Placement Preparation Booklet

Get Ready for Your Technical Interview

Author: Pushpendra Kumar Pateriya

Institute: Lovely Professional University

Date: January 10, 2024

Version: 1

Bio: Workbook

LOVELY PROFESSIONAL UNIVERSITY

Success won't come to us unless we go to it.

Contents

Chapter 1 Introduction to Operating Systems

1.1 Operating System (OS)

Definition: The software that manages computer hardware and provides services for computer programs. It acts as an intermediary between the hardware and the user/application software.

1.1.1 Types of Operating Systems:

- **Batch Operating System:** A type of operating system where tasks or jobs are grouped together and processed in batches without user interaction. It is efficient for repetitive tasks and reduces idle time.
- **Time-Sharing Operating System:** An operating system that allows multiple users to share a computer simultaneously. Each user receives a small time slice or quantum to interact with the system, providing the illusion of concurrent execution.
- **Multiprogramming Operating System:** An operating system that allows multiple programs to be loaded into memory at the same time. The CPU is switched between programs, increasing overall system utilization.
- **Multitasking Operating System:** An operating system that allows multiple tasks or processes to run concurrently, providing the appearance of simultaneous execution. Each task receives a time slice to execute.
- **Real-Time Operating System (RTOS):** An operating system designed for applications with specific timing requirements. It guarantees a response within a predetermined time frame, critical for systems such as embedded devices and control systems.
- **Distributed Operating System:** An operating system that runs on multiple interconnected computers and enables them to work together as a single system. It provides transparency, communication, and resource sharing in a networked environment.
- **Network Operating System (NOS):** An operating system designed to support networked computing. It includes features for file sharing, printer sharing, and communication between computers in a network.
- **Embedded Operating System:** An operating system designed to run on embedded systems, which are specialized computing devices with a dedicated function. Examples include operating systems for smartphones, IoT devices, and industrial machines.
- **Mobile Operating System:** An operating system specifically designed for mobile devices such as smartphones and tablets. It provides features like touch input, application management, and wireless communication.
- **Single-User Operating System:** An operating system designed for a single user at a time. It is common in personal computers and provides a straightforward interface for individual tasks.
- **Multi-User Operating System:** An operating system that allows multiple users to access the computer system concurrently. It provides features for user authentication, resource sharing, and access control.

1.2 Kernel

Definition: The core component of an operating system that provides essential services, such as process scheduling, memory management, and device drivers. It directly interacts with the hardware.

1.2.1 Types of Kernels:

- **Monolithic Kernel:** A type of kernel that incorporates all essential operating system functions and services into a single, large executable program. It operates in a single address space and has high performance but can be less modular.

- **Microkernel:** A kernel architecture that keeps the core functions minimal, moving non-essential functions, such as device drivers and file systems, to user space as separate processes. It enhances modularity but may incur higher communication overhead.
- **Hybrid Kernel:** A kernel that combines elements of both monolithic and microkernel designs. It includes a small, efficient kernel core and places some traditional kernel functions in user space. This design aims to achieve a balance between performance and modularity.
- **Exokernel:** A kernel design that exposes the underlying hardware resources directly to applications, allowing them to manage resources more efficiently. It offers fine-grained control over resources but places a greater burden on application developers.
- **Nanokernel:** An extremely lightweight kernel that handles only the most basic functions, such as process scheduling and inter-process communication. It is designed for resource-constrained systems and focuses on minimalism.
- **Real-Time Kernel:** A kernel optimized for real-time systems, ensuring predictable and timely response to external events. It prioritizes tasks based on their deadlines and is commonly used in applications like embedded systems and robotics.
- **Hypervisor (Virtualization Kernel):** A kernel designed for virtualization that allows multiple operating systems to run on the same physical hardware simultaneously. It manages virtual machines, allocating resources and providing isolation between them.
- **Hurd Microkernel:** The microkernel used in the GNU Hurd operating system. It follows a microkernel architecture, with core services running in user space and communication occurring through inter-process communication (IPC).
- **Windows NT Kernel:** The kernel at the core of the Windows NT family of operating systems. It is a hybrid kernel that combines elements of monolithic and microkernel designs, providing features like preemptive multitasking, security, and hardware abstraction.
- **Linux Kernel:**
  The core of the Linux operating system. It follows a monolithic architecture and provides essential services, including process management, memory management, and device drivers. It is part of the broader Linux operating system.

1.2.2 Mode Bit

The mode bit in an operating system is a fundamental mechanism for managing privilege levels, enforcing security, protecting critical resources, and isolating user processes from the core functions of the operating system. It contributes to the overall stability, security, and reliability of the operating system by controlling the level of access that processes have to system resources.

1.3 System Calls

System calls are functions or routines provided by the operating system that allow user-level processes or programs to request services or functionality from the operating system kernel. These calls act as an interface between the user-level application and the lower-level operating system, enabling the application to perform operations that require higher privileges or direct access to system resources.

Types of System Calls:

1. Process Control System Calls:
   (a). `fork()`: Creates a new process by duplicating the calling process.

   (b). `exec()`: Replaces the current process's memory image with a new one.
   (c). `wait()`: Causes a process to wait until one of its child processes exits.
2. File Management System Calls:
   (a). `open()`: Opens a file or device, creating a file descriptor.
   (b). `read()`: Reads data from a file descriptor into a buffer.
   (c). `write()`: Writes data from a buffer to a file descriptor.
   (d). `close()`: Closes a file descriptor.
3. Device Management System Calls:
   (a). `ioctl()`: Performs I/O control operations on devices.
   (b). `read()`: Reads data from a device.
   (c). `write()`: Writes data to a device.
4. Information Maintenance System Calls:
   (a). `getpid()`: Returns the process ID of the calling process.
   (b). `getuid()`: Returns the user ID of the calling process.
   (c). `time()`: Returns the current time.
5. Communication System Calls:
   (a). `pipe()`: Creates a communication pipe between two processes.
   (b). `msgget()`, `msgsnd()`, `msgrcv()`: Create, send, and receive messages using message queues.
   (c). `semget()`, `semop()`, `semctl()`: Create, perform operations on, and control semaphore sets.
6. Memory Management System Calls:
   (a). `brk()`: Changes the end of the data (heap) segment of a process.
   (b). `mmap()`: Maps files or devices into memory.
   (c). `munmap()`: Unmaps files or devices from memory.
7. Network Communication System Calls:
   (a). `socket()`: Creates a new communication endpoint (socket).
   (b). `bind()`, `listen()`, `accept()`: Set up a server socket for network communication.
   (c). `connect()`: Establishes a connection to a remote socket.

1.4 Process

Definition: A program in execution; it represents the basic unit of work in an operating system. Each process has its own memory space and resources.

1.4.1 Types of Processes:

- **Foreground Process:** A process that actively interacts with the user and requires user input. It typically runs in the foreground, and the user interface may be blocked until the process completes.
- **Background Process:** A process that runs independently of the user interface and does not require immediate user interaction. Background processes often execute concurrently with foreground processes.
- **Parent Process:** A process that creates one or more child processes. The parent process typically initiates the execution of child processes and may communicate with them during their execution.
- **Child Process:** A process created by another process, known as the parent process. Child processes inherit certain attributes from their parent and may have their own execution context.
- **Daemon Process:** A background process that runs continuously and provides specific services or functions. Daemons often perform tasks such as handling system events, network services, or scheduled jobs.
- **Orphan Process:** A child process whose parent process has terminated or ended unexpectedly. Orphan processes are typically adopted by the operating system or an init process to ensure their completion.

- **Zombie Process:** A process that has completed execution but still has an entry in the process table. Zombie processes exist until their parent acknowledges their termination, after which they are removed from the system.
- **Critical Process:** A process that is essential for the proper functioning of the system. Critical processes often handle core system functions, and their failure can lead to system instability.
- **Interactive Process:** A process that interacts directly with the user or responds to user input in real-time. Interactive processes often have a graphical user interface (GUI) or command-line interface for user interaction.
- **Batch Process:** A process that is executed without direct user interaction. Batch processes are typically scheduled to run at specific times or in response to certain events, and they may process large volumes of data.
- **User-Level Process:** A process that runs in user space rather than kernel space. User-level processes are subject to user-level protection mechanisms and do not have direct access to hardware resources.
- **System-Level Process:** A process that runs in kernel space and has privileged access to hardware resources. System-level processes handle critical system functions and interact closely with the operating system kernel.
- **Cooperative Process:** A process that voluntarily yields control to other processes, typically through cooperative multitasking. Cooperative processes work together to share resources and execute tasks.
- **Preemptive Process:** A process that can be interrupted or preempted by the operating system, allowing other processes to run. Preemptive processes are essential for preemptive multitasking, ensuring fair CPU time allocation.
- **Foreground Process:** A process that actively interacts with the user and requires user input. It typically runs in the foreground, and the user interface may be blocked until the process completes.

1.4.2 Process State Transition Diagram

Processes in an operating system go through various states during their lifecycle. Here are the typical states a process can go through:

1. **New:** In this state, a process is being created. The operating system is allocating resources, initializing data structures, and preparing the process for execution. Once the setup is complete, the process transitions to the ”Ready” state.
2. **Ready:** A process in the ”Ready” state is prepared to execute but is waiting for the CPU to be assigned. Multiple processes may be in this state, and the operating system scheduler determines which one to execute next.
3. **Running:** When a process is assigned the CPU, it enters the ”Running” state. The CPU executes the instructions of the process during this state. A process typically moves to this state from the ”Ready” state, and it can stay here until it voluntarily relinquishes the CPU or until the operating system decides to preempt it.
4. **Waiting (Blocked):** A process enters the ”Waiting” state when it cannot proceed until a specific event occurs, such as the completion of an I/O operation or the reception of a signal. While in this state, the process is not using the CPU and is temporarily blocked. Once the required event occurs, the process moves back to the ”Ready” state.
5. **Terminated:** The ”Terminated” state indicates that a process has completed its execution. This could be due to reaching the end of its program or encountering an error. In this state, the operating system releases the resources allocated to the process.

1.5 CPU Scheduling

CPU scheduling is a key component of operating systems that manages the execution of processes in a computer system with a single central processing unit (CPU). The primary goal of CPU scheduling is to optimize system performance by efficiently utilizing the CPU and minimizing the waiting time for processes in the ready queue.

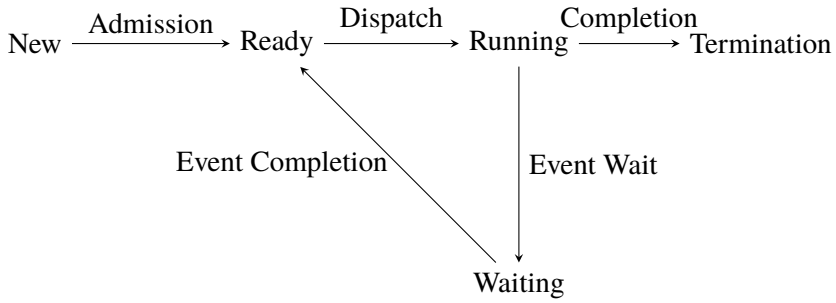

Figure 1.1: Process State Transition Diagram

1.5.1 Various CPU Scheduling Algorithms:

1. **First-Come-First-Serve (FCFS):**
   - **Description:** Processes are executed in the order they arrive in the ready queue.
   - **Advantages:** Simple and easy to implement.
   - **Disadvantages:** May result in poor average waiting time, known as the ”convoy effect.”
2. **Shortest Job Next (SJN) or Shortest Job First (SJF):**
   - **Description:** The process with the shortest burst time is scheduled first.
   - **Advantages:** Minimizes average waiting time.
   - **Disadvantages:** Difficult to predict burst times accurately.
3. **Priority Scheduling:**
   - **Description:** Each process is assigned a priority, and the process with the highest priority is scheduled first.
   - **Advantages:** Allows for prioritization of important processes.
   - **Disadvantages:** May lead to starvation if lower-priority processes are consistently postponed.
4. **Round Robin (RR):**
   - **Description:** Each process is assigned a fixed time slice or quantum. When a process's time expires, it is moved to the back of the queue.
   - **Advantages:** Fair distribution of CPU time.
   - **Disadvantages:** High turnaround time for certain types of workloads.
5. **Multilevel Queue Scheduling:**
   - **Description:** Processes are divided into multiple queues, each with a different priority level. Processes move between queues based on their behavior and priority.
   - **Advantages:** Efficient for systems with diverse workloads.
   - **Disadvantages:** May suffer from issues like starvation and priority inversion.
6. **Multilevel Feedback Queue Scheduling:**
   - **Description:** Similar to multilevel queue scheduling, but processes can move between queues based on their behavior over time.
   - **Advantages:** Adapts to changing process behavior.
   - **Disadvantages:** Complex to implement.
7. **Highest Response Ratio Next (HRRN):**
   - **Description:** Prioritizes processes based on their response ratio, which considers waiting time and expected burst time.
   - **Advantages:** Minimizes response time.
   - **Disadvantages:** Can be computationally intensive.
8. **Lottery Scheduling:**
   - **Description:** Each process is assigned a number of lottery tickets. The scheduler randomly selects a ticket,

and the corresponding process is scheduled.
- **Advantages:** Provides a probabilistic approach to scheduling.
- **Disadvantages:** May not guarantee fairness.

1.6 Thread

Definition: A **thread** is the smallest unit of execution within a process. In a multi-threaded environment, a process can be divided into multiple threads, each of which can independently execute code. Threads within the same process share the same resources, including memory space, file descriptors, and other process-specific attributes.

