Problem Solver (DI03016061) - Problem Solver

Antigravity AI

June 4, 2026

Problem Solver

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction to Operating System

1.1 Definition and Functions

Methodology:

Core Functions of an Operating System An Operating System (OS) is system software that acts as an intermediary between the computer hardware and the user. To analyze and troubleshoot system behavior we categorize its operations into core functions:

1. **Process Management**: Creation execution and termination of processes. Allocating CPU time to different processes based on scheduling algorithms.
2. **Memory Management**: Keeping track of primary memory. Deciding which process gets memory and when. Allocating and deallocating memory space.
3. **File Management**: Organizing files into directories. Managing creation deletion and access control of files on storage media.
4. **Device Management**: Keeping track of all hardware devices. The I/O controller manages communication via device drivers.
5. **Security and Protection**: Preventing unauthorized access to programs and data using authentication authorization and encryption algorithms.

Worked Example:

Identifying OS Functions in Real World Scenarios **Problem Statement:** Map the following user actions to the corresponding primary operating system function responsible for handling the execution logic:

1. Opening a saved document from the hard drive into RAM.
2. Running multiple applications like a web browser and a media player simultaneously.
3. Connecting a new USB printer to the computer and printing a page.
4. Setting up a user account with a password for system login.

Solution:

1. **Opening a saved document**: This involves reading binary data from secondary storage organizing it as a file object and allocating RAM space. Primary Functions: **File Management** and **Memory Management**.
2. **Running multiple applications**: This requires rapidly switching CPU allocation between active execution threads. Primary Function: **Process Management**.
3. **Connecting a USB printer**: The OS must recognize hardware interrupts and load the appropriate instruction sets (drivers) to interact with the device. Primary Function: **Device Management**.

4. **Setting up a user account**: This restricts system access and validates cryptographic credentials. Primary Function: **Security and Protection**.

1.2 Evolution of Operating Systems

Methodology:

Evolutionary Stages of Operating Systems The computational paradigm of operating systems has evolved through distinct technological stages:

1. **No Operating System (1940s)**: Programs were entered directly into the machine hardware via physical switches. Execution was entirely manual.
2. **Batch Processing Systems (1950s)**: Jobs with similar resource requirements were grouped (batched) and executed sequentially without user interaction to reduce setup time.
3. **Multiprogramming Systems (1960s)**: Multiple active jobs are kept in main memory simultaneously. When one job waits for I/O the CPU context-switches to another to maximize CPU utilization.
4. **Time-Sharing Systems (1970s)**: CPU time is divided into quantums (time slices) shared rapidly among multiple interactive users concurrently.
5. **Personal Computer Systems (1980s onwards)**: OS designed for single users focusing on graphical user interfaces and responsive peripheral interaction.

Worked Example:

Determining the Generational OS Concept **Problem Statement:** Analyze the following historical processing scenarios and identify the evolutionary OS concept being implemented:

1. A programmer leaves a deck of punched cards with a human operator. The operator groups these cards with other FORTRAN programs to be compiled and run sequentially overnight.
2. A university mainframe computer allows 20 different students to interact with it simultaneously via serial terminals giving each student the illusion of a dedicated machine.

Solution:

1. **Scenario 1:** The algorithm groups similar jobs together to be executed sequentially without user interaction. This is the defining characteristic of **Batch Processing Systems**.
2. **Scenario 2:** Multiple users interact concurrently with a single processing unit by dividing the CPU time into discrete slices. This core scheduling mechanism is **Time-Sharing Systems**.

1.3 Types of Operating Systems

Methodology:

Characteristics of Various OS Types

1. **Batch OS**: Executes predefined sequences of commands without user intervention.
2. **Time-Sharing OS**: Distributes CPU time among multiple interactive terminals. Emphasizes low response time.
3. **Multiprogramming OS**: Maximizes CPU utilization by overlapping CPU and I/O operations of multiple programs.

4. **Multiprocessing OS**: Utilizes multiple physical CPUs within a single tightly coupled system to execute threads in parallel.
5. **Multitasking OS**: A logical extension of multiprogramming where CPU switching is fast enough to allow user interaction with multiple running programs.
6. **Real-Time OS**: Processing must complete within strictly defined time constraints (deadlines). Used in deterministic embedded systems.
7. **Network OS**: Runs on a server to manage network traffic facilitate file sharing and enforce network-wide security policies.

Worked Example:

Selecting the Appropriate OS Architecture **Problem Statement:** For each of the following computational scenarios recommend the most suitable type of operating system architecture and provide the technical justification:

1. The flight control computer of a commercial airliner.
2. A corporate file server managing access permissions for 500 employee workstations.
3. A scientific supercomputer calculating fluid dynamics simulations utilizing 1000 cores.
4. A banking system calculating monthly interest for millions of accounts overnight.

Solution:

1. **Flight control computer**: Requires immediate deterministic execution. Missing a processing deadline can cause catastrophic physical failure. **Selection: Real-Time OS**.
2. **Corporate file server**: Requires robust management of shared resources directory protocols and remote authentication over a LAN. **Selection: Network OS**.
3. **Fluid dynamics simulation**: Requires massive parallel computational power distributed across multiple processing units. **Selection: Multiprocessing OS**.
4. **Interest calculation**: Requires executing a massive volume of highly similar data-processing tasks without interactive user input. **Selection: Batch OS**.

1.4 Operating System Services

Methodology:

Analyzing System Services The operating system abstracts hardware complexity by providing discrete services:

1. **Program Execution**: Memory allocation instruction loading and CPU scheduling.
2. **I/O Operations**: Standardized interfaces for reading/writing to varied peripheral devices.
3. **File-System Manipulation**: Low-level disk block management for logical file operations (create read write delete).
4. **Communications**: Inter-process communication (IPC) via shared memory or message passing.
5. **Error Detection**: Hardware polling parity checking and exception handling logic.
6. **Resource Allocation**: Dynamic distribution of CPU cycles RAM and bus bandwidth among competing processes.

Worked Example:

Tracing Service Interactions in Application Execution **Problem Statement:** A background script calculates a dataset and writes the output ‘data.csv‘ to an external hard drive. Trace the logical sequence of OS services invoked to complete this operation.
Solution:

1. **Step 1 (Program Execution)**: The OS loads the script instructions into RAM and assigns CPU cycles to perform the dataset calculations.
2. **Step 2 (Resource Allocation)**: As the dataset grows the OS allocates additional heap memory to the active process.
3. **Step 3 (File-System Manipulation)**: The process requests to create ‘data.csv‘. The OS updates the Master File Table on the external drive.
4. **Step 4 (I/O Operations)**: The OS translates the logical file write command into physical sector write commands sent to the hard drive controller.
5. **Step 5 (Error Detection)**: The OS verifies the write operation using checksums to ensure data integrity during the transfer.

