preventing external fragmentation by leaving larger blocks available for future allocations.

Disadvantages:

- **Inefficient use of memory**: Since the largest block is chosen, smaller processes may unnecessarily occupy large blocks, leaving smaller ones unusable.

- Can still lead to **fragmentation**.

Example:

Consider memory blocks of sizes: **100 KB, 500 KB, 200 KB, 300 KB, 600 KB**.

- A process requiring **212 KB** arrives.
- **Worst Fit** will allocate the process to the **600 KB block** (the largest block). The remaining **388 KB** will be left as a free block.

---

4. Next Fit Algorithm

Description:

- **Next Fit** is a variation of **First Fit**, but instead of starting the search from the beginning of memory every time, it starts from where the **previous allocation** was made.
- This improves performance by not searching through blocks that have already been processed.

Steps:

1. Start the search from the last allocated block.
2. Traverse memory **circularly** until a suitable free block is found.
3. Allocate the process to that block.
4. Move the pointer to the next block for future allocations.

Advantages:

- **Faster** than First Fit, as it avoids re-scanning blocks that were already processed.
- **Simpler** to implement compared to Best Fit and Worst Fit.

Disadvantages:

- Can still cause **fragmentation**, as it may skip over smaller free spaces that could otherwise be used.

Example:

Consider memory blocks of sizes: **100 KB, 500 KB, 200 KB, 300 KB, 600 KB**.

- If the previous allocation was made at the **300 KB block** and a new process requires **212 KB**, the **Next Fit** algorithm will start scanning from the next block (600 KB) and allocate the process there, leaving **388 KB** free.

---

5. Buddy System

Description:

- The **Buddy System** is a memory allocation algorithm that divides memory into blocks that are **powers of two**. It dynamically splits or coalesces memory blocks to satisfy allocation requests.

Steps:

1. Initially, the entire memory is treated as a single large block (of size 2^k).
2. When a process requests memory, the system splits blocks into halves (buddies) until it finds the smallest power-of-two block that can accommodate the process.
3. When a block is freed, it is **merged with its buddy** (if the buddy is also free) to form a larger block.

Advantages:

- **Efficient allocation and deallocation**: Easy to split and merge memory blocks.
- **Reduces external fragmentation** by keeping blocks of memory powers of two.

Disadvantages:

- Can lead to **internal fragmentation**, as the allocated block may be slightly larger than the process needs.

Example:

- Suppose a system has **1024 KB** of memory. If a process requests **130 KB**, the Buddy System will allocate a **256 KB** block (next power of two), and leave **768 KB** available for future requests.

10.1.2. Non-Contiguous Memory Allocation

In non-contiguous memory allocation, processes are allowed to occupy multiple memory blocks, which do not have to be adjacent. This helps to minimize fragmentation and improve memory utilization.

Key Techniques:

10.1.2.1. Paging

Definition:

- **Paging** is a memory management scheme that eliminates the need for contiguous allocation of physical memory. It divides both **physical** and **logical memory** into fixed-sized blocks called **pages** and **frames**, respectively.
- The size of the pages and frames is the same, and a **page table** maps logical pages to physical frames.

Key Points:

- **No External Fragmentation**: Paging avoids external fragmentation because pages can be loaded into any available frame.
- **Internal Fragmentation**: Some internal fragmentation can still occur if a process does not fully utilize the last page.

Example:

- A process of size 8 KB is divided into four 2 KB pages. If physical memory has frames of 2 KB, each page is mapped into an available frame via a page table.

10.1.2.2. Segmentation

Definition:

- **Segmentation** is a memory management technique where the memory is divided into different **segments** of varying sizes, corresponding to different logical divisions of a process (e.g., code, data, stack).
- Each segment is mapped into a physical memory block, and a **segment table** keeps track of segment locations.

Key Points:

- **Logical Division**: Unlike paging, segmentation reflects the logical divisions of a process (e.g., code and data segments).

- **External Fragmentation**: Segmentation can suffer from external fragmentation due to varying segment sizes.

Example:

- A process has a **code segment** of 4 KB and a **data segment** of 6 KB. These are mapped separately to physical memory using a segment table.

10.1.2.3. Inverted Paging

Definition:

- In **inverted paging**, there is a **single page table** for the entire system, rather than one page table per process. The table tracks which page of which process is stored in each frame.

Key Points:

- **Memory Efficiency**: Inverted paging reduces the amount of memory needed for page tables.
- **Hashing**: Typically, a **hashing function** is used to map logical addresses to the frame number.

Example:

- Instead of having separate page tables for each process, the OS maintains one large page table where each entry contains information about which process owns a particular frame.

10.1.2.4. Thrashing

Definition:

- **Thrashing** occurs when a system spends more time **swapping pages in and out of memory** than executing processes, resulting in a significant decline in performance.

Key Points:

- **Caused by Overloading Memory**: Thrashing occurs when processes do not have enough pages to run efficiently, leading to frequent page faults.
- **Working Set Model**: To reduce thrashing, the **working set model** can be used, which keeps track of the set of pages that a process frequently accesses.

Example:

- A process continuously accesses more pages than can fit in memory, causing the OS to repeatedly swap pages in and out, leading to thrashing.

---

7. Page Faults

Definition:

- A **page fault** occurs when a process tries to access a page that is not currently in memory.
- The OS must fetch the page from disk and load it into memory.

Key Points:

- **Minor Page Fault**: Occurs when the page is not in the current working set but still in memory (e.g., swapped out).
- **Major Page Fault**: Occurs when the page is not in memory and must be loaded from disk, which is much slower.

Example:

- A process tries to access a page that is currently not in memory. The OS triggers a page fault, loads the page from disk, and updates the page table.