Key characteristics of threads include:

- **Concurrency:** Threads within a process can execute concurrently, allowing for parallelism and improved performance.
- **Shared Resources:** Threads in the same process share the same address space and resources, simplifying communication and data sharing.
- **Lightweight:** Threads are generally lightweight compared to processes, as they share resources and require less overhead to create and manage.
- **Communication:** Threads within a process can communicate through shared data or inter-thread communication mechanisms provided by the programming language or operating system.

Threads are commonly used to parallelize tasks, improve responsiveness in user interfaces, and optimize resource utilization in multi-core systems.

1.6.1 Types of Threads

1. **User-Level Threads:** Threads that are managed by user-level thread libraries rather than the operating system kernel. User-level threads are lightweight and faster to create but may not take full advantage of multi-core processors.
2. **Kernel-Level Threads:** Threads that are managed by the operating system kernel. Kernel-level threads have the advantage of being able to run in parallel on multi-core systems, but they may have higher overhead compared to user-level threads.

1.6.2 Thread Models

Thread models refer to different approaches or strategies for implementing and managing threads in a programming or operating system environment. Thread models dictate how threads are created, scheduled, and synchronized. Here are some common thread models:

1. **Many-to-One Model:**
   - **Description:** Many user-level threads are mapped to a single kernel-level thread. All thread management and scheduling are handled by the user-level thread library, and the operating system is unaware of the existence of threads.
   - **Advantages:** Lightweight and efficient for systems with limited thread support.
   - **Disadvantages:** Limited parallelism; if one thread is blocked, all threads are affected.
2. **One-to-One Model:**
   - **Description:** Each user-level thread corresponds to a kernel-level thread. The operating system manages and schedules each thread independently. This model allows for true parallelism, as multiple threads can run simultaneously on multiple processors.
   - **Advantages:** Provides better parallelism; one blocked thread does not affect others.
   - **Disadvantages:** Thread creation and management can be more resource-intensive.

3. **Many-to-Many Model:**
   - **Description:** Many user-level threads are multiplexed onto a smaller or equal number of kernel-level threads. Both user-level and kernel-level threads can be scheduled independently. This model aims to combine the efficiency of the many-to-one model with the parallelism of the one-to-one model.
   - **Advantages:** Balances efficiency and parallelism.
   - **Disadvantages:** Complexity in managing interactions between user-level and kernel-level threads.
4. **Hybrid Model:**
   - **Description:** Combines features of multiple thread models. For example, a system may use one-to-one mapping for threads within a process and many-to-one mapping for threads across different processes.
   - **Advantages:** Provides flexibility in optimizing thread management for different scenarios.
   - **Disadvantages:** Can increase system complexity.

1.7 Synchronization

Process or thread synchronization is a mechanism employed in operating systems to control the access to shared resources by multiple processes or threads. When multiple processes or threads run concurrently, there may be scenarios where they need to coordinate their activities to ensure proper execution and avoid conflicts. Synchronization mechanisms help in achieving orderly and predictable execution of concurrent processes or threads.

1.7.1 Some key concepts related to process or thread synchronization

Critical Section:

The critical section is a segment of code in a process or thread that accesses shared resources. Only one process or thread is allowed to execute within the critical section at a time.

Mutual Exclusion:

Mutual exclusion ensures that only one process or thread can enter the critical section at a time. This prevents multiple processes or threads from simultaneously modifying shared data, avoiding data corruption and inconsistency.

Semaphore:

A semaphore is a synchronization primitive used to control access to a shared resource. It maintains a counter, and processes or threads must acquire and release the semaphore to enter and exit the critical section.

Mutex (Mutual Exclusion):

A mutex, or mutual exclusion, is a synchronization primitive utilized in concurrent programming to enforce exclusive access to a shared resource. It acts as a lock, permitting only one thread or process to enter a critical section of code at any given time.

Condition Variable:

A condition variable is used to signal and control the order of execution of processes or threads. It allows processes or threads to wait until a certain condition is met before proceeding.

Deadlock:

Deadlock is a situation where two or more processes or threads are unable to proceed because each is waiting for the other to release a resource. Proper synchronization mechanisms and careful design are required to avoid deadlock.

Race Condition:

A race condition occurs when the behavior of a system depends on the relative timing of events, such as the order in which processes or threads are scheduled. Race conditions can lead to unpredictable results and data inconsistencies.

Atomic Operations:

Atomic operations are those that are performed as a single, indivisible unit. In the context of synchronization, atomic operations are often used to ensure that certain critical operations are executed without interruption.

Barrier:

A barrier is a synchronization construct that forces a group of processes or threads to wait until all members of the group have reached a certain point in their execution before any of them proceeds.

Read-Write Lock:

A read-write lock allows multiple threads to read a shared resource concurrently, but only one thread can write to the resource at a time. This is useful when reads do not modify the shared data.

Busy Waiting:

Busy waiting, also known as spinning or busy looping, is a synchronization technique where a process or thread repeatedly checks for a condition to be true, without performing any significant work in the meantime. Instead of yielding the processor or putting the process to sleep, the process continuously executes a tight loop until the desired condition is met.

1.7.2 Differences between Mutex and Semaphore

Table 1.1: Differences between Mutex and Semaphore

Mutex	Semaphore
Mutex provides exclusive access to a shared resource, allowing only one thread to enter the critical section at a time.	Semaphore is a generalized synchronization primitive that can be used to control access to a shared resource by multiple threads simultaneously.
Mutex is typically used for protecting critical sections in a program where mutual exclusion is essential.	Semaphore is used for a broader range of synchronization scenarios, including signaling, coordination, and controlling access to a pool of resources.
Mutex has a binary state, indicating whether it is locked or unlocked.	Semaphore has a counter that can be incremented or decremented, allowing multiple threads to acquire and release it.
Acquiring and releasing a mutex is performed by the same thread or process.	Acquiring and releasing a semaphore can be done by different threads or processes.
Mutex is simpler and more lightweight, suitable for scenarios where only mutual exclusion is required.	Semaphore is more versatile but may introduce additional complexity due to its multiple states and functionalities.

1.7.3 Critical Section Problem Solution and Desired Properties

The critical section problem is concerned with finding a solution to coordinate the execution of processes or threads in such a way that they can safely access and manipulate shared resources without interfering with each other.

1.7.3.1 Desired Properties for Critical Section Problem Solution:

1. **Mutual Exclusion:**
   - **Desired Property:** Only one process or thread can execute in the critical section at a time.
   - **Explanation:** This property ensures that no two processes or threads concurrently execute their critical sections, preventing conflicts and preserving the integrity of shared data.
2. **Progress:**
   - **Desired Property:** If no process or thread is in its critical section and some processes or threads want to enter the critical section, then one of them should be able to enter.

- **Explanation:** This property ensures that progress is made even if multiple processes or threads are contending for access to the critical section.

3. **Bounded Waiting:**
   - **Desired Property:** There exists a bound on the number of times that other processes or threads are allowed to enter their critical sections after a process or thread has made a request to enter its critical section and before that request is granted.
   - **Explanation:** This property prevents a process or thread from being indefinitely delayed in entering its critical section, ensuring fairness in resource allocation.

1.7.3.2 Solutions to the Critical Section Problem:

1. **Mutex (Mutual Exclusion):**
   - **Solution:** Use mutex locks to enforce exclusive access to the critical section. A process or thread must acquire the mutex before entering the critical section and release it afterward.
   - **Desired Properties:** Mutual Exclusion, Progress, Bounded Waiting.
2. **Semaphore:**
   - **Solution:** Use semaphores to control access to the critical section. Semaphores can have values greater than 1, allowing multiple threads to enter the critical section simultaneously.
   - **Desired Properties:** Mutual Exclusion, Progress, Bounded Waiting.
3. **Monitors:**
   - **Solution:** Encapsulate shared data and critical sections within a monitor, providing a higher-level abstraction that automatically handles synchronization.
   - **Desired Properties:** Mutual Exclusion, Progress, Bounded Waiting.
4. **Atomic Operations:**
   - **Solution:** Use hardware or atomic instructions to perform critical operations as a single, indivisible unit.
   - **Desired Properties:** Mutual Exclusion.

1.7.4 Classical Synchronization Problems

Classical synchronization problems are well-known challenges in concurrent programming that involve coordinating the activities of multiple processes or threads to achieve a specific goal. These problems often highlight the need for synchronization mechanisms to prevent race conditions, deadlocks, and other concurrency-related issues. Some classical synchronization problems include:

1. **Producer-Consumer Problem:** Involves two types of processes, producers that generate data, and consumers that consume the data. The challenge is to ensure that the producers and consumers can work concurrently without data corruption or deadlock.
2. **Reader-Writer Problem:** Deals with multiple readers and writers accessing a shared resource. Readers can access the resource simultaneously, but writers must have exclusive access to update it. The goal is to maximize parallelism while ensuring data consistency.
3. **Dining Philosophers Problem:** Represents a scenario where multiple philosophers sit around a dining table. Each philosopher alternates between thinking and eating, but they need access to shared forks. The challenge is to avoid deadlock and ensure that philosophers can eat without conflicting with each other.
4. **Bounded-Buffer Problem (or Producer-Consumer with a finite buffer):** Extends the producer-consumer problem by introducing a finite-size buffer. Producers must wait if the buffer is full, and consumers must wait if the buffer is empty.
5. **The Sleeping Barber Problem:** Models a barber shop with one barber and multiple customers. Customers arrive and wait for the barber, who serves one customer at a time. The problem is to synchronize customer arrivals and barber service to avoid race conditions.

6. **Cigarette Smokers Problem:** Involves three processes representing smokers with different ingredients and a mediator. The challenge is to synchronize the processes so that a smoker with the right ingredients can smoke only when the mediator provides them.

1.8 Solution of Classical Synchronization Problems Using Semaphores/Mutex

1.8.1 Solution to the Reader-Writer Problem using Semaphores

Refer Algorithm 1.

Algorithm 1 Reader-Writer Problem Solution with Semaphores

```
Input:
readers_count: semaphore controlling access to the number of readers
writers_count: semaphore controlling access to the number of writers
mutex: semaphore controlling access to the critical sections
read_mutex: semaphore controlling access to the reader-specific critical section
procedure READER
    while true do
        WAIT(readers_count)                          ▷ Increment the number of readers
        WAIT(read_mutex)                             ▷ Enter reader-specific critical section
        SIGNAL(readers_count)                        ▷ Decrement the number of readers
        READDATA                                     ▷ Read data from the shared resource
        WAIT(mutex)                                  ▷ Enter critical section
        if readers_count == 0 then
            SIGNAL(writers_count)                    ▷ Allow writers to access
        SIGNAL(mutex)                                ▷ Exit critical section
        SIGNAL(read_mutex)                           ▷ Exit reader-specific critical section
procedure WRITER
    while true do
        WAIT(writers_count)                          ▷ Increment the number of writers
        WAIT(mutex)                                  ▷ Enter critical section
        SIGNAL(writers_count)                        ▷ Decrement the number of writers
        WRITEDATA                                    ▷ Write data to the shared resource
        SIGNAL(mutex)                                ▷ Exit critical section
```

1.8.2 Bounded Buffer Problem Solution with Semaphores

Refer Algorithm 2.

1.8.3 Dining Philosophers Problem Solution with Semaphores

Refer Algorithm 3.

1.9 Deadlock

A deadlock is a situation in computing where two or more processes are unable to proceed because each is waiting for the other to release a resource.

The occurrence of a deadlock requires the simultaneous satisfaction of four conditions, often known as the Coffman conditions:

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode. This means that only one process at a time can use the resource.

Algorithm 2 Bounded Buffer Problem Solution with Semaphores

Input:
buffer_size: the size of the bounded buffer
mutex: semaphore controlling access to the buffer
empty_slots: semaphore tracking the number of empty slots in the buffer
filled_slots: semaphore tracking the number of filled slots in the buffer
procedure PRODUCER
    while *true* **do**
        PRODUCEITEM ▷ Generate an item to be added to the buffer
        WAIT(*empty_slots*) ▷ Wait for an empty slot in the buffer
        WAIT(*mutex*) ▷ Enter critical section
        ADDITEMTOBUFFER ▷ Add the item to the buffer
        SIGNAL(*mutex*) ▷ Exit critical section
        SIGNAL(*filled_slots*) ▷ Signal that a slot is filled
procedure CONSUMER
    while *true* **do**
        WAIT(*filled_slots*) ▷ Wait for a filled slot in the buffer
        WAIT(*mutex*) ▷ Enter critical section
        REMOVEITEMFROMBUFFER ▷ Remove an item from the buffer
        SIGNAL(*mutex*) ▷ Exit critical section
        SIGNAL(*empty_slots*) ▷ Signal that a slot is empty
        CONSUMEITEM ▷ Consume the item

Algorithm 3 Dining Philosophers Problem Solution with Semaphores

Input:
num_philosophers: the number of philosophers
mutex: semaphore controlling access to critical sections
forks[*num_philosophers*]: semaphores representing the forks, initialized to 1
procedure PHILOSOPHER(*id*)
    while *true* **do**
        THINK ▷ Philosopher is thinking
        WAIT(*mutex*) ▷ Enter critical section
        WAIT(*forks*[*id*]) ▷ Pick up left fork
        WAIT(*forks*[(*id* + 1) % *num_philosophers*]) ▷ Pick up right fork
        SIGNAL(*mutex*) ▷ Exit critical section
        EAT ▷ Philosopher is eating
        WAIT(*mutex*) ▷ Enter critical section
        SIGNAL(*forks*[*id*]) ▷ Put down left fork
        SIGNAL(*forks*[(*id* + 1) % *num_philosophers*]) ▷ Put down right fork
        SIGNAL(*mutex*) ▷ Exit critical section

2. **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently held by other processes.
3. **No Preemption:** Resources cannot be forcibly taken away from a process; they can only be released voluntarily.
4. **Circular Wait:** There must be a circular chain of two or more processes, each waiting for a resource held by the next one in the chain.