1.5 System Components

Methodology:

OS Architecture Components The logical architecture of an operating system separates user applications from hardware via three distinct layers:

1. **Kernel**: The core foundational program residing continuously in memory. It operates in privileged mode executing the lowest-level functions like process scheduling memory paging and hardware interrupts.
2. **Shell**: The command interpreter interface. It runs in user mode accepting user commands parsing them and invoking the necessary internal programs.
3. **System Calls**: The programmatic interface API used by applications to request services from the kernel. This acts as a controlled gateway shifting the CPU from user mode to kernel mode.

Worked Example:

Tracing a System Call Execution Path **Problem Statement:** Analyze the step-by-step architectural execution path when a developer types the command ‘mkdir newdir‘ in a terminal window.
Solution:

1. **Step 1 (User Input)**: The user types ‘mkdir newdir‘ into the terminal application.
2. **Step 2 (Shell Parsing)**: The **Shell** interprets the text string. It locates the ‘mkdir‘ executable binary and passes ‘newdir‘ as a parameter.
3. **Step 3 (System Call Invocation)**: The ‘mkdir‘ program executes but lacks hardware privileges to modify the disk directly. It executes a **System Call** (e.g. ‘mkdir()‘).
4. **Step 4 (Context Switch)**: A software interrupt is generated. The CPU saves the state of the user application and switches to kernel mode.
5. **Step 5 (Kernel Execution)**: The **Kernel** receives the request validates user permissions allocates disk blocks and creates the directory record on the storage device.
6. **Step 6 (Return and Resume)**: The kernel returns a status code (success/fail) restores the saved state switches the CPU back to user mode and returns control to the shell.

CHAPTER 2

Process Management and Inter Process Communication

Process Management and Inter Process Communication (IPC) are core functions of an Operating System. This chapter focuses on the computational aspects of process scheduling, synchronization using semaphores, and deadlock handling algorithms.

2.1 Process and Threads

A **Process** is an executing instance of a program. A **Thread** is the smallest unit of execution within a process. Multiple threads of the same process share the same code section, data section, and operating system resources, but have separate program counters, registers, and stacks.

2.2 States and Lifecycle

A process goes through various states during its lifecycle:

1. **New:** The process is being created.
2. **Ready:** The process is waiting to be assigned to a processor.
3. **Running:** Instructions are being executed.
4. **Waiting (Blocked):** The process is waiting for some event to occur (e.g., I/O completion).
5. **Terminated:** The process has finished execution.

2.3 Process Control Block (PCB)

The PCB is a data structure maintained by the OS for every process. It contains information such as Process State, Program Counter, CPU Registers, CPU Scheduling Information, Memory-Management Information, Accounting Information, and I/O Status Information.

2.4 CPU Scheduling

2.4.1 Scheduling Criteria and Metrics

CPU scheduling algorithms decide which process in the ready queue is to be allocated the CPU.

Methodology:

Scheduling Metrics Formulas To evaluate a scheduling algorithm, the following times are calculated for each process:

1. **Arrival Time (AT):** Time at which the process arrives in the ready queue.

2. **Burst Time (BT):** Time required by the process for CPU execution.
3. **Completion Time (CT):** Time at which the process completes its execution.
4. **Turnaround Time (TAT):** Total time spent by the process in the system.

$$TAT = CT - AT$$

5. **Waiting Time (WT):** Total time spent by the process waiting in the ready queue.

$$WT = TAT - BT$$

6. **Response Time (RT):** Time from arrival to the first time the process is scheduled.

2.4.2 First Come First Serve (FCFS)

Methodology:

FCFS Scheduling Algorithm FCFS is a non-preemptive scheduling algorithm.

1. Sort the processes based on their Arrival Time (AT).
2. The process that arrives first is allocated the CPU first.
3. The process runs to completion (non-preemptive).
4. Calculate Completion Time (CT) for each process sequentially. If the CPU is idle, the completion time of the next process is its Arrival Time + Burst Time. Otherwise, it is the previous CT + Burst Time.
5. Calculate TAT and WT using standard formulas.

Worked Example:

FCFS Scheduling Problem Consider the following set of processes with given Arrival Times and Burst Times. Calculate the Average Turnaround Time and Average Waiting Time using FCFS scheduling.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	8
P4	3	6

Solution:

1. **Gantt Chart Construction:**
   - At time 0, P1 arrives and gets the CPU. It executes for 5 units. (CT of P1 = 5)
   - At time 5, P2, P3, and P4 have arrived. FCFS selects P2. It executes for 3 units. (CT of P2 = 5 + 3 = 8)
   - At time 8, P3 executes for 8 units. (CT of P3 = 8 + 8 = 16)
   - At time 16, P4 executes for 6 units. (CT of P4 = 16 + 6 = 22)
2. **Calculate TAT and WT:**

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	5	5	$5-0=5$	$5-5=0$
P2	1	3	8	$8-1=7$	$7-3=4$
P3	2	8	16	$16-2=14$	$14-8=6$
P4	3	6	22	$22-3=19$	$19-6=13$

3. **Calculate Averages:** Average TAT $= (5+7+14+19)/4 = 45/4 = 11.25$
   Average WT $= (0+4+6+13)/4 = 23/4 = 5.75$

2.4.3 Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)

Methodology:

Non preemptive SJF Scheduling

1. At each scheduling point, check the ready queue for available processes.
2. Select the process with the shortest Burst Time (BT) among the available processes.
3. If two processes have the same BT, break the tie using FCFS (earliest Arrival Time).
4. The process runs to completion.

Worked Example:

SJF Non preemptive Scheduling Given the following processes, calculate Average TAT and Average WT using Non-preemptive SJF.

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Solution:

1. **Gantt Chart Construction:**
   - Time 0: Only P1 is available. It executes for 7 units. (CT of P1 = 7).
   - Time 7: P2, P3, and P4 are available. Their burst times are 4, 1, and 4. P3 has the shortest BT.
   - P3 executes for 1 unit. (CT of P3 = 7 + 1 = 8).
   - Time 8: P2 and P4 are available. Both have BT = 4. P2 arrived earlier (at 2).
   - P2 executes for 4 units. (CT of P2 = 8 + 4 = 12).
   - Time 12: P4 executes for 4 units. (CT of P4 = 12 + 4 = 16).
2. **Calculate TAT and WT:**

Process	AT	BT	CT	TAT	WT
P1	0	7	7	7	0
P2	2	4	12	10	6
P3	4	1	8	4	3
P4	5	4	16	11	7

3. **Calculate Averages:** Average TAT $= (7+10+4+11)/4 = 32/4 = 8.0$
   Average WT $= (0+6+3+7)/4 = 16/4 = 4.0$

Methodology:

Shortest Remaining Time First Scheduling SRTF is the preemptive version of SJF.

1. At every unit of time (or whenever a new process arrives), check the ready queue.

2. Select the process with the shortest remaining burst time.
3. Execute the selected process. If a new process arrives with a shorter remaining time than the currently executing process, preempt the current process and allocate the CPU to the new process.