---

8. Page Replacement Algorithms

When memory is full, the OS must **replace** a page in memory to load a new one. **Page replacement algorithms** decide which page to remove.

1. First-In, First-Out (FIFO)

- **Description**: The oldest page in memory is replaced first.
- **Advantage**: Simple to implement.

- **Disadvantage**: Can lead to poor performance if the oldest page is frequently used.
- **Example**:
    - If pages `1, 2, 3, 4` are in memory and `5` needs to be loaded, page `1` (the oldest) is replaced.

2. Least Recently Used (LRU)

- **Description**: Replaces the page that has not been used for the longest period of time.
- **Advantage**: More efficient than FIFO in many cases.
- **Disadvantage**: Requires keeping track of access times, which can be complex.
- **Example**:
    - If pages `1, 2, 3, 4` are in memory and `5` needs to be loaded, the page that hasn't been accessed the longest (e.g., `2`) is replaced.

3. Optimal Page Replacement

- **Description**: Replaces the page that will not be used for the longest time in the future.
- **Advantage**: The best possible performance.
- **Disadvantage**: Impossible to implement in real-time (requires future knowledge).
- **Example**:
    - If pages `1, 2, 3, 4` are in memory and `5` needs to be loaded, the page that won't be used soon is replaced.

4. Clock (Second-Chance) Algorithm

- **Description**: A circular list of pages with a reference bit. If the bit is set, the page is given a second chance; otherwise, it is replaced.
- **Advantage**: Efficient and simple to implement.
- **Disadvantage**: May not always be the best performer.
- **Example**:
    - The OS scans pages and gives a second chance to pages that have been used recently, skipping them for replacement.

10.2. Virtual Memory

Virtual memory is a technique that allows processes to use more memory than what is physically available in the system by **swapping** parts of the program between the main memory and the disk. Virtual memory is essential for implementing large programs and multitasking on systems with limited physical memory.

Key Concepts of Virtual Memory:

1. **Paging**:
    - Virtual memory is divided into **pages**, and each page is mapped to a **page frame** in physical memory. If a page is not present in memory, a **page fault** occurs, and the required page is loaded from the disk (swap space).
    - **Example**:
        - A program may require 4 GB of memory, but the system only has 2 GB of RAM. Virtual memory allows the program to execute by swapping pages in and out of the physical memory.
2. **Demand Paging**:
    - Pages are only loaded into memory when they are needed (on-demand). This reduces the amount of memory used by loading only the required parts of a program.
3. **Page Replacement Algorithms**:
    - When the memory is full, and a new page needs to be loaded, the OS must decide which page to replace. Common page replacement algorithms include:
        - **FIFO (First In, First Out)**: The oldest page is replaced.
        - **LRU (Least Recently Used)**: The least recently accessed page is replaced.
        - **Optimal**: The page that will not be used for the longest time is replaced (theoretical).
    - **Example**:
        - In LRU, if a process has used pages 2, 3, and 4 in the last few operations, and page 1 has not been used for a while, page 1 would be replaced if a new page needs to be loaded.

---

3. Fragmentation

3.1. Internal Fragmentation:

Occurs when fixed-sized memory blocks (such as pages) are allocated, but the process does not use all the allocated space. The unused space within the allocated memory block is wasted.

- **Example**:
    - A process requires 15 KB, but it is allocated a 20 KB partition. The remaining 5 KB are wasted, resulting in internal fragmentation.

3.2. External Fragmentation:

Occurs when free memory is scattered in small, non-contiguous blocks, making it impossible to allocate large processes even if there is enough total free memory.

- **Example**:
    - If memory has free blocks of 5 KB, 10 KB, and 15 KB, and a process requires 25 KB, it cannot be allocated memory, even though there is a total of 30 KB available.

Solution: Compaction:

- One way to solve external fragmentation is through **compaction**, where memory is reallocated to move all free blocks together.

---

4. Swapping

Swapping is the process of moving processes between the main memory and secondary storage (usually the hard drive) when the memory is full. Swapping is used to free up memory for higher-priority processes or for processes that are ready to execute.

- **Example**:
    - If two processes are running, and the system needs to load a third process, it may swap one of the currently running processes to the disk and load the new process into memory.

---

5. Thrashing

Thrashing occurs when a system spends more time **swapping pages in and out of memory** than executing actual processes. This happens when the system is overloaded, and there is not enough memory to support all running processes, leading to constant page faults.

- **Example**:

    - If many processes require more memory than is physically available, the system keeps swapping pages in and out, leading to a significant slowdown.

Solution:

- **Increase physical memory** (RAM).
- **Reduce the degree of multiprogramming** by limiting the number of processes in memory.
- Use effective **page replacement algorithms**.

---

6. Memory Protection

Memory protection ensures that a process cannot access memory allocated to another process, protecting the data and code of each process. This is crucial for system security and stability.

Techniques for Memory Protection:

1. **Base and Limit Registers**:
    - The **base register** holds the starting address of the process in memory, and the **limit register** holds the size of the process. This ensures that the process can only access memory within its address space.
2. **Segmentation**:
    - Segments of memory are given permissions (read, write, execute) that dictate what actions can be performed on the segment.

---

7. Examples of Memory Management Techniques

Example 1: Paging in Virtual Memory

- A program requests memory for three pages: Page A, Page B, and Page C.
- Page A is loaded into Frame 1, Page B into Frame 2, and Page C into Frame 3.
- If Page D is needed and the memory is full, one of the existing pages (e.g., Page A) is swapped out using a page replacement algorithm (such as LRU).

Example 2: Segmentation

- A process is divided into three segments: a **code segment**, a **data segment**, and a **stack segment**.

- The OS assigns each segment to a different location in memory.
- If the process needs more memory for its stack, only the stack segment needs to be reallocated, not the entire process.