1.9.1 Resource Allocation Graph

A resource allocation graph (RAG) is a graphical representation used in the study and analysis of resource allocation and potential deadlocks in a system with multiple processes and resources.
The key components and concepts of a resource allocation graph are:

- **Processes (Nodes):** In a resource allocation graph, each process in the system is represented by a node. Processes are entities that execute independently and may request and release resources during their execution.
- **Resources (Nodes or Vertices):** Resources, such as printers, CPUs, memory, or any other sharable entity, are also represented as nodes in the graph. Resources are the objects that processes may request and hold.
- **Resource Requests (Edges):** Directed edges between processes and resources indicate resource requests. An edge from a process to a resource signifies that the process has requested the resource.
- **Resource Allocations (Edges):** Directed edges from resources to processes represent resource allocations. An edge from a resource to a process indicates that the resource is currently allocated to that process.

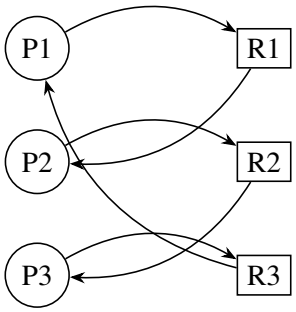

Figure 1.2: Resource Allocation Graph - Potential Deadlock

1.9.1.1 Safe Sequence

A ”safe sequence” refers to a sequence of processes in which each process can complete its execution without causing a deadlock. Specifically, a safe sequence ensures that every process in the sequence can obtain all the resources it needs, execute to completion, and release all its resources before the next process begins.

1.9.2 Managing and Mitigating the Risk of Deadlocks

Deadlock prevention and avoidance are two strategies employed in operating systems and concurrent systems to manage and mitigate the risk of deadlocks.

1.9.2.1 Deadlock Prevention

Deadlock prevention involves designing the system in such a way that at least one of the necessary conditions for a deadlock is not satisfied. The four Coffman conditions for a deadlock are:

1. Mutual Exclusion: Processes must request exclusive control of resources.
2. Hold and Wait: Processes must hold resources while waiting for others.
3. No Preemption: Resources cannot be forcibly taken from a process.
4. Circular Wait: A circular chain of processes must exist, with each waiting for a resource held by the next one.

1.9.2.2 Methods for Deadlock Prevention:

1. **Mutual Exclusion Relaxation:** Allow multiple processes to share certain resources rather than granting exclusive access.
2. **Hold and Wait Elimination:** Require processes to request all necessary resources at once, or release held resources before requesting new ones.
3. **Preemption:** Allow the operating system to preemptively take resources from processes when necessary. This approach is challenging and may not be applicable to all types of resources.
4. **Circular Wait Elimination:** Impose a total ordering of resource types and require processes to request resources in increasing order.

1.9.3 Deadlock Avoidance:

Deadlock avoidance involves dynamically managing resource allocation to ensure that the system remains in a safe state, i.e., a state where deadlock cannot occur. The system checks for the potential for deadlock before allocating resources and denies a request if it could lead to a deadlock.

1.9.3.1 Banker's Algorithm:

The Banker's algorithm is a well-known deadlock avoidance algorithm. It uses a set of heuristics to determine whether granting a resource request would put the system in a safe state. The key idea is to avoid allocating resources if doing so might lead to an unsafe state where deadlock is possible.

Algorithm 4 Banker's Algorithm

Input:
$Available[1 \dots m]$: the number of available resources for each type
$Max[i, 1 \dots m]$: the maximum demand of process P_i for each resource type
$Allocation[i, 1 \dots m]$: the number of resources of each type currently allocated to process P_i
$Need[i, 1 \dots m]$: the remaining need of process P_i for each resource type
Output:
Safe sequence or indication of unsafe state
function BANKER($Available$, Max, $Allocation$)
    Initialize arrays $Work = Available$ and $Finish[1 \dots n] =$ false
    Initialize an empty list $SafeSequence$
    while there exists an index i such that $Finish[i] =$ false and $Need[i] \leq$ Work **do**
        $Work \leftarrow Work + Allocation[i]$
        $Finish[i] \leftarrow$ true
        Append P_i to $SafeSequence$
    if all processes are marked as finished **then**
        return $SafeSequence$ ▷ System is in a safe state
    else
        return unsafe state ▷ Deadlock may occur

Complexity of Banker's Algorithm:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. The complexity of the algorithm is generally considered to be polynomial, specifically $O(n \cdot m^2)$, where:

- n is the number of processes in the system.
- m is the number of resource types.

The key data structures used in the Banker's algorithm:

1. **Available:** This array represents the number of available resources of each type in the system. The array is denoted as $Available[1 \ldots m]$, where m is the number of resource types. The Banker's algorithm checks this array to determine if resources can be allocated to processes without causing a deadlock.
2. **Max:** The Max matrix specifies the maximum demand of each process for each resource type. It is denoted as $Max[i, 1 \ldots m]$, where i represents the process index and m is the number of resource types. This matrix is used to check whether allocating additional resources to a process could lead to an unsafe state.
3. **Allocation:** The Allocation matrix represents the number of resources of each type currently allocated to each process. It is denoted as $Allocation[i, 1 \ldots m]$, where i is the process index and m is the number of resource types. This matrix is used to track the current resource allocation status.
4. **Need:** The Need matrix represents the remaining needs of each process for each resource type. It is denoted as $Need[i, 1 \ldots m]$, where i is the process index and m is the number of resource types. The Need matrix is calculated as the difference between the Max matrix and the Allocation matrix. It helps determine whether a process can safely request additional resources without causing a deadlock.

Advantages and Disadvantages:

- Advantages of Avoidance: Provides a dynamic solution that allows resource allocation while avoiding deadlock. Suitable for systems with varying resource demands.
- Disadvantages of Avoidance: May lead to underutilization of resources as processes are sometimes unable to acquire the resources they need.

1.9.3.2 Prevention vs. Avoidance:

Deadlock prevention is a more static approach applied during the system design phase, whereas deadlock avoidance is a dynamic strategy applied during runtime.

1.10 Types of Memory

There are several types of memories, each serving a specific purpose in storing and retrieving data. Here are some of the common types of memories:

1. **Primary Memory (Main Memory)**
   - **RAM (Random Access Memory):** Volatile memory used for temporary storage of data and program code that is actively being used by the CPU. Data is lost when power is turned off.

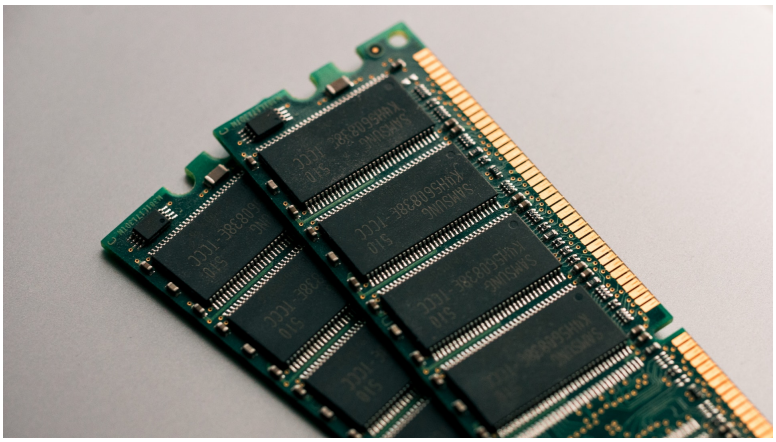

Figure 1.3: Random Access Memory

- **ROM (Read-Only Memory):** Non-volatile memory that contains firmware or permanent software instructions. The data in ROM is typically not modified during normal operation.

2. **Secondary Memory (Storage Devices)**
   - **Hard Disk Drives (HDD):** Non-volatile data storage devices with high capacity for long-term storage. Slower access compared to RAM.

Figure 1.4: Hard Disk Drive

   - **Solid State Drives (SSD):** Non-volatile storage devices that use NAND-based flash memory for faster data access than HDDs.

Figure 1.5: SSD

   - **Flash Drives (USB Drives):** Portable and non-volatile storage devices using NAND flash memory.
   - **Optical Drives (CDs, DVDs, Blu-rays):** Non-volatile storage devices that use optical technology to read and write data.

3. **Cache Memory L1, L2, and L3 Cache:** Small-sized, high-speed memory units located directly on or near the CPU. They store frequently accessed instructions and data for faster retrieval.
4. **Registers**

   CPU Registers: Fast, small-sized storage locations within the CPU. They store data, addresses, and intermediate results during program execution. Different types of registers serve various purposes in a computer system. Here are some common types of registers:
   - **Data Register (DR):** Also known as the accumulator, the data register is used for storing intermediate data during arithmetic and logic operations. It's often the primary register for arithmetic calculations.
   - **Address Register (AR):** The address register holds the memory address of the data that needs to be ac-

cessed or modified. It is crucial for memory-related operations.

- **Program Counter (PC):** The program counter keeps track of the address of the next instruction to be executed. It plays a vital role in the control flow of a program.
- **Memory Buffer Register (MBR):** The memory buffer register temporarily stores data that is read from or written to the memory. It acts as a buffer between the CPU and the memory.
- **Memory Address Register (MAR):** The memory address register holds the address of the memory location to be accessed or modified. It works in conjunction with the memory buffer register.
- **Index Register:** An index register is used for indexing, especially in addressing modes that involve indirect addressing or array operations. It holds an offset value to be added to a base address.
- **Status Register (SR/PSW):** The status register, also known as the program status word (PSW), contains flags that represent the current state of the processor. These flags indicate conditions such as zero, carry, overflow, etc.
- **Stack Pointer (SP):** The stack pointer points to the top of the stack in memory. It is crucial for managing the call stack in subroutine and function calls.
- **General-Purpose Registers (GPRs):** These registers can be used for a variety of purposes and are not dedicated to a specific function. They are versatile and can be employed by the programmer as needed.
- **Floating-Point Registers:** In systems that support floating-point arithmetic, there are registers specifically designed to store and handle floating-point numbers. They are distinct from integer registers.
- **Vector Registers:** In vector processors or SIMD (Single Instruction, Multiple Data) architectures, vector registers store multiple data elements simultaneously, allowing parallel processing.

5. **Virtual Memory:** Virtual memory is a memory management technique used by operating systems to provide an illusion to users and applications that they have a larger amount of available memory than is physically installed on the computer. It allows programs to use more memory than the physical RAM (Random Access Memory) available by temporarily transferring data to and from the storage, typically the computer's hard drive or SSD.

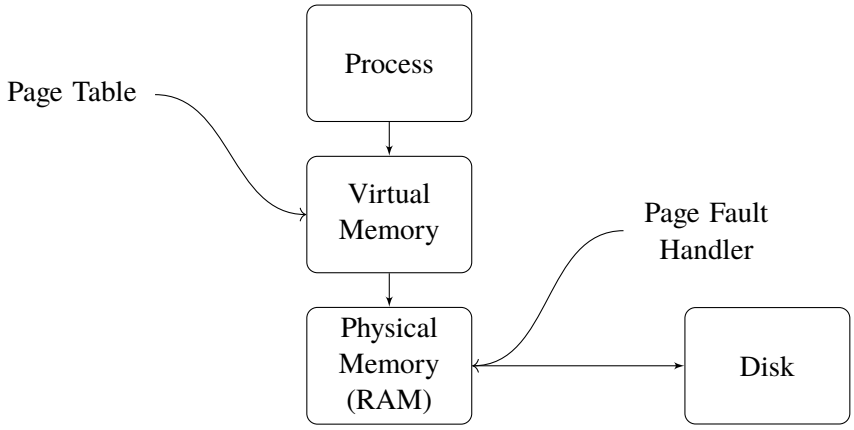

Figure 1.6: Virtual Memory Representation

6. **Graphics Memory (GPU Memory)**
   Video RAM (VRAM): Memory on a graphics card used to store graphical data, textures, and frame buffers.
7. **Memory Cards**
   SD Cards, microSD Cards: Removable non-volatile storage devices used in cameras, smartphones, and other portable devices. A few sample SD cards are shown in Figure 1.7

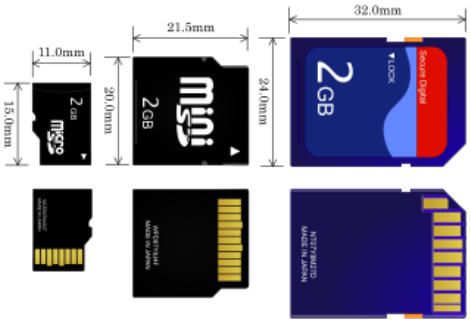

Figure 1.7: SD Cards

1.11 Memory Management Schemes

1.11.1 Fixed Partitioning:

Description: The physical memory is divided into fixed-size partitions, and each partition is assigned to a process.