Worked Example:

SRTF Scheduling Calculate the Average TAT and Average WT for the following processes using SRTF.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Solution:

1. **Gantt Chart Construction and Preemption Tracking:**
   - Time 0: P1 available. Remaining times: [P1:8]. P1 executes.
   - Time 1: P2 arrives. Remaining times: [P1:7, P2:4]. P2 is shorter. P1 is preempted. P2 executes.
   - Time 2: P3 arrives. Remaining times: [P1:7, P2:3, P3:9]. P2 is shortest. P2 continues.
   - Time 3: P4 arrives. Remaining times: [P1:7, P2:2, P3:9, P4:5]. P2 is shortest. P2 continues.
   - Time 5: P2 finishes. Remaining times: [P1:7, P3:9, P4:5]. P4 is shortest. P4 executes.
   - Time 10: P4 finishes. Remaining times: [P1:7, P3:9]. P1 is shortest. P1 executes.
   - Time 17: P1 finishes. Remaining times: [P3:9]. P3 executes.
   - Time 26: P3 finishes.
2. **Calculate TAT and WT:**

Process	AT	BT	CT	TAT	WT
P1	0	8	17	17	9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

3. **Calculate Averages:** Average TAT $= (17 + 4 + 24 + 7)/4 = 52/4 = 13.0$
   Average WT $= (9 + 0 + 15 + 2)/4 = 26/4 = 6.5$

2.4.4 Round Robin (RR)

Methodology:

Round Robin Scheduling Algorithm

1. Maintain a FIFO Ready Queue of available processes.
2. Define a Time Quantum (TQ).
3. Dequeue the first process from the Ready Queue and allocate the CPU for a duration of TQ.
4. If the process finishes before the TQ expires, it leaves the system, and the next process is dispatched immediately.
5. If the process does not finish within the TQ, it is preempted and placed at the **tail** of the Ready Queue.

6. **Crucial Rule:** If a new process arrives exactly at the same time a currently running process finishes its time quantum, the newly arrived process is added to the ready queue **before** the preempted process.

Worked Example:

Round Robin Scheduling Problem Calculate Average WT and Average TAT using Round Robin scheduling with Time Quantum = 2.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2

Solution:

1. **Execution tracking with Ready Queue (RQ):**
   - **t=0:** P1 arrives. RQ: [P1]. Dequeue P1 to execute for 2 units.
   - **t=1:** P2 arrives. RQ: [P2].
   - **t=2:** P3 arrives. P1 finishes its quantum. RQ becomes [P2, P3, P1]. Dequeue P2 to execute. P1 remaining = 3.
   - **t=3:** P4 arrives. RQ: [P3, P1, P4].
   - **t=4:** P2 finishes its quantum. RQ becomes [P3, P1, P4, P2]. Dequeue P3 to execute. P2 remaining = 1.
   - **t=5:** P3 finishes (BT was 1). RQ: [P1, P4, P2]. Dequeue P1. P3 is complete. (CT of P3 = 5).
   - **t=7:** P1 finishes its quantum. RQ becomes [P4, P2, P1]. Dequeue P4. P1 remaining = 1.
   - **t=9:** P4 finishes (BT was 2). RQ: [P2, P1]. Dequeue P2. P4 is complete. (CT of P4 = 9).
   - **t=10:** P2 finishes (needed 1 unit). RQ: [P1]. Dequeue P1. P2 is complete. (CT of P2 = 10).
   - **t=11:** P1 finishes (needed 1 unit). P1 is complete. (CT of P1 = 11).
2. **Calculate TAT and WT:**

Process	AT	BT	CT	TAT	WT
P1	0	5	11	11	6
P2	1	3	10	9	6
P3	2	1	5	3	2
P4	3	2	9	6	4

3. **Calculate Averages:** Average TAT = $(11 + 9 + 3 + 6)/4 = 29/4 = 7.25$
   Average WT = $(6 + 6 + 2 + 4)/4 = 18/4 = 4.5$

2.5 Inter Process Communication (IPC)

Processes need to communicate and synchronize their actions to ensure data consistency.

- **Race Condition:** A situation where multiple processes access and manipulate shared data concurrently, and the final outcome depends on the order of execution.
- **Critical Section:** A segment of code where a process modifies shared variables, tables, or files. Only one process should be in its critical section at a time.

- **Mutual Exclusion:** If process P is executing in its critical section, then no other processes can be executing in their critical sections.

2.5.1 Semaphores

A semaphore is an integer variable accessed only through two standard atomic operations: `wait()` and `signal()` (originally called P and V operations).

Methodology:

Semaphore Operations **1. Wait operation (P or down):** Decrements the semaphore value. If the value becomes negative, the process gets blocked and is placed in the semaphore's queue.

```
wait(S) {
    S.value--;
    if (S.value < 0) {
        add this process to S.list;
        block();
    }
}
```

2. Signal operation (V or up): Increments the semaphore value. If the value is less than or equal to zero, a blocked process is removed from the queue and awakened.

```
signal(S) {
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.list;
        wakeup(P);
    }
}
```

Worked Example:

Calculating Semaphore Values A counting semaphore was initialized to 10. Then 6 `wait()` operations and 4 `signal()` operations were completed on this semaphore. What is the resulting value of the semaphore?
Solution:

1. Initial value of semaphore $S = 10$.
2. Each `wait()` operation decrements the value by 1.
3. Total decrement $= 6 \times 1 = 6$.
4. Each `signal()` operation increments the value by 1.
5. Total increment $= 4 \times 1 = 4$.
6. Final value of S = Initial Value − Decrements + Increments
7. Final value of $S = 10 - 6 + 4 = 8$.

The resulting value of the semaphore is 8.

2.6 Deadlock

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

2.6.1 Characteristics of Deadlock

Deadlock can arise if four conditions hold simultaneously:

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode.
2. **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources held by other processes.
3. **No Preemption:** Resources cannot be preempted; a resource can be released only voluntarily by the process holding it.
4. **Circular Wait:** A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., and P_n is waiting for a resource held by P_0.

2.6.2 Deadlock Prevention and Avoidance

- **Deadlock Prevention:** Ensuring that at least one of the four necessary conditions for deadlock cannot hold.
- **Deadlock Avoidance:** Requires the operating system to have a priori information about how resources are to be requested. The OS uses this information to dynamically examine the resource-allocation state to ensure that there can never be a circular-wait condition (a safe state).

2.6.3 Bankers Algorithm for Deadlock Avoidance

Methodology:

Bankers Algorithm Let n = number of processes, and m = number of resources types.

1. **Initialization:**
   - Let `Work` and `Finish` be vectors of length m and n, respectively.
   - `Work` = `Available` (current free instances of each resource type).
   - `Finish[i]` = `false` for $i = 0, 1, \ldots, n-1$.
   - Calculate `Need` matrix: `Need[i, j] = Max[i, j] - Allocation[i, j]`
2. **Find an executable process:** Find an index i such that both:
   - `Finish[i] == false`
   - `Need[i]` $\leq$ `Work` (i.e., for all resources j, `Need[i, j]` $\leq$ `Work[j]`)

   If no such i exists, go to Step 4.
3. **Execute process and reclaim resources:**
   - `Work = Work + Allocation[i]`
   - `Finish[i] = true`
   - Go to Step 2.
4. **Check safe state:** If `Finish[i] == true` for all i, then the system is in a **Safe State**. The sequence in which processes were executed is a Safe Sequence. Otherwise, the system is in an Unsafe State.