Fixed partitioning simply means that the total memory is divided into fixed-size regions or partitions, and each partition is then assigned to a process.

In fixed partitioning, it is not strictly necessary for all partitions to have equal sizes, although it is a common implementation.

Partition 1 (200 KB)	Partition 2 (200 KB)	Partition 3 (200 KB)	Partition 4 (200 KB)	Partition 5 (200 KB)

- **Advantages:**
  - Simple and easy to implement.
  - No external fragmentation.
- **Disadvantages:**
  - Inflexible for varying memory requirements.
  - Internal fragmentation can occur if a partition is larger than the size of the process.

1.11.2 Variable Partitioning:

Description: Memory is divided into variable-sized partitions to accommodate processes with different memory requirements.

- **Advantages:**
  - More flexible than fixed partitioning.
  - Helps reduce internal fragmentation.
- **Disadvantages:**
  - May suffer from external fragmentation.
  - Requires more complex memory management algorithms.

Process A (150 KB)	Process B (300 KB)	Process C (200 KB)	Unused (Unused)	Unused (Unused)

Table 1.2: Comparative Analysis of Memory Allocation Algorithms

Algorithm	Advantages	Disadvantages	Comments
First Fit	• Simple and easy to implement. • Quick allocation.	• May lead to fragmentation. • Suboptimal space utilization.	Suitable for systems with low fragmentation tolerance and quick allocation needs.
Best Fit	• Reduces fragmentation effectively. • Efficient use of memory.	• Higher time complexity. • May leave small gaps.	Suitable for systems where reducing fragmentation is a priority, and time complexity is not critical.
Worst Fit	• May reduce long-term fragmentation. • Keeps larger blocks for future use.	• Inefficient use of memory. • May not be effective for short-term allocations.	Suitable for systems where long-term fragmentation reduction is a priority, and memory usage patterns allow for larger blocks.
Next Fit	• Better fragmentation performance than First Fit. • Utilizes contiguous blocks.	• Can still suffer from fragmentation. • May not perform as well as Best Fit.	Improved version of First Fit, suitable for systems with moderate fragmentation tolerance.

1.11.2.1 Memory Allocation Algorithms

1.11.3 Paging:

Description: Memory is divided into fixed-size blocks called pages. Processes are divided into fixed-size blocks as well. Pages of a process do not need to be contiguous in physical memory.

- **Advantages:**
  - Reduces external fragmentation.
  - Simplifies memory management.
- **Disadvantages:**
  - May suffer from internal fragmentation within pages.
  - Requires a page table for address translation.

1.11.3.1 Page Table:

A page table is a data structure used by the operating system to manage the mapping between logical addresses (used by programs) and physical addresses (locations in the physical memory). It plays a crucial role in implementing virtual memory, allowing processes to have an illusion of a contiguous address space while the physical memory may be fragmented.

The typical fields found in a page table:

1. **Page Number (or Page Index):**
   - The page number is an index or identifier for a specific page in the logical address space of a process. It's extracted from the logical address when performing address translation.
   - In the page table, each row corresponds to a page, and the page number serves as the index to access information about that page.

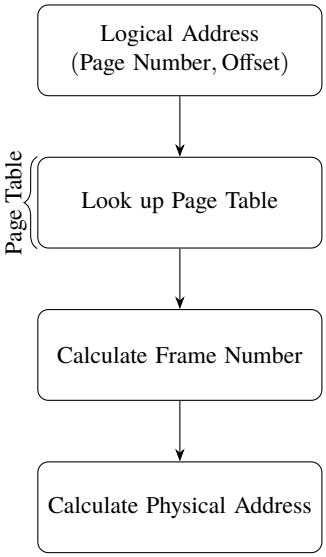

Figure 1.8: Paging Flowchart

2. **Frame Number (or Frame Index):**
   - The frame number represents the corresponding frame (or block) in the physical memory where the content of a specific page is stored.
   - When a page is referenced, the page table is consulted to retrieve the frame number, and the combination of the frame number and the offset within the page yields the physical memory address.
3. **Valid/Invalid Bit:**
   - The valid/invalid bit indicates whether a particular entry in the page table is currently valid or not.
   - If the bit is set to valid, it means the mapping between the page and the frame is valid, and the page can be accessed. If set to invalid, accessing this page would result in a page fault.
4. **Protection/Permission Bits:**
   - These bits define the access permissions for the associated page. Common permission bits include read, write, and execute.
   - For example, a page may be marked as read-only to prevent write operations, or execute-only to prevent modification.
5. **Dirty Bit:**
   - The dirty bit is used to track whether the contents of a page have been modified since it was loaded into memory.
   - This information is crucial for optimizing memory management, especially during page replacement policies.
6. **Reference (or Accessed) Bit:**
   - The reference (or accessed) bit is set whenever the associated page is accessed. It helps in tracking the usage pattern of pages and is used by certain page replacement algorithms.
7. **Other Fields:**
   - Depending on the operating system and hardware architecture, additional fields may exist in a page table. These could include timestamp information, shared/not-shared flags, etc.

A sample page table is depicted in Table 1.3.

1.11.3.2 Logical to Physical Address Conversion:

The conversion of logical to physical address involves the lookup of mapping information (from page table or segment table) and the calculation of the actual physical address based on the offset within the page or segment. This process is essential for providing a virtualized and protected memory space for each process running on the system.

Table 1.3: Page Table with Different Fields

Page Number	Frame Number	Valid	Permission	Dirty	Accessed
0	3	1	RW	0	1
1	5	1	R	1	0
2	1	1	RWX	1	1
3	2	0	–	–	0
4	4	1	RO	0	1

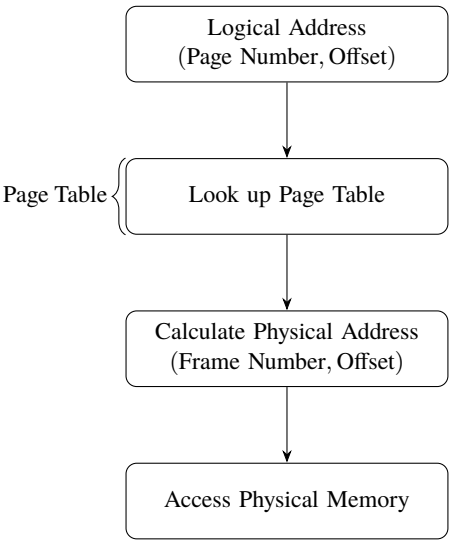

Figure 1.9: Logical to Physical Address Conversion

1.11.3.3 Demand Paging

Demand paging is a memory management scheme used in operating systems to optimize the use of physical memory (RAM). The main idea behind demand paging is to bring in only the necessary pages of a process into memory when they are needed, rather than loading the entire process into memory at once. This approach helps to minimize the initial loading time and efficiently utilize available memory resources.

Key concepts and mechanisms associated with demand paging:

- **Page Table:**
  A page table is used to keep track of the mapping between logical addresses (used by the CPU) and physical addresses (locations in RAM). Each entry in the page table corresponds to a page, which is a fixed-size contiguous block of virtual memory.
- **Page Faults:**
  When a process tries to access a page that is not currently in physical memory, a page fault occurs. This triggers the operating system to bring the required page into memory from the secondary storage (usually the disk).
- **Backing Store (Swap Space):**
  The portion of the disk used to store pages that are not currently in physical memory is called the backing store or swap space. Pages are swapped in and out of this space as needed.
- **Page Replacement:**
  If physical memory is full and a page fault occurs, the operating system needs to decide which page to remove from memory to make room for the required page. This process is known as page replacement.
- **Lazy Loading:**
  Instead of loading an entire process into memory when it starts, only the initial pages needed to execute the program are loaded. Additional pages are loaded as they are referenced during the execution of the program.
- **Copy-on-Write:**

When a page is shared among multiple processes, the operating system uses a copy-on-write strategy. Initially, the pages are shared, and if one process attempts to modify the content of a shared page, a copy of that page is created for the modifying process.

Advantages of Demand Paging

- **Efficient Memory Utilization:** Only the required pages are loaded into memory, reducing wastage of physical memory.
- **Fast Program Loading:** Programs can start quickly because only the necessary portions are loaded initially.
- **Support for Large Programs:** Demand paging enables the execution of programs that are larger than the available physical memory by bringing in only the required pages.

Drawbacks of Demand Paging

- **Page Faults Overhead:** Page faults, especially during initial program execution, can introduce overhead as required pages are brought into memory from secondary storage.
- **I/O Operations:** Swapping pages in and out of the backing store involves I/O operations, which may impact system performance.
- **Page Replacement Overhead:** The process of deciding which page to replace in case of a page fault introduces computational overhead and may require the use of complex algorithms.

1.11.3.4 Comparative Analysis of Page Replacement Algorithms

Refer Table 1.4.

1.11.3.5 Problems and Anomalies Associated with Page Replacement Algorithms

Several problems and anomalies are associated with page replacement algorithms, and addressing these challenges is essential for optimizing system performance.

Belady's Anomaly: Belady's Anomaly refers to the phenomenon where increasing the number of page frames (allocated memory) may result in an increase in the number of page faults.

Significance: This anomaly challenges the intuition that providing more memory should improve performance.

Optimality vs. Implementability: The optimal page replacement algorithm is to always replace the page that will not be used for the longest period in the future. However, this requires knowledge of future references, which is practically impossible.

Algorithms like the Least Recently Used (LRU) approximation attempt to strike a balance between optimality and implementability.

Thrashing: Thrashing occurs when a high page-fault rate leads to a constant state of swapping pages in and out of memory, causing the system to spend more time on page faults than on useful computation.

Significance: Thrashing can severely degrade system performance, and effective page replacement algorithms are needed to mitigate this issue.

Local vs. Global Replacement: The decision to replace a page can be made based on only the local history of a process or the global history of all processes.

Significance: Choosing between local and global replacement policies can impact how effectively pages are managed, and there is often a trade-off between fairness and efficiency.

Table 1.4: Comparison of Page Replacement Algorithms

Algorithm	Advantages	Disadvantages	Comments
FIFO (First-In-First-Out)	• Simple and easy to implement. • Low overhead.	• May suffer from Belady's anomaly (increased page faults with additional frames). • Poor performance in certain scenarios.	Widely used due to simplicity, but not always optimal.
LRU (Least Recently Used)	• Generally effective in reducing page faults. • Can adapt to changing access patterns.	• Implementation complexity and overhead. • May suffer from the difficulty of accurate timestamp tracking.	Commonly used in practice, but may require efficient implementation.
LFU (Least Frequently Used)	• Effective for certain access patterns. • Can adapt to varying workloads.	• Difficulty in accurately maintaining frequency counts. • May not perform well in scenarios with sudden changes in access patterns.	Suitable for scenarios where infrequently used pages should be replaced.
Optimal	• Theoretically optimal solution (reduces page faults to the minimum possible).	• Requires knowledge of future page accesses, which is not practical. • Used as a benchmark for comparison with other algorithms.	Used for performance comparison but not practical in real-world implementations.
Clock	• Simplicity and low overhead. • Can be efficient in certain scenarios.	• May not perform well in scenarios with frequent page faults. • May not adapt well to varying access patterns.	A balance between simplicity and efficiency, often used in practical implementations.

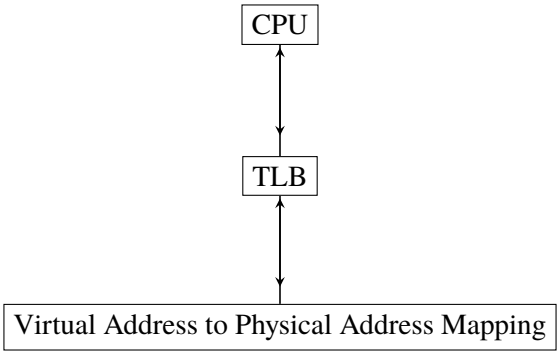

Figure 1.10: TLB

1.11.3.6 Translation Lookaside Buffer (TLB):

It is a cache that stores a mapping between virtual addresses and their corresponding physical addresses in a computer system. TLB is a component of the memory management unit (MMU) and is used in the process of translating virtual addresses generated by the CPU into physical addresses in the main memory.

1.11.4 Segmentation:

Description: Memory is divided into logical segments based on the program's structure (e.g., code segment, data segment). Segments do not need to be contiguous in physical memory.

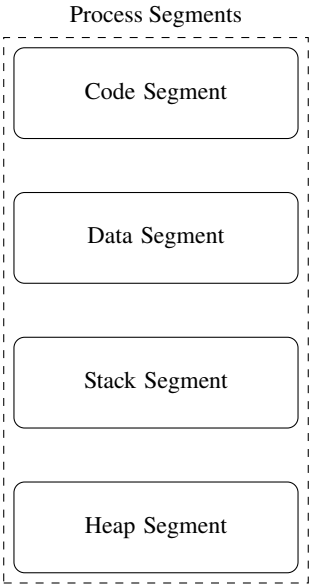

Figure 1.11: Process Segmentation

- **Advantages:**
  - Supports dynamic memory allocation.
  - Provides a logical structure to memory.
- **Disadvantages:**
  - May suffer from fragmentation within segments.
  - More complex than paging.

1.11.4.1 Segmentaion Flowchart

Refer Figure 1.12

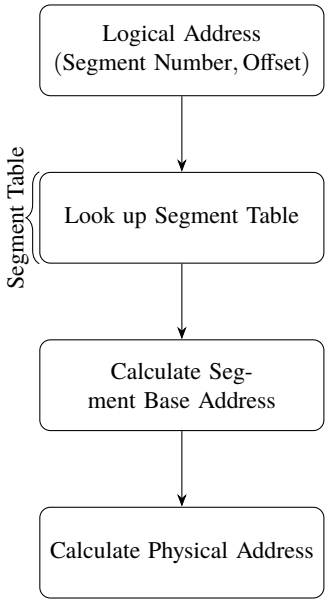

Figure 1.12: Segmentation Flowchart

1.11.4.2 Segment Table

Table 1.5: Segment Table

Segment	Base Address	Limit	Type
1	0x1000	0x0FFF	Code
2	0x5000	0x1FFF	Data
3	0xA000	0x7FFF	Stack

1.11.4.3 Fields in a Segment Table:

1. **Segment Number:** An identifier or index for a specific segment in the logical address space of a process. It distinguishes one segment from another.
2. **Base Address:** The starting physical address in memory where a particular segment begins. The logical address is mapped to the physical address by adding the offset to the base address.
3. **Limit:** The size or length of the segment, indicating the maximum offset allowed within the segment. It helps prevent programs from accessing memory outside their allocated segment.
4. **Access Rights/Permissions:** Defines the access permissions or privileges associated with the segment. Common permission bits include read, write, and execute. These bits control the type of operations that can be performed on the segment.
5. **Segment Type:** Specifies the type of segment, such as code, data, stack, or other types depending on the system. Different types of segments may have different access rights and behaviors.
6. **Present Bit:** Indicates whether the segment is currently loaded into memory (present) or not (absent). If a segment is not present, accessing it would result in a segment fault.
7. **Descriptor Privilege Level (DPL):** Specifies the privilege level required to access the segment. It is used in a multi-level protection mechanism to control access rights based on the privilege level of the executing code.
8. **System/Reserved Bits:** Some bits in the segment table entry might be reserved for system use or future expansion. These bits are typically reserved and should be set to specific values.
9. **Segment Grows Down/Up:** In some systems, a bit may indicate whether the segment grows down or up in memory. This information affects the direction in which the segment expands.
10. **Descriptor Type:** Some architectures use a descriptor type field to distinguish between different types of segment descriptors. For example, code segments and data segments may have different descriptor types.

1.11.5 Combination of Paging and Segmentation:

Description: A combination of paging and segmentation techniques, aiming to combine their advantages and mitigate their disadvantages.

- **Advantages:**
  - Offers the benefits of both paging and segmentation.
- **Disadvantages:**
  - Requires complex management algorithms.

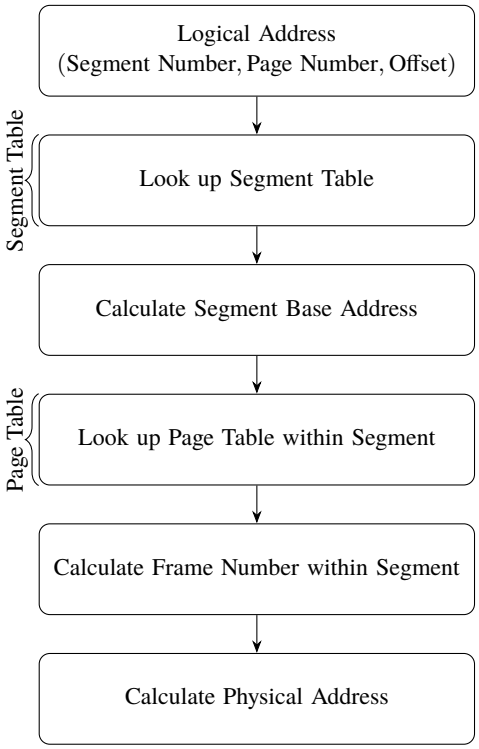

Figure 1.13: Segmented Paging Flowchart

1.12 Secondary Storage

1.12.1 Hard Disk Drives (HDDs)

1. HDDs consist of one or more magnetic platters, typically made of aluminum or glass. Data is stored on these platters in the form of magnetic patterns.
2. Each platter is divided into concentric circles called tracks. Each track is further divided into sectors, which are the smallest storage units on a disk.
3. Read/write heads are positioned above and below each platter to read and write data. They move across the surface of the platters to access different tracks and sectors.
4. The set of tracks at the same position on each platter, i.e., one track from each platter, forms a cylinder. Accessing data on the same cylinder is faster than moving the heads to different positions.
5. The file system organizes data into files and directories. It keeps track of the physical locations of files on the disk.
6. Disks are often partitioned into logical sections. Each partition may have its file system. Partitioning allows for better organization and management of data.
7. The file system maintains a table that maps file names to their physical locations on the disk. The structure may vary based on the file system used (e.g., FAT, NTFS, ext4).

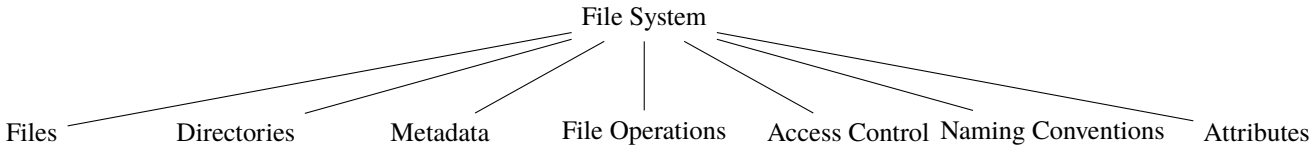

Figure 1.14: A File System

1.12.2 Solid State Drives (SSDs)

1. SSDs use non-volatile memory cells (typically NAND flash) to store data. Data is stored by varying the charge in the memory cells.
2. Data is organized into pages, and pages are grouped into blocks. Writing to an SSD often involves writing an entire block, which may require moving existing data.
3. An SSD has a controller that manages data storage, wear leveling, and garbage collection. Wear leveling ensures that write/erase cycles are distributed evenly across the memory cells.
4. TRIM is a command that allows the operating system to inform the SSD which blocks of data are no longer considered in use. Helps in improving performance and prolonging the lifespan of the SSD.
5. Unlike HDDs, SSDs have no moving parts, resulting in faster data access times and better durability.
6. Similar to HDDs, SSDs use a file system to organize and manage data.

1.13 RAID (Redundant Array of Independent Disks)

RAID is a technology used to improve the performance and reliability of computer storage. It involves combining multiple physical disk drives into a single logical unit. There are different RAID levels, each with its own way of organizing and utilizing these disks.

1.13.1 RAID Levels:

1. **RAID 0 (Striping):**
   Data is split into blocks, and each block is written to a different disk. This improves speed but offers no redundancy. If one disk fails, all data is lost.
2. **RAID 1 (Mirroring):**
   Data is duplicated across two or more disks. If one disk fails, the data is still available on the mirrored disk(s), providing a backup. Storage capacity is reduced as each drive contains identical data.
3. **RAID 5 (Striping with Distributed Parity):**
   Data is striped across multiple disks, and parity information is distributed. RAID 5 provides a balance between speed and fault tolerance. It can withstand the failure of one disk without losing data.
4. **RAID 6 (Striping with Dual Parity):**
   Similar to RAID 5, but can tolerate the failure of two disks simultaneously. RAID 6 provides higher fault tolerance.
5. **RAID 10 (Striping and Mirroring):**
   Combines RAID 0 and RAID 1. Data is both striped for performance and mirrored for redundancy. RAID 10 offers high performance and fault tolerance but requires more disks.

1.14 File System

In Figure 1.14 a general file system is represented.

1.14.0.1 Common File Attributes

Refer Table 1.6.

Table 1.6: Common File Attributes

Attribute	Description
Size	Indicates the size of the file in bytes.
Date Created	Specifies the date and time when the file was originally created.
Date Modified	Represents the date and time when the file was last modified.
Date Accessed	Indicates the date and time when the file was last accessed.
Permissions	Defines the access permissions for the file, such as read, write, and execute.
Read-Only	Specifies whether the file can only be read and not modified.
Hidden	Indicates whether the file is hidden from standard directory listings.
System	Marks the file as a system file, typically used by the operating system.

1.14.0.2 Inode

An *inode*, short for ”index node,” is a data structure used in Unix-like file systems to store information about a file or a directory. Each file or directory in the file system is associated with a unique inode, and the inode contains metadata about the file or directory rather than the actual data. The inode serves as an index or reference to the actual data blocks on the disk where the file’s content is stored.

Key information stored in an inode typically includes:

1. **File Type:** Indicates whether the inode refers to a regular file, directory, symbolic link, or other file types.
2. **File Permissions:** Specifies the access permissions (read, write, execute) for the file owner, group, and others.
3. **Owner Information:** Identifies the user (user ID) who owns the file.
4. **Group Information:** Identifies the group (group ID) associated with the file.
5. **File Size:** Specifies the size of the file in bytes.
6. **Timestamps:** Records various timestamps, including the time the file was created, last modified, and last accessed.
7. **Number of Links:** Indicates the number of hard links pointing to the inode. When a file has multiple hard links, they share the same inode.
8. **Pointers to Data Blocks:** Stores pointers or references to the actual data blocks on the disk where the file content is stored. For small files, these pointers may directly reference the data blocks. For larger files, additional indirect blocks may be used.

1.14.0.3 File Descriptor

A *file descriptor* is a non-negative integer that uniquely identifies an open file or a socket in a computer operating system. It is a fundamental concept in Unix-like operating systems and is used by programs to access files, sockets, or other input/output (I/O) resources. File descriptors are typically managed by the operating system’s kernel.

In Unix-like systems, there are three standard file descriptors associated with every process:

1. **Standard Input (stdin - File Descriptor 0):** Represented by the integer 0, this file descriptor is associated with the process’s input stream. It is commonly used for reading data from the keyboard or another input source.
2. **Standard Output (stdout - File Descriptor 1):** Represented by the integer 1, this file descriptor is associated with the process’s output stream. It is used for writing data to the terminal or another output destination.
3. **Standard Error (stderr - File Descriptor 2):** Represented by the integer 2, this file descriptor is associated with the process’s error output stream. It is used for writing error messages or diagnostic information.

1.15 Disc Scheduling

Table 1.7: Disk Scheduling Algorithms Overview

Algorithm	Description
FCFS	First-Come-First-Serve: Serves requests in the order they arrive. Simple but may result in high seek time.
SSTF	Shortest Seek Time First: Minimizes seek time by serving the closest request. May lead to starvation.
SCAN	SCAN (Elevator): Reduces total movement by traversing in one direction. Delays at ends.
C-SCAN	C-SCAN: Circular SCAN, avoids delays at ends but delays for distant requests.
LOOK	LOOK: Similar to SCAN but does not traverse to ends, reducing delays.
C-LOOK	C-LOOK: Circular LOOK, avoids delays at ends but delays for distant requests.

Table 1.8: Comparative Analysis of Disk Scheduling Algorithms

Algorithm	Advantages	Disadvantages	Suitable Use Cases
First-Come-First-Serve (FCFS)	Simple and easy to implement.	May result in high seek time and poor performance with varying loads.	Suitable for systems with low to moderate I/O loads.
Shortest Seek Time First (SSTF)	Minimizes seek time, better performance than FCFS.	May lead to starvation of some requests if there is a consistently higher number of nearby requests.	Effective for systems with varying I/O loads and seeks.
SCAN	Reduces the total movement of the disk arm, avoiding starvation.	May cause delays for requests at the ends of the disk.	Suitable for systems with a mix of short and long-distance requests.
C-SCAN	Avoids delays for requests at the ends by servicing in a circular manner.	May still result in high seek time for distant requests.	Effective for systems with periodic I/O patterns and avoids starvation.
LOOK	Similar to SCAN but does not go all the way to the end, reducing delays.	May still result in some delays for requests at the farthest point.	Suitable for systems with varying I/O loads and seeks, reducing total arm movement.
C-LOOK	Similar to C-SCAN but avoids delays for requests at the ends.	May still have delays for distant requests, but less than C-SCAN.	Effective for systems with periodic I/O patterns, avoiding starvation and minimizing delays.