Worked Example:

Bankers Algorithm Application Consider a system with 5 processes (P_0 to P_4) and 3 resource types (A, B, C). Total instances of $A = 10, B = 5, C = 7$. The current state is given below:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Find the Need matrix and determine if the system is in a safe state.
Solution:

1. **Calculate the Need matrix** (`Need = Max - Allocation`):
   - P0 Need $= (7-0, 5-1, 3-0) = (7, 4, 3)$
   - P1 Need $= (3-2, 2-0, 2-0) = (1, 2, 2)$
   - P2 Need $= (9-3, 0-0, 2-2) = (6, 0, 0)$
   - P3 Need $= (2-2, 2-1, 2-1) = (0, 1, 1)$
   - P4 Need $= (4-0, 3-0, 3-2) = (4, 3, 1)$
2. **Initialize:** `Work` = Available $= (3, 3, 2)$. `Finish` = (F, F, F, F, F).
3. **Find a process to execute:**
   - Can P0 run? Need $(7, 4, 3) \leq$ Work $(3, 3, 2)$? False.
   - Can P1 run? Need $(1, 2, 2) \leq$ Work $(3, 3, 2)$? True.
   - Execute P1. New Work = Work + Allocation of P1 $= (3, 3, 2) + (2, 0, 0) = (5, 3, 2)$. `Finish[1]` = True.
   - Can P3 run? Need $(0, 1, 1) \leq$ Work $(5, 3, 2)$? True.
   - Execute P3. New Work $= (5, 3, 2) + (2, 1, 1) = (7, 4, 3)$. `Finish[3]` = True.
   - Can P4 run? Need $(4, 3, 1) \leq$ Work $(7, 4, 3)$? True.
   - Execute P4. New Work $= (7, 4, 3) + (0, 0, 2) = (7, 4, 5)$. `Finish[4]` = True.
   - Can P0 run? Need $(7, 4, 3) \leq$ Work $(7, 4, 5)$? True.
   - Execute P0. New Work $= (7, 4, 5) + (0, 1, 0) = (7, 5, 5)$. `Finish[0]` = True.
   - Can P2 run? Need $(6, 0, 0) \leq$ Work $(7, 5, 5)$? True.
   - Execute P2. New Work $= (7, 5, 5) + (3, 0, 2) = (10, 5, 7)$. `Finish[2]` = True.
4. All processes are marked `True`. The system is in a **Safe State**. A Safe Sequence is $\langle P_1, P_3, P_4, P_0, P_2 \rangle$.

2.6.4 Deadlock Detection and Recovery

If a system does not use deadlock prevention or avoidance, deadlocks may occur. The system must provide algorithms to examine its state, determine if a deadlock has occurred, and recover from it.

Methodology:

Deadlock Detection Algorithm The algorithm is similar to Banker's Algorithm but operates on current requests rather than maximum possible needs.

1. Initialize `Work = Available`.
2. For $i = 0$ to $n-1$: if `Allocation[i]` $\neq 0$, then `Finish[i] = false`; else `Finish[i] = true`.
3. Find an index i such that:

- `Finish[i] == false`
- `Request[i]` $\leq$ `Work`

If no such i exists, go to Step 5.

4. `Work = Work + Allocation[i]`; `Finish[i] = true`; Go to Step 3.
5. If `Finish[i] == false` for some i, then the system is in a deadlock state. Moreover, if `Finish[i] == false`, then process P_i is deadlocked.

Recovery from Deadlock:

- **Process Termination:** Abort all deadlocked processes, or abort one process at a time until the deadlock cycle is eliminated.
- **Resource Preemption:** Preempt some resources from processes and give them to other processes until the deadlock cycle is broken.

CHAPTER 3

Memory Management

3.1 Process Address Space

Methodology:

Address Translation with Base and Limit Registers Steps for translating a logical address to a physical address using base and limit registers:

1. Identify the logical address generated by the CPU.
2. Retrieve the limit register value for the current process.
3. Compare the logical address with the limit register. If the logical address is greater than or equal to the limit register value then generate an addressing error (trap).
4. If valid then retrieve the base register value.
5. Calculate the physical address by adding the base register value to the logical address:

$$\text{Physical Address} = \text{Logical Address} + \text{Base Register}$$

Worked Example:

Computing Physical Addresses for a Process Consider a process with a base register of 300000 and a limit register of 120000. Calculate the physical addresses for the following logical addresses: 25000 and 150000.
Solution for Logical Address 25000:

1. Logical Address = 25000
2. Limit Register = 120000
3. Check Limit: $25000 < 120000$ (Valid)
4. Base Register = 300000
5. Physical Address = $300000 + 25000 = 325000$

Solution for Logical Address 150000:

1. Logical Address = 150000
2. Limit Register = 120000
3. Check Limit: $150000 < 120000$ (False)
4. Result: Trap to operating system (Illegal Addressing Error).

3.2 Swapping

Methodology:

Calculating Process Swap Time Steps to compute the total time required to swap a process in and out of main memory:

1. Identify the size of the process to be swapped (S) and the transfer rate of the backing store (R).
2. Identify the disk latency or seek time (L) if provided.
3. Calculate the transfer time for one direction:
$$t_{\text{transfer}} = \frac{S}{R}$$
4. Calculate total swap time for one process (swap out and swap in):
$$t_{\text{total\_swap}} = 2 \times (t_{\text{transfer}} + L)$$

Worked Example:

Swap Time Calculation for a Hard Disk A system uses swapping with a hard disk transfer rate of 50 MB per second. The average latency of the disk is 8 ms. Calculate the total time required to swap out a 10 MB process and swap in another 10 MB process.
Solution:

1. Process Size $S = 10$ MB.
2. Transfer Rate $R = 50$ MB/s.
3. Latency $L = 8$ ms $= 0.008$ seconds.
4. Transfer time for one process:
$$t_{\text{transfer}} = \frac{10 \text{ MB}}{50 \text{ MB/s}} = 0.2 \text{ seconds} = 200 \text{ ms}$$
5. Swap out time:
$$t_{\text{out}} = t_{\text{transfer}} + L = 200 \text{ ms} + 8 \text{ ms} = 208 \text{ ms}$$
6. Swap in time:
$$t_{\text{in}} = 200 \text{ ms} + 8 \text{ ms} = 208 \text{ ms}$$
7. Total Swap Time:
$$t_{\text{total}} = 208 \text{ ms} + 208 \text{ ms} = 416 \text{ ms}$$

3.3 Contiguous Memory Allocation

Methodology:

Dynamic Storage Allocation Algorithms When allocating contiguous memory to processes from a list of available free holes (blocks) use the following algorithms:

1. **First-Fit:** Scan the list of memory holes from the beginning. Allocate the process to the first hole that is large enough to contain it.
2. **Best-Fit:** Scan the entire list of memory holes. Allocate the process to the smallest hole that is large enough to contain it. This leaves the smallest possible leftover hole.

3. **Worst-Fit:** Scan the entire list of memory holes. Allocate the process to the largest available hole. This leaves the largest possible leftover hole.

Worked Example:

FirstFit BestFit and WorstFit Allocations Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order). How would the First-Fit, Best-Fit, and Worst-Fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

First-Fit Allocation:

- **212 KB Process:** Needs a hole $\geq$ 212 KB. 100 KB (No). 500 KB (Yes). Placed in 500 KB partition. (Remaining: 288 KB)
- **417 KB Process:** Needs a hole $\geq$ 417 KB. 100 KB (No). 288 KB (No). 200 KB (No). 300 KB (No). 600 KB (Yes). Placed in 600 KB partition. (Remaining: 183 KB)
- **112 KB Process:** Needs a hole $\geq$ 112 KB. 100 KB (No). 288 KB (Yes). Placed in 288 KB partition. (Remaining: 176 KB)
- **426 KB Process:** Needs a hole $\geq$ 426 KB. Remaining holes: 100 KB, 176 KB, 200 KB, 300 KB, 183 KB. None are large enough. Must Wait.

Best-Fit Allocation:

- **212 KB Process:** Smallest hole $\geq$ 212 KB is 300 KB. Placed in 300 KB partition. (Remaining: 88 KB)
- **417 KB Process:** Smallest hole $\geq$ 417 KB is 500 KB. Placed in 500 KB partition. (Remaining: 83 KB)
- **112 KB Process:** Smallest hole $\geq$ 112 KB is 200 KB. Placed in 200 KB partition. (Remaining: 88 KB)
- **426 KB Process:** Smallest hole $\geq$ 426 KB is 600 KB. Placed in 600 KB partition. (Remaining: 174 KB)

All processes are allocated.

Worst-Fit Allocation:

- **212 KB Process:** Largest hole is 600 KB. Placed in 600 KB partition. (Remaining: 388 KB)
- **417 KB Process:** Largest hole is 500 KB. Placed in 500 KB partition. (Remaining: 83 KB)
- **112 KB Process:** Largest hole is 388 KB. Placed in 388 KB partition. (Remaining: 276 KB)
- **426 KB Process:** Remaining holes: 100 KB, 83 KB, 200 KB, 300 KB, 276 KB. None are large enough. Must Wait.

Conclusion: Best-Fit makes the most efficient use of memory for this sequence as it successfully allocates all processes.

3.4 Segmentation

Methodology:

Logical to Physical Address Translation in Segmentation Steps to translate a segmented logical address (s, d) to a physical address:

1. Identify the Segment Number (s) and the Offset (d) from the given logical address.
2. Use the segment number s as an index into the Segment Table.
3. Retrieve the Segment Limit and Segment Base Address for segment s from the table.
4. Compare the offset d with the Segment Limit. If $d \geq$ Segment Limit then generate an addressing error (trap).
5. If d is valid calculate the physical address:

$$\text{Physical Address} = \text{Segment Base Address} + d$$

Worked Example:

Calculating Physical Addresses using a Segment Table Consider the following Segment Table:

Segment	Base	Limit
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Calculate the physical address for the following logical addresses specified as $(Segment, Offset)$: a) $(0, 430)$ b) $(1, 10)$ c) $(2, 500)$ d) $(3, 400)$

Solution:

- **a) Logical Address (0, 430):** Segment $s = 0$, Offset $d = 430$. Limit for segment 0 is 600. Since $430 < 600$, the address is valid. Physical Address = Base + Offset $= 219 + 430 = 649$.
- **b) Logical Address (1, 10):** Segment $s = 1$, Offset $d = 10$. Limit for segment 1 is 14. Since $10 < 14$, the address is valid. Physical Address = Base + Offset $= 2300 + 10 = 2310$.
- **c) Logical Address (2, 500):** Segment $s = 2$, Offset $d = 500$. Limit for segment 2 is 100. Since $500 \geq 100$, the offset exceeds the limit. Result: Trap (Addressing Error).
- **d) Logical Address (3, 400):** Segment $s = 3$, Offset $d = 400$. Limit for segment 3 is 580. Since $400 < 580$, the address is valid. Physical Address = Base + Offset $= 1327 + 400 = 1727$.

3.5 Paging

Methodology:

Memory Geometry Calculations for Paging Formulas for calculating sizes and capacities in a paging system:

1. Number of Pages $= \frac{\text{Logical Address Space Size}}{\text{Page Size}}$
2. Number of Frames $= \frac{\text{Physical Memory Size}}{\text{Page Size}}$
3. If Logical Address Space is 2^m bytes and Page Size is 2^n bytes, then the logical address has m bits.

4. The offset (d) requires n bits.
5. The page number (p) requires $m - n$ bits.
6. Page Table Size = Number of Pages × Entry Size

Worked Example:

Calculating Paging Parameters A computer system has a 32-bit logical address space and 8 KB page size. The physical memory is 4 GB. Calculate the number of bits in the page offset, the number of pages, and the number of frames.

Solution:

1. Logical Address Space = 2^{32} bytes. Total logical address bits $m = 32$.
2. Page Size = 8 KB = 8×1024 bytes = $2^3 \times 2^{10}$ bytes = 2^{13} bytes.
3. Page Offset bits $n = 13$ bits.
4. Number of Pages = $\frac{\text{Logical Address Space}}{\text{Page Size}} = \frac{2^{32}}{2^{13}} = 2^{19} = 524288$ pages.
5. Alternatively, page number bits = $m - n = 32 - 13 = 19$ bits. So, Number of Pages = 2^{19}.
6. Physical Memory Size = 4 GB = 4×1024 MB = $4 \times 1024 \times 1024$ KB = $2^2 \times 2^{30}$ bytes = 2^{32} bytes.
7. Number of Frames = $\frac{\text{Physical Memory Size}}{\text{Page Size}} = \frac{2^{32}}{2^{13}} = 2^{19} = 524288$ frames.

Methodology:

Logical to Physical Address Translation in Paging Steps to translate a logical address to a physical address:

1. Extract the Page Number (p) and Page Offset (d) from the logical address.
   - $p = \lfloor \frac{\text{Logical Address}}{\text{Page Size}} \rfloor$
   - $d = \text{Logical Address} \pmod{\text{Page Size}}$
2. Use the page number p as an index into the Page Table to find the corresponding Frame Number (f).
3. Calculate the physical address using the frame number and the offset:

$$\text{Physical Address} = (f \times \text{Page Size}) + d$$

Worked Example:

Paging Address Translation Consider a system with a page size of 4 KB. The page table for a process is given below:

Page Number	Frame Number
0	5
1	9
2	14
3	2

Calculate the physical address for the logical addresses 5000 and 10000.

Solution for Logical Address 5000:

1. Page Size = 4 KB = 4096 bytes.
2. Calculate Page Number (p): $p = \lfloor \frac{5000}{4096} \rfloor = 1$.

3. Calculate Page Offset (d): $d = 5000 \pmod{4096} = 904$.
4. Look up Page Table: For $p = 1$, Frame Number $f = 9$.
5. Calculate Physical Address: $(f \times \text{Page Size}) + d = (9 \times 4096) + 904 = 36864 + 904 = 37768$.