1.16 Important Linux Commands

Basic Commands

Command	Description	Example
$ ls	List directory contents	$ ls -l
$ cd	Change directory	$ cd /path/to/directory
$ pwd	Print working directory	$ pwd
$ cp	Copy files or directories	$ cp file1 file2
$ mv	Move files or directories	$ mv file1 /path/to/destination
$ rm	Remove files or directories	$ rm file
$ mkdir	Create a directory	$ mkdir new_directory
$ rmdir	Remove an empty directory	$ rmdir empty_directory
$ cat	Concatenate and display file content	$ cat file.txt
$ grep	Search for a pattern in files	$ grep pattern file.txt
$ chmod	Change file permissions	$ chmod 755 file
$ chown	Change file owner and group	$ chown user:group file

File Handling

Command	Description	Example
File Permissions	Read (`r`), Write (`w`), Execute (`x`)	-
File Ownership	Owner, Group, Others	-
$ touch	Create an empty file	$ touch new_file.txt
$ nano or $ vim	Text editors	$ nano file.txt
$ head and $ tail	Display the beginning or end of a file	$ head file.txt
$ cp and $ rsync	Copy files and directories with options	$ cp -r dir1 dir2

Processes

Command	Description	Example
$ ps	Display information about processes	$ ps aux
$ top or $ htop	Display dynamic view of system processes	$ htop
$ kill and $ killall	Terminate processes	$ kill -9 PID
$ pkill	Signal processes based on name	$ pkill process_name
$ bg and $ fg	Background and foreground processes	$ bg %1
$ jobs	Display background jobs	$ jobs
$ nice and $ renice	Adjust process priority	$ nice -n 10 command

System Information

Command	Description	Example
$ uname	Display system information	$ uname -a
$ hostname	Display or set the system hostname	$ hostname
$ uptime	Display system uptime	$ uptime
$ free	Display memory usage	$ free -m
$ df and $ du	Display disk space usage	$ df -h
$ ifconfig or $ ip	Display network configuration	$ ifconfig
$ who and $ w	Display information about logged-in users	$ who

Package Management

Command	Description	Example
$ apt or $ apt-get	Debian/Ubuntu package management	$ sudo apt-get update
$ yum	Red Hat/CentOS package management	$ sudo yum install package
$ dpkg	Debian package management (individual package operations)	$ dpkg -i package.deb
$ rpm	Red Hat package management (individual package operations)	$ rpm -ivh package.rpm

Networking

Command	Description	Example
$ ping	Check network connectivity	$ ping google.com
$ traceroute or $ mtr	Trace the route to a destination	$ traceroute example.com
$ netstat	Display network statistics	$ netstat -an
$ ss	Socket statistics	$ ss -t
$ iptables or $ firewalld	Configure firewall rules	$ sudo iptables -A INPUT -p tcp –dport 80 -j ACCEPT
$ hostnamectl	Control the system hostname	$ hostnamectl set-hostname newhostname

User and Group Management

Command	Description	Example
$ useradd and $ adduser	Create a new user	$ sudo useradd newuser
$ passwd	Change user password	$ passwd username
$ usermod	Modify user properties	$ sudo usermod -aG groupname username
$ userdel	Delete a user	$ sudo userdel username
$ groupadd, $ addgroup, $ groupdel	Group management	$ sudo groupadd newgroup

System Logs

Command	Description	Example
$ /var/log	Directory containing various log files	-
$ dmesg	Display kernel messages	$ dmesg | grep error
$ journalctl	Query and display messages from the journal	$ journalctl -u servicename

File System

Command	Description	Example
$ mount and $ umount	Mount and unmount file systems	$ sudo mount /dev/sdX1 /mnt
$ df and $ du	Display disk space usage	$ df -h
$ fdisk or $ parted	Partition management	$ sudo fdisk /dev/sdX
$ mkfs and $ mkfs.ext4	Create a file system	$ sudo mkfs.ext4 /dev/sdX1

Shell Scripting

Command	Description	Example

Variables	Store and manipulate data	`#!/bin/bash`
		`name="John"`
		`age=25`
		`echo "Name: $name, Age: $age"`

Conditionals	Make decisions in the script	`#!/bin/bash`
		`if [ "$1" -eq 1 ]; then`
		`  echo "Parameter is 1"`
		`else`
		`  echo "Parameter is not 1"`
		`fi`

Loops	Repeat code execution	`#!/bin/bash`
		`for i in {1..5}; do`
		`  echo "Iteration $i"`
		`done`

Command Substitution	Capture command output	`#!/bin/bash`
		`files=$(ls -l)`
		`echo "Files: $files"`

Functions	Organize code into reusable units	`#!/bin/bash`
		`greet() {`
		`  echo "Hello, $1!"`
		`}`
		`greet "Alice"`

Arguments	Accept input parameters	`#!/bin/bash`
		`echo "Script Name: $0"`
		`echo "First Argument: $1"`
		`echo "Second Argument: $2"`

File Redirection	Manage input and output	`#!/bin/bash`
		`echo "Hello" > output.txt`
		`echo "World" >> output.txt`
		`cat < input.txt`

Security

Command	Description	Example
$ sudo	Execute a command with superuser privileges	$ sudo command
$ chmod	Change file permissions	$ chmod 700 file
$ chown	Change file owner and group	$ chown user:group file
$ ufw or $ iptables	Configure firewall rules	$ sudo ufw allow 80
$ fail2ban	Protect against brute-force attacks	$ sudo fail2ban-client status

Operating Systems Formulas and Facts

CPU Scheduling

1. **Turnaround Time:**
   (a). $Turnaround\ Time = Completion\ Time - Arrival\ Time$
   (b). $Turnaround\ Time =$ Waiting Time + Burst Time
2. **Waiting Time:** $Waiting\ Time = Turnaround\ Time - Burst\ Time$
3. **Response Time:**
   - The time from submitting a request until the first response is produced.
   - Response Time = Time of First Response − Arrival Time
4. **Throughput:** Throughput $= \frac{\text{Number of processes completed}}{\text{Total time}}$
5. **CPU Utilization:** CPU Utilization $= \frac{\text{Total CPU time}}{\text{Total time}} \times 100\%$
6. **Average Waiting Time:** Average Waiting Time $= \frac{\sum(\text{Waiting Time of each process})}{\text{Number of processes}}$
7. **Average Turnaround Time:** Average Turnaround Time $= \frac{\sum(\text{Turnaround Time of each process})}{\text{Number of processes}}$
8. **Average Response Time:** Average Response Time $= \frac{\sum(\text{Response Time of each process})}{\text{Number of processes}}$

Memory Management

1. **Effective Access Time (EAT):**
   (a). $EAT = (1 - p) \times$ Memory Access Time $+ p \times$ Page Fault Rate $\times$ Page Fault Service Time
   (b). $EAT =$ Hit Ratio$\times$Memory Access Time (Cache)$+$Miss Ratio$\times$Memory Access Time (Main Memory)
2. **Degree of Multiprogramming:** $Degree\ of\ Multiprogramming = \frac{\text{Number of Processes in Main Memory}}{\text{Total Number of Processes}} \times 100\%$
3. **Memory Access Time:** $Memory\ Access\ Time =$ Memory Access Time (RAM)+Memory Transfer Time (if applicabl
4. **Memory Cycle Time:** $Memory\ Cycle\ Time =$ Memory Access Time + Time to complete one cycle
5. **Page Table Size:** $Page\ Table\ Size = \frac{\text{Size of Logical Address Space}}{\text{Page Size}}$
6. **Internal Fragmentation:** $Internal\ Fragmentation =$ Partition Size − Process Size
7. **External Fragmentation:** $External\ Fragmentation =$ Total Free Memory − Largest Free Block
8. **Page Fault Rate (PFR):** $Page\ Fault\ Rate = \frac{\text{Number of Page Faults}}{\text{Number of Memory Accesses}}$
9. **Memory Mapping Overhead:**
   $Memory\ Mapping\ Overhead =$ Size of Logical Address Space − Size of Physical Address Space

File Systems

1. **Disk Access Time:** $Disk\ Access\ Time = Seek\ Time + Rotational\ Latency + Transfer\ Time$
2. **File Allocation Table (FAT):** A table that maps file clusters to disk blocks.
3. **File Organization:** Records per Block $= \frac{\text{Block Size}}{\text{Record Size}}$

4. **Disk Space Efficiency:** Disk Space Efficiency $= \frac{\text{Allocated Disk Space}}{\text{Total Disk Space}} \times 100\%$
5. **File Storage Efficiency:** File Storage Efficiency $= \frac{\text{Used File Space}}{\text{Total File Space}} \times 100\%$
6. **File Organization Overhead:** File Organization Overhead $=$ Total Space Occupied by File Organization
7. **File Block Address Calculation:** Block Address $=$ Starting Address of File $+$ (Logical Block Number $\times$ Block Size)
8. **File Compression Ratio:** Compression Ratio $= \frac{\text{Original File Size}}{\text{Compressed File Size}}$
9. **File Density:** File Density $= \frac{\text{Actual Data Size}}{\text{Allocated File Space}}$
10. **File Allocation Table (FAT) Size:** FAT Size $= \frac{\text{Size of Disk}}{\text{Size of a FAT Entry}}$
11. **Disk I/O Time:** Disk I/O Time $=$ Seek Time $+$ Rotational Delay $+$ Transfer Time $+$ Additional Overhead

1.17 Frequently Asked Interview Questions

1. Define a file allocation table (FAT).
2. Define a zombie process.
3. Define cache memory and its significance.
4. Define CPU scheduling.
5. Define distributed shared memory (DSM) and its advantages.
6. Define distributed shared memory (DSM).
7. Define fork and exec system calls.
8. Define interrupt and trap.
9. Define multi-programming and multi-tasking.
10. Define paging and segmentation.
11. Define preemptive and non-preemptive scheduling.
12. Define process migration.
13. Define RAID and its levels.
14. Define real-time operating system (RTOS).
15. Define symmetric multiprocessing (SMP).
16. Define the terms big-endian and little-endian in computer architecture.
17. Define the terms cold start and warm start in system booting.
18. Define the terms cold start and warm start.
19. Define the terms context switch overhead and blocking time.
20. Define the terms contiguous and non-contiguous memory allocation.
21. Define the terms CPU burst and I/O burst in the CPU scheduling context.
22. Define the terms CPU burst and I/O burst.
23. Define the terms critical section and semaphore in process synchronization.
24. Define the terms critical section and semaphore.
25. Define the terms deadlock and livelock.
26. Define the terms deadlock detection and recovery.
27. Define the terms deadlock prevention and deadlock avoidance.
28. Define the terms demand-paging and pre-paging in virtual memory.
29. Define the terms demand-paging and pre-paging.
30. Define the terms dirty bit and valid bit in page tables.
31. Define the terms distributed deadlock and global deadlock.
32. Define the terms distributed system and decentralized system.
33. Define the terms dynamic linking and static linking in the context of libraries.
34. Define the terms fork and join in parallel programming.
35. Define the terms global variable and local variable in the context of processes.
36. Define the terms hard and soft real-time systems.
37. Define the terms I/O-bound and CPU-bound processes in resource management.
38. Define the terms I/O-bound and CPU-bound processes.
39. Define the terms integrity and confidentiality in the context of security.
40. Define the terms inter-process communication (IPC) and intra-process communication.
41. Define the terms job control language (JCL) and batch processing.
42. Define the terms job queue and ready queue in CPU scheduling.
43. Define the terms logical address and physical address in memory management.
44. Define the terms logical address space and physical address space.

45. Define the terms logical clock and physical clock in distributed systems.
46. Define the terms mutual exclusion and race condition.
47. Define the terms NUMA (Non-Uniform Memory Access) and UMA (Uniform Memory Access).
48. Define the terms parallel processing and distributed processing.
49. Define the terms preemptive and non-preemptive kernel.
50. Define the terms preemptive and non-preemptive multitasking.
51. Define the terms preemptive and non-preemptive scheduling.
52. Define the terms priority inversion and priority inheritance in scheduling.
53. Define the terms process group and session in process management.
54. Define the terms process group and session.
55. Define the terms process priority and process scheduling priority.
56. Define the terms process spawning and process termination in process management.
57. Define the terms process spawning and process termination.
58. Define the terms process synchronization and interprocess communication (IPC).
59. Define the terms resource allocation graph and deadlock cycle.
60. Define the terms response time and turnaround time.
61. Define the terms semaphores and mutex.
62. Define the terms spin lock and mutex.
63. Define the terms starvation and aging in process scheduling.
64. Define the terms starvation and aging in scheduling.
65. Define the terms strong and weak consistency.
66. Define the terms strong consistency and weak consistency in distributed systems.
67. Define the terms superblock and inode in file systems.
68. Define the terms symmetric multiprocessing (SMP) and asymmetric multiprocessing.
69. Define the terms symmetrical multiprocessing (SMP) and asymmetrical multiprocessing.
70. Define the terms system image backup and incremental backup.
71. Define the terms task and process in an operating system.
72. Define the terms throughput and bandwidth.
73. Define the terms throughput and latency.
74. Define the terms throughput and turnaround time.
75. Define the terms time-sharing and space-sharing in resource allocation.
76. Define the terms token ring and Ethernet in the context of networking.
77. Define the terms transparent and explicit file access.
78. Define the terms user mode and kernel mode in CPU operation.
79. Define the terms voluntary and involuntary context switch in scheduling.
80. Define the terms voluntary and involuntary context switch.
81. Define the terms weak and strong consistency in distributed systems.
82. Define thrashing and its effects on system performance.
83. Define thread synchronization.
84. Define virtualization.
85. Differentiate between a process and a program.
86. Differentiate between internal and external fragmentation.
87. Differentiate between process and thread.
88. Differentiate between user-level threads and kernel-level threads.
89. Explain the concept of a command interpreter or shell.
90. Explain the concept of a condition variable in process synchronization.