Solution for Logical Address 10000:

1. Page Size = 4096 bytes.
2. Calculate Page Number (p): $p = \lfloor \frac{10000}{4096} \rfloor = 2$.
3. Calculate Page Offset (d): $d = 10000 \pmod{4096} = 1808$.
4. Look up Page Table: For $p = 2$, Frame Number $f = 14$.
5. Calculate Physical Address: $(14 \times 4096) + 1808 = 57344 + 1808 = 59152$.

CHAPTER 4
File Management

This chapter focuses on the computational aspects of file systems, including block calculations, access methods, disk allocation techniques, and permission management.

4.1 File Concept

A file is a named collection of related information recorded on secondary storage. From a computational perspective, files are divided into logical records, which are then mapped onto physical disk blocks.

Methodology:

Calculating File Size and Block Requirements To determine the number of physical blocks required to store a file with fixed-length records:

1. Determine the logical file size: $S_{logical} = N \times R$, where N is the number of records and R is the size of each record in bytes.
2. Determine the number of records that fit in a single block (blocking factor): $B_f = \lfloor B/R \rfloor$, where B is the block size.
3. Calculate the total number of blocks needed: $K = \lceil N/B_f \rceil$.
4. Calculate the internal fragmentation (wasted space) per block: $F_{internal} = B - (B_f \times R)$.

Worked Example:

Block Calculation for Fixed Records A file contains 2500 records, each 120 bytes long. The disk block size is 512 bytes. Calculate the logical file size, the blocking factor, the total number of blocks required, and the total internal fragmentation.
Solution:

1. **Logical File Size**: $S_{logical} = 2500 \times 120 = 300,000$ bytes.
2. **Blocking Factor**: $B_f = \lfloor 512/120 \rfloor = \lfloor 4.26 \rfloor = 4$ records per block.
3. **Total Blocks Required**: $K = \lceil 2500/4 \rceil = \lceil 625 \rceil = 625$ blocks.
4. **Internal Fragmentation**: Waste per block $= 512 - (4 \times 120) = 512 - 480 = 32$ bytes. Total internal fragmentation $= 625 \times 32 = 20,000$ bytes.

4.2 File Attributes and Type

File attributes (metadata) such as name, identifier, type, size, and protection are stored in the directory structure.

Methodology:

Directory Size and Overhead Computation To compute the size of a directory file and the overhead of storing attributes:

1. Identify the size of the file control block (FCB) or directory entry for each file. Let this be S_{entry}.
2. For a directory containing F files, the directory logical size is $D_{size} = F \times S_{entry}$.
3. The number of blocks required for the directory is $B_{dir} = \lceil D_{size}/B \rceil$, where B is the block size.

Worked Example:

Calculating Directory Overhead A file system has a block size of 4 KB (4096 bytes). Each directory entry requires 64 bytes to store file attributes. A directory contains 150 files. Calculate the number of disk blocks required to store this directory.

Solution:

1. **Directory Entry Size**: $S_{entry} = 64$ bytes.
2. **Logical Directory Size**: $D_{size} = 150 \times 64 = 9600$ bytes.
3. **Blocks Required**: $B_{dir} = \lceil 9600/4096 \rceil = \lceil 2.34 \rceil = 3$ blocks.

4.3 Access Method

Files can be accessed sequentially or directly (random access). Direct access involves computing the exact block address of a given record.

Methodology:

Direct Access Record Addressing For a file with fixed-length records of size R bytes and a block size of B bytes:

1. To access logical record i (where records are 0-indexed: $0, 1, 2 \ldots$):
2. Compute the byte offset of the record: $O = i \times R$.
3. Compute the logical block number (LBN) containing the record: $LBN = \lfloor O/B \rfloor$.
4. Compute the offset within that block: $Block_{offset} = O \pmod{B}$.

Worked Example:

Finding Record Location in Direct Access A direct access file has fixed-length records of 200 bytes. The disk block size is 1024 bytes. Find the logical block number and the block offset for accessing record number 45 (assuming 0-indexed records).

Solution:

1. **Byte Offset**: $O = 45 \times 200 = 9000$ bytes.
2. **Logical Block Number**: $LBN = \lfloor 9000/1024 \rfloor = \lfloor 8.789 \rfloor = 8$.
3. **Block Offset**: $Block_{offset} = 9000 \pmod{1024} = 9000 - (8 \times 1024) = 9000 - 8192 = 808$ bytes.

The record begins at byte 808 within logical block 8.

4.4 Allocation Method

Disk allocation methods dictate how disk blocks are assigned to files. The three primary methods are Contiguous, Linked, and Indexed.

4.4.1 Contiguous Allocation

Methodology:

Contiguous Allocation Access Computation In contiguous allocation, a file occupies a set of contiguous blocks on the disk.

1. A directory entry contains the *starting block* (S) and *length* (L).
2. To access logical block i (where $0 \leq i < L$), the physical block address is $P = S + i$.
3. **Disk Accesses**: Reading a specific block requires 1 disk access.

4.4.2 Linked Allocation

Methodology:

Linked Allocation Capacity and Overhead In linked allocation, each block contains a pointer to the next block.

1. Let the block size be B bytes and the pointer size be P bytes.
2. The usable data capacity per block is $B_{data} = B - P$.
3. To store a file of S bytes, the number of blocks required is $K = \lceil S/B_{data} \rceil$.
4. **Disk Accesses**: To read logical block i (0-indexed), the system must traverse the pointers from the beginning, requiring $i + 1$ disk accesses.

Worked Example:

Linked Allocation Calculations A file system uses linked allocation. The block size is 512 bytes, and a block pointer is 4 bytes long. A file is 2500 bytes in size. Calculate the number of disk blocks required and the usable space wasted in the last block.
Solution:

1. **Usable Data per Block**: $B_{data} = 512 - 4 = 508$ bytes.
2. **Blocks Required**: $K = \lceil 2500/508 \rceil = \lceil 4.92 \rceil = 5$ blocks.
3. **Wasted Space in Last Block**: The 5 blocks can hold $5 \times 508 = 2540$ bytes of data. Wasted data space $= 2540 - 2500 = 40$ bytes.

4.4.3 Indexed Allocation

Methodology:

Indexed Allocation Maximum File Size In indexed allocation, an index block contains pointers to the data blocks.

1. Let block size be B and pointer size be P.
2. The number of pointers an index block can hold is $N_p = \lfloor B/P \rfloor$.

3. For a single-level index, the maximum file size is $S_{max} = N_p \times B$.
4. For a two-level index, the maximum file size is $S_{max} = (N_p)^2 \times B$.
5. **Disk Accesses**: Reading a data block requires 2 disk accesses for single-level indexing (one for the index block, one for the data block).