91. Explain the concept of a critical section in process synchronization.
92. Explain the concept of a critical section.
93. Explain the concept of a data race in multithreading.
94. Explain the concept of a deadlock detection algorithm.
95. Explain the concept of a dirty bit in the page table.
96. Explain the concept of a dirty page in virtual memory.
97. Explain the concept of a fault-tolerant system and its characteristics.
98. Explain the concept of a fault-tolerant system.
99. Explain the concept of a file descriptor in file management.
100. Explain the concept of a file system hierarchy.
101. Explain the concept of a file system journaling and its advantages.
102. Explain the concept of a fork system call.
103. Explain the concept of a job queue.
104. Explain the concept of a kernel panic and its implications.
105. Explain the concept of a kernel panic.
106. Explain the concept of a kernel thread.
107. Explain the concept of a logical address and a physical address.
108. Explain the concept of a monitor.
109. Explain the concept of a multi-level feedback queue in CPU scheduling.
110. Explain the concept of a multithreaded kernel and its benefits.
111. Explain the concept of a multithreaded kernel.
112. Explain the concept of a page frame.
113. Explain the concept of a process group.
114. Explain the concept of a process pool and its applications.
115. Explain the concept of a process pool.
116. Explain the concept of a process state.
117. Explain the concept of a process table and its entries.
118. Explain the concept of a process table and its structure.
119. Explain the concept of a process table.
120. Explain the concept of a race condition in concurrent programming.
121. Explain the concept of a race condition.
122. Explain the concept of a real-time clock in operating systems.
123. Explain the concept of a real-time operating system (RTOS).
124. Explain the concept of a reentrant function.
125. Explain the concept of a shadow page table.
126. Explain the concept of a spin lock.
127. Explain the concept of a superuser or root user.
128. Explain the concept of a system call table.
129. Explain the concept of a system call wrapper.
130. Explain the concept of a system V IPC (Inter-Process Communication) mechanism.
131. Explain the concept of a trap in interrupt handling.
132. Explain the concept of a virtual file system.
133. Explain the concept of a watchdog timer in real-time systems.
134. Explain the concept of a working set in process management.
135. Explain the concept of CPU affinity.
136. Explain the concept of deadlock avoidance.

137. Explain the concept of demand paging.
138. Explain the concept of dynamic loading in operating systems.
139. Explain the concept of mutual exclusion in process synchronization.
140. Explain the concept of mutual exclusion.
141. Explain the concept of process migration and its applications.
142. Explain the concept of process priority inversion and its resolution.
143. Explain the concept of process priority inversion.
144. Explain the concept of spooling.
145. Explain the concept of thread safety.
146. Explain the concept of virtual memory.
147. Explain the difference between a monolithic kernel and a microkernel.
148. Explain the purpose of a bootloader.
149. Explain the purpose of a loadable kernel module.
150. Explain the purpose of a memory barrier in multithreading.
151. Explain the purpose of a page replacement algorithm in virtual memory.
152. Explain the purpose of a page table in virtual memory management.
153. Explain the purpose of a PCB (Process Control Block).
154. Explain the purpose of a process group in process management.
155. Explain the purpose of a process group.
156. Explain the purpose of a process identifier (PID).
157. Explain the purpose of a process state diagram.
158. Explain the purpose of a root file system.
159. Explain the purpose of a spin lock in synchronization.
160. Explain the purpose of a system call interface.
161. Explain the purpose of a system call.
162. Explain the purpose of a system V IPC (Inter-Process Communication) mechanism.
163. Explain the purpose of a thread-local storage (TLS).
164. Explain the purpose of a thread-safe data structure.
165. Explain the purpose of a thread-safe function.
166. Explain the purpose of an operating system.
167. Explain the purpose of CPU affinity and its impact on system performance.
168. Explain the purpose of the FAT (File Allocation Table).
169. Explain the purpose of the FAT32 file system.
170. Explain the role of a barrier synchronization in parallel computing.
171. Explain the role of a process scheduler.
172. Explain the role of a thread pool in multithreading.
173. Explain the role of a watchdog timer in real-time operating systems.
174. Explain the role of an interrupt vector in interrupt handling.
175. Explain the role of the bootloader.
176. Explain the role of the file system.
177. Explain the role of the master boot record (MBR).
178. What is a barrier synchronization in parallel computing?
179. What is a command interpreter or shell?
180. What is a command-line interface (CLI)?
181. What is a context switch cost, and how is it measured?
182. What is a context switch?

183. What is a daemon process?
184. What is a deadlock, and how can it be prevented?
185. What is a distributed file system, and how does it differ from a centralized file system?
186. What is a distributed file system, and how does it differ from a traditional file system?
187. What is a distributed lock manager, and how does it handle distributed locks?
188. What is a distributed lock manager, and why is it needed?
189. What is a distributed operating system?
190. What is a distributed shared memory (DSM) system?
191. What is a file descriptor?
192. What is a file system journaling, and why is it important?
193. What is a kernel?
194. What is a memory hierarchy, and how is it implemented in modern systems?
195. What is a memory-mapped file, and how is it used in operating systems?
196. What is a memory-mapped file, and how is it utilized in operating systems?
197. What is a message-passing system in the context of distributed operating systems?
198. What is a page fault handler, and how does it work?
199. What is a page fault?
200. What is a page table?
201. What is a priority inversion, and how can it be resolved?
202. What is a process control block (PCB)?
203. What is a process synchronization?
204. What is a process tree, and how is it structured?
205. What is a process tree, and how is it used in process management?
206. What is a race condition in a concurrent system?
207. What is a segmentation fault?
208. What is a semaphore and its types?
209. What is a semaphore?
210. What is a shadow page table, and how does it relate to virtual memory?
211. What is a shell in the context of an operating system?
212. What is a shell script, and how is it used in operating systems?
213. What is a soft link and a hard link in a file system?
214. What is a system call?
215. What is a system image and how is it used in disaster recovery?
216. What is a system image and its importance?
217. What is a system image backup, and how is it different from a regular backup?
218. What is a system image?
219. What is a thread pool?
220. What is a trap handler, and how does it handle traps in an operating system?
221. What is a trap handler?
222. What is a zombie process, and how does it occur?
223. What is a zombie process?
224. What is an operating system?
225. What is process migration, and in what scenarios is it useful?
226. What is the difference between a process and a lightweight process?
227. What is the difference between static linking and dynamic linking?
228. What is the difference between symmetric and asymmetric multiprocessing?

229. What is the purpose of a buffer cache?
230. What is the purpose of a cache coherency protocol in multiprocessor systems?
231. What is the purpose of a CPU affinity mask in multiprocessing systems?
232. What is the purpose of a CPU affinity mask?
233. What is the purpose of a device driver?
234. What is the purpose of a Gantt chart in CPU scheduling?
235. What is the purpose of a loadable kernel module in a modular kernel?
236. What is the purpose of a page fault handler in virtual memory management?
237. What is the purpose of a page fault in virtual memory management?
238. What is the purpose of a page table entry in virtual memory systems?
239. What is the purpose of a pipe in interprocess communication?
240. What is the purpose of a process control block (PCB) in process management?
241. What is the purpose of a system clock in timekeeping?
242. What is the purpose of a system image in backup and recovery?
243. What is the purpose of a system image?
244. What is the purpose of a thread-safe data structure in multithreading?
245. What is the purpose of a TLB (Translation Lookaside Buffer) in virtual memory?
246. What is the purpose of a TLB (Translation Lookaside Buffer)?
247. What is the purpose of a trap in interrupt handling?
248. What is the purpose of the C-LOOK scheduling algorithm for disk drives?
249. What is the purpose of the FAT16 file system?
250. What is the purpose of the LRU (Least Recently Used) algorithm in page replacement?
251. What is the purpose of the process control block (PCB) in a process?
252. What is the purpose of the swap space?
253. What is the role of a clock algorithm in page replacement?
254. What is the role of a deadlock detection algorithm in a distributed system?
255. What is the role of a deadlock detection algorithm in distributed systems?
256. What is the role of a process scheduler in a multiprogramming environment?
257. What is the role of a system administrator in managing an operating system?
258. What is the role of a system call handler in an operating system?
259. What is the role of an interrupt handler?
260. What is the role of an interrupt vector table in interrupt handling?
261. What is the role of the file allocation table (FAT) in file systems?
262. What is the role of the Memory Management Unit (MMU) in virtual memory?
263. What is the role of the MMU (Memory Management Unit) in a computer system?
264. What is the role of the page replacement algorithm?
265. What is the role of the process scheduler in a real-time operating system?
266. What is the role of the root directory in a file system hierarchy?
267. What is the significance of a fork bomb in the context of system security?
268. What is the significance of a Gantt chart in CPU scheduling?
269. What is the significance of the boot sector in the boot process?
270. What is the significance of the Least Recently Used (LRU) algorithm in page replacement?
271. What is the significance of the Master Boot Record (MBR) in the boot process?
272. What is the significance of the root directory in a file system?
273. What is the working set model?
274. What is the working set of a process?

1.18 Worksheets

Worksheet 1

Multiple Choice Questions

1. Which of the following statements about the kernel in an operating system is true?
   A. The kernel is responsible for managing user interfaces and applications.
   B. The kernel is a separate program that runs only when the user interacts with the system.
   C. The kernel is the core component that manages hardware resources and provides essential services.
   D. The kernel is primarily responsible for handling user-level processes and multitasking.
2. What is the primary purpose of an operating system?
   A. Managing hardware resources
   B. Running user applications
   C. Providing a graphical user interface
   D. All of the above
3. Which of the following is not a function of the operating system?
   A. Process management
   B. File management
   C. Application development
   D. Memory management
4. What does a system call provide in an operating system?
   A. User interface
   B. A way for programs to request services from the operating system
   C. Hardware resources
   D. File management
5. Which component of the operating system handles communication between hardware and software?
   A. Scheduler
   B. Kernel
   C. File Manager
   D. Shell
6. What is the significance of the bootstrap program in the boot process of an operating system?
   A. Manages user interfaces
   B. Loads the kernel into memory
   C. Allocates memory to applications
   D. Handles peripheral devices
7. What does the fork system call do in an operating system?
   A. Allocates memory for a new process
   B. Creates a new process by duplicating the calling process
   C. Terminates the calling process
   D. Reads data from a file
8. What is the return value of the fork system call in the parent process?
   A. -1
   B. 0
   C. Process ID (PID) of the child process
   D. Process ID (PID) of the parent process

9. In the context of operating systems, which statement accurately describes a characteristic of microkernels?
   A. Microkernels generally have a larger kernel size compared to monolithic kernels.
   B. Microkernels move most of the operating system services into kernel space.
   C. Microkernels provide higher performance due to reduced inter-process communication.
   D. Microkernels emphasize minimalism, with essential services implemented as user-level processes.
10. What is the primary goal of multiprogramming in operating systems?
   A. To improve the performance of a single program
   B. To execute multiple programs concurrently for better CPU utilization
   C. To simplify the user interface
   D. To reduce the size of the operating system

Subjective Questions

1. Compare and contrast the characteristics of real-time operating systems (RTOS) and general-purpose operating systems.

2. Discuss the differences between microkernels and monolithic kernels. Evaluate the strengths and weaknesses of each architecture and provide examples of operating systems that use each type of kernel.

3. Describe the booting process of an operating system. Include the role of the bootloader and the initialization of the kernel.

4. Explain the role of device drivers in an operating system and how they facilitate communication between software and hardware.

5. Describe the purpose and functionality of system calls in an operating system. Provide examples of common system calls.

6. Explain the concept of interrupts in the context of computer systems.

. . . . . .
. . . . . .
. . . . . .
. . . . . .

Worksheet 2

Multiple Choice Questions

1: What is a process in the context of operating systems?
   A. A program in execution
   B. A system utility
   C. A file stored on the hard disk
   D. An input device
2: What is the purpose of a process control block (PCB)?
   A. To store the program's source code
   B. To manage file I/O operations
   C. To store information about a process
   D. To allocate memory to a process
3: What is CPU scheduling in an operating system?
   A. Allocating memory to processes
   B. Assigning tasks to peripheral devices
   C. Determining the order in which processes are executed by the CPU
   D. Managing file systems
4: Which scheduling algorithm aims to minimize the turnaround time?
   A. First-Come-First-Serve (FCFS)
   B. Shortest Job Next (SJN)
   C. Round Robin (RR)
   D. Priority Scheduling
5: What is a thread in the context of multitasking?
   A. A process in execution
   B. A lightweight process sharing the same address space
   C. A file in use by the operating system
   D. A system utility for file management
6: In Round Robin CPU scheduling, what does the term "time quantum" refer to?
   A. The total time required to complete a process.
   B. The amount of time a process is allowed to run in one continuous time slot.
   C. The time taken by the CPU to switch between processes.
   D. The priority assigned to each process in the ready queue.
7: Which of the following statements accurately distinguishes between user-level threads (ULTs) and kernel-level threads (KLTs)?
   A. User-level threads are managed by the operating system kernel, while kernel-level threads are managed by user-level libraries.
   B. User-level threads are more efficient in terms of context switching compared to kernel-level threads.
   C. Kernel-level threads are visible to the operating system scheduler, allowing for better utilization of multiple processors.
   D. User-level threads provide stronger isolation between threads, preventing interference with each other.
8: Which of the following is a characteristic of user-level threads (ULTs)?
   A. Better responsiveness to system events.
   B. Lower context-switching overhead.
   C. Directly visible to the operating system scheduler.

D. Kernel support is required for their management.

9: In a system with user-level threads, if one thread in a process is blocked, what happens to the other threads in the same process?

A. All threads in the process are blocked.
B. Other threads continue executing independently.
C. The process is terminated.
D. A system interrupt is triggered.

10: Consider a system with three processes (P1, P2, and P3) scheduled using the First-Come-First-Serve (FCFS) scheduling algorithm. The arrival time and burst time for each process are as follows:

Process	Arrival Time	Burst Time
$P1$	0	6
$P2$	2	4
$P3$	4	8

If the waiting time is defined as the total time a process spends waiting in the ready queue, what is the waiting time for process P2?

A. 4 units of time
B. 5 units of time
C. 7 units of time
D. 9 units of time

Subjective Questions

1. Explain the concept of a process in operating systems. Highlight the key components of a process and the role they play in program execution.

2. Describe the life cycle of a process. Discuss the transitions between different states and the events triggering these transitions.