Worked Example:

Indexed Allocation Maximum Size A file system uses indexed allocation with a block size of 4 KB (4096 bytes) and 4-byte pointers. Calculate the maximum file size supported by a single-level index and a two-level index.
Solution:

1. **Pointers per Block**: $N_p = \lfloor 4096/4 \rfloor = 1024$ pointers.
2. **Single-Level Index Maximum File Size**: $S_{single} = 1024 \times 4096$ bytes $= 4,194,304$ bytes $= 4$ MB.
3. **Two-Level Index Maximum File Size**: $S_{double} = (1024)^2 \times 4096$ bytes $= 1,048,576 \times 4096 = 4,294,967,296$ bytes $= 4$ GB.

4.5 Sharing

File sharing between multiple users requires a mechanism to define and check access rights.

Methodology:

Access Control Matrix Evaluation An access control matrix defines permissions where rows represent users/domains and columns represent files.

1. Let $M[i, j]$ be the set of rights User i has on File j. Rights are typically Read (R), Write (W), and Execute (X).
2. To determine if User i can perform operation Op on File j, verify if $Op \in M[i, j]$.
3. If a user belongs to a group, effective rights are the union of user rights and group rights.

Worked Example:

Evaluating Access Control Matrices Given the following access control matrix:

User	File A	File B	File C
User 1	R, W	R	R, W, X
User 2	R	R, W	None
User 3	None	R, X	R, W

Evaluate whether the following operations are allowed: 1. User 1 executing File C. 2. User 2 writing to File A. 3. User 3 reading File B.
Solution:

1. **User 1 on File C**: $M[1, C] = \{R, W, X\}$. Execute (X) is present. **Allowed**.
2. **User 2 on File A**: $M[2, A] = \{R\}$. Write (W) is not present. **Denied**.
3. **User 3 on File B**: $M[3, B] = \{R, X\}$. Read (R) is present. **Allowed**.

4.6 Protection

File protection ensures that only authorized users can access or modify files. A common scheme is the UNIX file permission system.

Methodology:

UNIX Octal Permission Calculation UNIX permissions consist of three sets: Owner, Group, and Others. Each set has Read (r), Write (w), and Execute (x) bits.

1. Assign binary values to permissions: $r = 4$, $w = 2$, $x = 1$, and $- = 0$.
2. For a permission string (e.g., `rwxr-x-x`), split it into three groups of three characters.
3. Calculate the octal value for each group by summing the values of the letters present.
4. The final permission is a 3-digit octal number.

Worked Example:

Converting Symbolic Permissions to Octal Convert the UNIX file permission string `rw-r-r-x` to its octal representation.
Solution:

1. **Split into groups**: Owner: `rw-`
   Group: `r-`
   Others: `r-x`
2. **Calculate octal for Owner**: `r` (4) + `w` (2) + `-` (0) = 6.
3. **Calculate octal for Group**: `r` (4) + `-` (0) + `-` (0) = 4.
4. **Calculate octal for Others**: `r` (4) + `-` (0) + `x` (1) = 5.
5. **Final Octal Permission**: 645.

Worked Example:

Applying Umask to Default Permissions A system has a default file creation permission of `666` (in octal) and a `umask` of `022`. Calculate the effective permission of a newly created file.
Solution: The effective permission is calculated by clearing the bits specified in the `umask` from the default permissions.

1. **Default Permission**: `666` (binary `110 110 110`).
2. **Umask**: `022` (binary `000 010 010`).
3. **Effective Permission Calculation** (Default AND NOT Umask):

Default:	110	110	110
NOT Umask:	111	101	101
Bitwise AND:	110	100	100

4. **Convert back to octal**: `110` → 6, `100` → 4, `100` → 4.
5. **Result**: The effective permission is `644` (`rw-r-r-`).

CHAPTER 5

Linux Commands and Shell Programming

5.1 Installation

Methodology:

Linux OS Installation Steps 1. Download a Linux distribution ISO (e.g. Ubuntu). 2. Create a bootable USB drive using standard flashing utilities. 3. Boot the system from the USB drive and enter the setup interface. 4. Configure basic localization: select language and timezone. 5. Partition the disk. Define root (`/`) swap and home (`/home`) partitions. 6. Create the initial user profile and assign root privileges. 7. Install the bootloader (GRUB) to the master boot record. 8. Reboot the system to complete the installation.

Worked Example:

Partitioning for Linux Installation Problem: A computer has 100GB of unallocated space. Determine an optimal partition scheme for a standard Linux installation.
Solution:
1. **Root Partition (`/`):** Allocate 30GB for operating system files and installed software. Format as `ext4`.
2. **Swap Space:** Allocate 8GB (equivalent to available system RAM) to act as virtual memory.
3. **Home Partition (`/home`):** Allocate the remaining 62GB for user documents downloads and personal settings. Format as `ext4`.

5.2 Process Commands

Methodology:

Process Management Methods 1. **`ps`**: Display currently running processes. Use `ps aux` to list processes for all users.
2. **`top`**: Display a dynamic real-time view of running processes sorting by CPU or memory usage.
3. **`kill`**: Terminate a process by its Process ID (PID). Syntax: `kill PID` or `kill -9 PID` (for a force kill).
4. **`jobs`**: List active jobs running in the background for the current terminal session.
5. **`fg`** / **`bg`**: Bring a suspended background job to the foreground (`fg`) or resume it in the background (`bg`).

Worked Example:

Finding and Terminating a Process Problem: A web browser process has become unresponsive. Identify its PID and forcibly terminate it.
Solution:
1. Search for the process by name:
`ps aux | grep browser`
2. Identify the PID from the second column of the resulting output (assume it is `4052`).
3. Force kill the process using signal 9:
`kill -9 4052`

5.3 Calendar Date and Sleep Commands

Methodology:

Time and Delay Operations 1. **`cal`**: Display a formatted calendar. Syntax: `cal [month] [year]`
2. **`date`**: Print or set the system date and time. Use format specifiers like `date "+%FORMAT"` to customize output.
3. **`sleep`**: Pause execution for a specified amount of time. Syntax: `sleep NUMBER[SUFFIX]`. Suffixes include `s` (seconds) `m` (minutes) `h` (hours).

Worked Example:

Formatting Date and Script Delays Problem: Display the calendar for August 1947. Then print the current date in `DD-MM-YYYY` format wait for 5 seconds and print a completion message.
Solution:
1. Print the specified calendar:
```
cal 8 1947
```
2. Display the formatted date:
```
date "+%d-%m-%Y"
```
3. Introduce a 5-second delay:
```
sleep 5
```
4. Print completion text:
```
echo "Execution Completed"
```

5.4 Directory and File Commands

Methodology:

File System Operations 1. **`pwd`**: Print the absolute path of the current working directory.
2. **`cd`**: Change the directory. Use `cd ..` to move up one hierarchy level.
3. **`ls`**: List directory contents. Use `ls -l` for permissions and ownership details.
4. **`mkdir`** / **`rmdir`**: Create or remove directories. Use `mkdir -p` to create nested parent directories.
5. **`touch`**: Create an empty file or update the access/modification timestamps of an existing file.
6. **`cp`**: Copy files or directories. Use `-r` to copy directories recursively.
7. **`mv`**: Move or rename files and directories.
8. **`rm`**: Remove files or directories. Use `-rf` to recursively and forcefully delete.
9. **`cat`**: Concatenate and print the content of files to standard output.