3. Compare and contrast preemptive and non-preemptive CPU scheduling algorithms. Provide examples of scenarios where each type of algorithm is beneficial.

4. Define the term ”thread” in the context of multitasking. Explain the advantages of using threads over processes and how they contribute to parallelism.

5. Explain the difference between user-level threads (ULTs) and kernel-level threads (KLTs). Discuss the advantages and disadvantages of each type and scenarios where they are most suitable.

6. Consider a system with three processes scheduled using the Round Robin (RR) scheduling algorithm. The time quantum is set to 4 milliseconds. The arrival time and burst time for each process are as follows:

Process	Arrival Time	Burst Time
$P1$	0	8
$P2$	2	5
$P3$	4	6

If the processes follow the FCFS order when multiple processes have the same remaining time, what is the turnaround time of process P2?

Worksheet 3

Multiple Choice Questions

1: What is the primary goal of synchronization in operating systems?
   A. Minimizing memory usage
   B. Ensuring fair CPU scheduling
   C. Coordinating the execution of multiple processes
   D. Enhancing disk I/O performance
2: In synchronization, what does the term ”race condition” refer to?
   A. A competition between processes
   B. A condition of deadlock
   C. Undesirable interference between concurrent operations
   D. A priority inversion scenario
3: What is the critical section problem in concurrent programming?
   A. Ensuring all processes run concurrently
   B. Managing access to shared resources
   C. Coordinating process termination
   D. Balancing system load
4: Which condition must be satisfied for a solution to the critical section problem to be effective?
   A. Mutual exclusion
   B. Starvation
   C. Deadlock
   D. Priority inversion
5: What is a deadlock in the context of operating systems?
   A. Simultaneous execution of multiple processes
   B. Inability to acquire necessary resources and proceed
   C. Efficient scheduling of processes
   D. Fair distribution of CPU time
6: Which of the following is a classic synchronization problem that involves two processes sharing a single, finite-sized buffer?
   A. Readers-Writers Problem
   B. Dining Philosophers Problem
   C. Producer-Consumer Problem
   D. Banker’s Algorithm
7: What is the purpose of a semaphore in process synchronization?
   A. Identify the priority of a process
   B. Ensure mutual exclusion among processes
   C. Schedule processes based on their arrival time
   D. Allocate memory to processes
8: In Banker’s algorithm, what information does the ”maximum need matrix” represent?
   A. The maximum number of resources that each process may request.
   B. The current allocation of resources to each process.
   C. The total available resources in the system.
   D. The resources released by each process.
9: What does Banker’s algorithm consider when deciding to grant or deny a resource request?

A. Only the maximum need of the process.
B. Only the available resources in the system.
C. Both the maximum need and available resources.
D. Only the current allocation of the process.

10: In an operating system, which of the following represents a valid sequence of process state transitions?
A. Ready → Running → Blocked → Ready → Terminated
B. Blocked → Terminated → Running → Ready → Blocked
C. Running → Ready → Terminated → Blocked → Ready
D. Ready → Blocked → Running → Terminated → Ready

Subjective Questions

1. Explain the concept of synchronization in operating systems. Why is it essential for concurrent program execution, and what challenges does it address?

2. Define the critical section problem and explain why it is a fundamental concern in concurrent programming. Describe the requirements that a solution to the critical section problem must satisfy.

3. Define deadlock in the context of operating systems. Discuss the necessary conditions for deadlock occurrence and how they contribute to the formation of a deadlock.

4. Explain the difference between deadlock prevention and deadlock avoidance strategies.

5. A system has three processes with the following burst times (in milliseconds) and priorities:

Process	Burst Time	Priority
P1	6	3
P2	4	1
P3	8	2

   Assuming Priority scheduling algorithm, calculate the average waiting time and average turnaround time.

6. Consider a system with five processes ($P1, P2, P3, P4$, and $P5$) and three resource types (A, B, and C). The current allocation matrix, maximum demand matrix, and available matrix are given as follows:

Current Allocation Matrix:

	A	B	C
$P1$	1	2	2
$P2$	3	1	3
$P3$	1	3	5
$P4$	4	2	2
$P5$	2	4	2

Maximum Demand Matrix:

	A	B	C
$P1$	3	4	3
$P2$	6	2	5
$P3$	3	6	8
$P4$	7	4	7
$P5$	5	8	3

Available Matrix:

A	B	C
2	1	3

Using Banker's algorithm, determine if the system is in a safe state. If it is, provide a safe sequence; otherwise, explain why the system is not safe.

Worksheet 4

Multiple Choice Questions

1: Which of the following is a type of volatile memory?
   A. ROM
   B. RAM
   C. SSD
   D. Hard Disk

2: What is the purpose of a page table in a paging memory management scheme?
   A. To store page faults
   B. To translate virtual addresses to physical addresses
   C. To manage cache memory
   D. To allocate memory to processes

3: Which file system is commonly used in Linux operating systems?
   A. NTFS
   B. FAT32
   C. ext4
   D. HFS+

4: Which of the following is a characteristic of SSD (Solid State Drive) as compared to traditional HDD (Hard Disk Drive)?
   A. Magnetic storage
   B. Mechanical components
   C. Slower access times
   D. No moving parts

5: Which disk scheduling algorithm uses a queue to organize pending requests and serves them in the order they are received?
   A. FCFS (First-Come-First-Serve)
   B. SSTF (Shortest Seek Time First)
   C. C-SCAN (Circular SCAN)
   D. LOOK

6: What is thrashing in the context of memory management?
   A. Excessive page faults leading to degraded performance
   B. Efficient use of virtual memory
   C. Rapid data transfer between RAM and cache
   D. Successful page replacement

7: Which disk scheduling algorithm aims to minimize the total movement of the disk arm by choosing the request that is closest to the current arm position?
   A. First-Come-First-Serve (FCFS)
   B. Shortest Seek Time First (SSTF)
   C. Circular SCAN
   D. LOOK

8: What is the primary advantage of using demand paging in virtual memory systems?
   A. Reduced page faults
   B. Increased RAM capacity
   C. Faster data retrieval

D. Simplicity of implementation

9: Which of the following is a primary purpose of RAID technology?

A. Disk Encryption

B. Improved File Compression

C. Increased Data Redundancy and Fault Tolerance

D. Enhanced Disk Formatting

10: In a paging system, if the logical address space is divided into pages of size 2^{12} bytes and the physical memory is divided into frames of the same size, how many bits are needed for the page number and the offset within the page, respectively, for a 32-bit logical address?

A. 20 bits for page number, 12 bits for offset

B. 10 bits for page number, 22 bits for offset

C. 12 bits for page number, 20 bits for offset

D. 14 bits for page number, 18 bits for offset

Subjective Questions

1. Explain the working principles of the Least Recently Used (LRU) page replacement algorithm. Discuss its advantages and potential drawbacks. Can you suggest scenarios where LRU might perform exceptionally well or poorly?

2. Explain the structure of a disk in detail, including the terms like tracks, sectors, and cylinders. How do these components contribute to efficient data storage and retrieval on a disk?

3. Discuss the advantages and disadvantages of disk partitioning in terms of organization and performance. How

does partitioning contribute to disk management and data security in an operating system?

4. Consider the following scenario:

   Page Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

   Number of Page Frames: 3

   Use the Least Recently Used (LRU) page replacement algorithm to calculate the number of page faults.

   LRU Page Replacement Algorithm:

   - Initially, all page frames are empty.
   - When a page is referenced:
     - If the page is already in a frame, it becomes the most recently used.
     - If the page is not in any frame, a page fault occurs, and the least recently used page is replaced.

   Question:

   Calculate the number of page faults using the LRU page replacement algorithm for the given page reference string and a total of 3 page frames.

5. Provide an overview of what RAID is and describe at least three common RAID levels (e.g., RAID 0, RAID 1, RAID 5).

6. Consider a disk with 100 tracks numbered from 0 to 99. The current position of the disk arm is at track 50. The following disk access requests are given:

$$45, 60, 20, 90, 10$$

Assuming the SSTF (Shortest Seek Time First) disk scheduling algorithm, calculate the total head movement using the current disk arm position.

Solution Steps:

(a). Calculate the absolute seek time for each request by finding the absolute difference between the current position and the requested track.

(b). Select the request with the shortest seek time and move the disk arm to that track.

(c). Repeat steps 1 and 2 until all requests are processed, and calculate the total head movement.

Provide the final total head movement as the answer.

Appendix A Important Algorithms

A.1 First-Come-First-Serve (FCFS) CPU Scheduling

Algorithm 5 First-Come-First-Serve (FCFS) CPU Scheduling

Initialize the queue to store processes.
Insert all processes into the queue in the order they arrive.
Initialize the `current_time` variable to 0.
while the queue is not empty **do**
    Dequeue the process from the front of the queue.
    Set the `start_time` of the process as the maximum of its arrival time and the `current_time`.
    Update the `current_time` to the `end_time` of the dequeued process.
    Execute the process until completion.
Calculate the turnaround time and waiting time for each process:
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.

A.2 Shortest Job Next (SJN) or Shortest Job First (SJF) CPU Scheduling

Algorithm 6 Shortest Job Next (SJN) or Shortest Job First (SJF) CPU Scheduling

Initialize the queue to store processes.
Insert all processes into the queue.
while the queue is not empty **do**
    Sort the queue based on the burst time of each process in ascending order.
    Dequeue the process with the shortest burst time.
    Set the `start_time` of the process as the maximum of its arrival time and the current time.
    Execute the process until completion.
    Update the current time to the end time of the dequeued process.
Calculate the turnaround time and waiting time for each process:
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.

Algorithm 7 Priority Scheduling CPU Scheduling

Initialize the queue to store processes.
Insert all processes into the queue.
while the queue is not empty **do**
    Sort the queue based on the priority of each process in descending order.
    Dequeue the process with the highest priority.
    Set the `start_time` of the process as the maximum of its arrival time and the current time.
    Execute the process until completion.
    Update the current time to the end time of the dequeued process.
Calculate the turnaround time and waiting time for each process:
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.

Algorithm 8 Round Robin (RR) CPU Scheduling

Initialize the queue to store processes.
Insert all processes into the queue.
Initialize the time quantum (slice) for each process.
while the queue is not empty **do**
    Dequeue the process from the front of the queue.
    Set the `start_time` of the process as the maximum of its arrival time and the current time.
    Execute the process for the time quantum.
    Update the current time to the end time of the executed process.
    if the process is not completed **then**
        Enqueue the process back to the end of the queue.
Calculate the turnaround time and waiting time for each process:
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.

Algorithm 9 Multilevel Queue Scheduling

Initialize multiple queues, each representing a different priority level.
while there are processes in any of the queues **do**
    Dequeue the process from the highest priority queue.
    Set the `start_time` of the process as the maximum of its arrival time and the current time.
    Execute the process until completion.
    Update the current time to the end time of the executed process.
    if the process is not completed **then**
        Enqueue the process into the next lower priority queue.
Calculate the turnaround time and waiting time for each process:
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.

Algorithm 10 Multilevel Feedback Queue Scheduling

```
Initialize multiple queues with different priority levels.
Assign a time quantum (slice) to each queue, where lower priority queues have longer time slices.
while there are processes in any of the queues do
    Dequeue the process from the highest priority queue.
    Set the start_time of the process as the maximum of its arrival time and the current time.
    Execute the process until completion or the end of its time quantum.
    Update the current time to the end time of the executed process.
    if the process is not completed then
        if the process used the entire time quantum then
            Enqueue the process into the same or lower priority queue.
        else
            Enqueue the process into the next higher priority queue.
Calculate the turnaround time and waiting time for each process:
     Turnaround Time (TAT) = Completion Time - Arrival Time
     Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.
```

Algorithm 11 Highest Response Ratio Next (HRRN) CPU Scheduling

```
Initialize the queue to store processes.
while the queue is not empty do
    Calculate the response ratio for each process in the queue.
    Sort the queue based on the response ratio in descending order.
    Dequeue the process with the highest response ratio.
    Set the start_time of the process as the maximum of its arrival time and the current time.
    Execute the process until completion.
    Update the current time to the end time of the executed process.
Calculate the turnaround time and waiting time for each process:
     Turnaround Time (TAT) = Completion Time - Arrival Time
     Waiting Time (WT) = Turnaround Time - Burst Time
Calculate the average turnaround time and average waiting time for all processes.
Display the turnaround time, waiting time, and average values for analysis.
```

A.3 Priority Scheduling CPU Scheduling

A.4 Round Robin CPU Scheduling

A.5 Multilevel Queue Scheduling

A.6 Multilevel Feedback Queue Scheduling

A.7 Highest Response Ratio Next (HRRN) CPU Scheduling

A.8 Lottery Scheduling

Algorithm 12 Lottery Scheduling

1: Initialize a lottery pool containing tickets.
2: Assign a number of tickets to each process based on its priority or other criteria.
3: **while** there are processes in the system **do**
4: Draw a winning ticket randomly from the lottery pool.
5: Select the process associated with the winning ticket for execution.
6: Set the `start_time` of the process as the maximum of its arrival time and the current time.
7: Execute the process until completion.
8: Update the current time to the end time of the executed process.
9: Return the winning ticket to the lottery pool.
10: Calculate the turnaround time and waiting time for each process:
11: Turnaround Time (TAT) = Completion Time - Arrival Time
12: Waiting Time (WT) = Turnaround Time - Burst Time
13: Calculate the average turnaround time and average waiting time for all processes.
14: Display the turnaround time, waiting time, and average values for analysis.