Worked Example:

Creating a Workspace Structure Problem: Create a directory structure `project/src` and `project/docs`. Create an empty file `main.c` in `src` copy it to `docs` and rename the copy to `backup.c`.
Solution:
1. Create the parent and child directories:
```
mkdir -p project/src project/docs
```
2. Create the empty source file:
```
touch project/src/main.c
```
3. Copy the file into the docs directory:
```
cp project/src/main.c project/docs/
```
4. Rename the newly copied file:
```
mv project/docs/main.c project/docs/backup.c
```

5.5 User Commands

Methodology:

User and Permission Management 1. **`who`** / **`whoami`**: Display a list of logged-in users or print the current effective username.
2. **`su`** / **`sudo`**: Switch the current user session or execute a specific command with superuser (root) privileges.
3. **`useradd`**: Create a new user account on the system.
4. **`passwd`**: Update user authentication tokens (passwords).
5. **`chmod`**: Modify file access permissions. Access modes can be symbolic (e.g. `u+x`) or octal (e.g. `755`).
6. **`chown`**: Change the owner and group assigned to a file or directory.

Worked Example:

Modifying File Permissions Problem: A file named `script.sh` requires execution rights. Change its permissions so the owner has read write and execute rights while the group and others have read and execute rights only.
Solution:
Using the numerical (octal) method:
1. Owner rights: Read (4) + Write (2) + Execute (1) = 7.
2. Group rights: Read (4) + Execute (1) = 5.
3. Other rights: Read (4) + Execute (1) = 5.
4. Apply the permissions to the file:

```
chmod 755 script.sh
```

5.6 Networking Commands

Methodology:

Network Configuration and Diagnostics 1. **`ping`**: Send ICMP ECHO_REQUEST packets to network hosts to diagnose connectivity issues.
2. **`ifconfig`** / **`ip`**: Display network interface parameters or configure routing and tunnel interfaces.
3. **`netstat`**: Display active network connections routing tables and interface statistics.
4. **`ssh`**: OpenSSH client for secure remote login over untrusted networks.
5. **`wget`**: Non-interactive command-line tool for downloading files from the web.

Worked Example:

Checking Network Status Problem: Verify internet connectivity by pinging the Google DNS server (8.8.8.8) limiting the test to 4 packets. Then download an image file from a provided URL.
Solution:
1. Test connectivity with a 4-packet limit:

```
ping -c 4 8.8.8.8
```

2. Download a file directly to the current directory:

```
wget http://example.com/logo.png
```

5.7 Shell Scripts

5.7.1 Basic Operators

Methodology:

Arithmetic and Relational Operators 1. **Arithmetic Operators**: Addition (+) Subtraction (-) Multiplication (*) Division (/) Modulus (%). Operations are typically enclosed in double parentheses.
Syntax: `result=$((a + b))`
2. **Relational Operators (Numeric)**: Equal (`-eq`) Not Equal (`-ne`) Greater Than (`-gt`) Less Than (`-lt`) Greater Than or Equal (`-ge`) Less Than or Equal (`-le`).

Worked Example:

Basic Arithmetic Script Problem: Write a shell script to assign two variables and compute their sum and product.
Solution:

```
#!/bin/bash
a=12
b=4
sum=$((a + b))
product=$((a * b))
echo "Sum is:  $sum"
echo "Product is:  $product"
```

5.7.2 Control Statements

Methodology:

Conditional Logic 1. **`if-else`**: Executes a block of code if a condition is true otherwise executes the alternative block.
Syntax:

```
if [ condition ]; then
    commands
else
    commands
fi
```

2. **`case`**: Executes blocks based on pattern matching acting as a multi-way branch.

Worked Example:

Checking Even or Odd Problem: Write a shell script to check whether a predefined number is even or odd using `if-else`.
Solution:

```
#!/bin/bash
num=14
if [ $((num % 2)) -eq 0 ]; then
    echo "$num is Even"
else
    echo "$num is Odd"
fi
```

5.7.3 Loop Statements

Methodology:

Iteration Structures 1. **`for` loop**: Iterates over a predefined sequence or list of items.
Syntax: `for var in list; do commands; done`
2. **`while` loop**: Repeatedly executes a block of commands as long as the evaluation condition remains true.
Syntax: `while [ condition ]; do commands; done`

Worked Example:

Printing a Multiplication Table Problem: Write a shell script to print the multiplication table of 5 using a `for` loop.
Solution:

```
#!/bin/bash
n=5
for i in {1..10}
do
    result=$((n * i))
    echo "$n x $i = $result"
done
```

Worked Example:

Factorial using a While Loop Problem: Write a shell script to calculate the factorial of a number using a `while` loop.
Solution:

```
#!/bin/bash
n=5
fact=1
while [ $n -gt 0 ]
do
    fact=$((fact * n))
    n=$((n - 1))
done
echo "Factorial is:  $fact"
```

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Unit 2: CPU Scheduling

Turnaround Time (TAT)

Turnaround Time is the total time taken from the submission of a process until its completion.

$$TAT = CT - AT$$

where CT is Completion Time and AT is Arrival Time.

Waiting Time (WT)

Waiting Time is the total time a process spends waiting in the ready queue.

$$WT = TAT - BT$$

where TAT is Turnaround Time and BT is Burst Time.

Unit 3: Memory Management

Contiguous Memory Allocation

When using base registers for memory protection and relocation:

$$\text{Physical Address} = \text{Logical Address} + \text{Base Register}$$

Paging

In a paging system, the physical address is derived from the frame number and the page offset.

$$\text{Physical Address} = (f \times \text{Page Size}) + d$$

where f is the Frame Number and d is the Page Offset.

Segmentation

In a segmented memory system, the physical address is derived from the segment base address and the offset.

$$\text{Physical Address} = \text{Segment Base Address} + d$$

where d is the Offset (which must be strictly less than the segment limit).

Swapping Time

The transfer time for swapping a process in or out of memory depends on the size of the process and the transfer rate of the backing store.

$$t_{\text{transfer}} = \frac{S}{R}$$

where S is the Size of the process and R is the Transfer Rate.

The time to swap out or swap in a process includes both transfer time and disk latency (seek time and rotational latency).

$$t_{\text{out}} = t_{\text{in}} = t_{\text{transfer}} + L$$

where L is the Latency.

The total time for a complete swap operation (swapping out one process and swapping in another of the same size) is:

$$t_{\text{total\_swap}} = t_{\text{out}} + t_{\text{in}} = 2 \times (t_{\text{transfer}} + L)$$

Unit 4: File Systems and Security

File Permissions (Umask)

When a new file or directory is created, its default permissions are modified by the system's `umask` value using bitwise operations.

$$\text{Effective Permission} = \text{Default Permission AND (NOT Umask)}$$