Los (4331602) - Problem Solver

Antigravity AI

June 4, 2026

Los

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

Operating System Basics

1.1 What is Operating System?

An Operating System (OS) is system software that manages computer hardware and software resources and provides common services for computer programs. It acts as an intermediary between the user of a computer and the computer hardware.

Methodology:

Identifying OS Functions To understand the role of an OS in a given scenario, use the following analytical steps:

1. **Identify the User Request:** Determine what the user or application is trying to achieve (e.g. read a file or print a document).
2. **Identify the Hardware Involved:** Note the specific hardware components required (e.g. Hard Disk Drive or Printer).
3. **Determine the OS Service:** Map the request to the corresponding OS subsystem (e.g. File Management or Device Management).
4. **Trace the Execution Flow:** Outline how the OS translates the software request into hardware instructions.

Worked Example:

Identify OS functions for saving a document **Problem:** A user clicks "Save" in a word processor. Identify the OS functions involved.
Solution:

1. **Identify the User Request:** The application requests to write data to persistent storage.
2. **Identify the Hardware Involved:** The Hard Disk Drive (HDD) or Solid State Drive (SSD).
3. **Determine the OS Service:** File Management and Storage Management.
4. **Trace the Execution Flow:**
   - The word processor issues a system call to the OS to write the file.
   - The OS file system maps the logical file to physical blocks on the disk.
   - The OS device driver sends specific electronic signals to the storage controller to write the data.

1.2 Need of Operating System

Without an OS, every application program would need its own UI and the comprehensive code required to handle all low-level functionality of the underlying computer hardware. The OS provides abstraction and resource

management.

Methodology:

Resource Allocation Evaluation To determine why an OS is needed for a specific workload:

1. **List Concurrent Tasks:** Identify multiple applications running simultaneously.
2. **Identify Resource Conflicts:** Determine which hardware resources (CPU or Memory) these tasks compete for.
3. **Apply OS Abstraction:** State how the OS resolves the conflict using scheduling or memory virtualization.

Worked Example:

Resolving Memory Conflicts **Problem:** Two applications simultaneously request 4GB of RAM on a system that only has 6GB of total physical RAM. Explain how the OS manages this need.
Solution:

1. **List Concurrent Tasks:** App A (needs 4GB) and App B (needs 4GB). Total requested is 8GB.
2. **Identify Resource Conflicts:** The system has only 6GB of physical RAM. Without an OS to manage this conflict the system would crash or applications would overwrite each other.
3. **Apply OS Abstraction:** The OS uses *Virtual Memory Management*. It allocates a portion of the hard drive as a swap file. Active memory pages for App A and App B remain in the physical RAM while inactive pages are temporarily swapped to the disk. Both applications operate believing they have their requested memory.

1.3 Types of Operating System

Operating systems are categorized based on their functional characteristics and how they interact with users and hardware.

Methodology:

OS Classification Algorithm To classify an operating system use the following decision process:

1. **Check for User Interaction:** If none and tasks run sequentially in groups it is a **Batch OS**.
2. **Check Time Constraints:** If the system must respond within a strict guaranteed time limit it is a **Real-Time OS (RTOS)**.
3. **Check Number of Users:** If multiple users access the system simultaneously via terminals it is a **Time-Sharing OS**.
4. **Check Node Distribution:** If the OS manages a group of independent computers and makes them appear as a single computer it is a **Distributed OS**.
5. **Check Networking Focus:** If the OS runs on a server to manage data users groups and security across a network it is a **Network OS**.

Worked Example:

Classifying Real-World Systems **Problem:** Classify the OS required for (A) a pacemaker (B) a mainframe processing payroll checks at night and (C) a university server allowing 50 students to compile code simultaneously.
Solution:

1. **System A (Pacemaker):** Needs strict timing constraints for heartbeats. Classification: **Real-Time OS**.
2. **System B (Payroll Mainframe):** Tasks run sequentially in groups with no user interaction during processing. Classification: **Batch OS**.
3. **System C (University Server):** Multiple users interacting with the system simultaneously. Classification: **Time-Sharing OS**.

1.4 Comparison between various Operating System

A systematic comparison of operating systems helps in selecting the right environment for specific computational problems.

Feature	**Linux**	**Windows**	**macOS**
Source Model	Open Source	Closed Source	Closed Source with open core
Kernel	Monolithic (Linux)	Hybrid (NT)	Hybrid (XNU)
Cost	Generally Free	Commercial License	Bundled with Hardware
Target Audience	Servers and Developers	General Consumers and Gaming	Creatives and Professionals

Methodology:

OS Selection Framework

1. **Analyze Workload Requirements:** Identify specific software dependencies and hardware architectures.
2. **Evaluate Budget Constraints:** Determine licensing costs versus support needs.
3. **Assess Security Needs:** Consider the threat model and the availability of source code for auditing.
4. **Final Selection:** Match requirements against OS profiles.

Worked Example:

Selecting an OS for a Web Server **Problem:** A startup needs to deploy 50 web servers on cloud instances. The budget for software licenses is zero. They require high customization and secure SSH access. Select the appropriate OS.
Solution:

1. **Analyze Workload:** Needs web hosting software (like Apache or Nginx) and SSH.
2. **Evaluate Budget:** $0 software budget rules out commercial licenses.
3. **Assess Security:** Needs high customization and robust network security.
4. **Final Selection: Linux** is the optimal choice because it is open-source (free) highly customizable and industry-standard for web servers.

1.5 Linux Operating System

Linux is a family of open-source Unix-like operating systems based on the Linux kernel.

1.5.1 History and Features

Linus Torvalds created the Linux kernel in 1991. Key features include portability open-source licensing multi-user capability multi-programming hierarchical file system and superior security.

1.5.2 Architecture and Components

The Linux architecture consists of distinct layers that manage the translation of user commands into hardware actions.

Methodology:

Linux Architecture Tracing To analyze how a command is processed in Linux:

1. **User Space (Application/Utility):** The user executes a program (e.g. `ls` or `cat`).
2. **Shell Interface:** The shell parses the command and initiates a system call.
3. **System Call Interface:** Acts as a bridge transferring the request from user space to kernel space.
4. **Kernel Space:** The core kernel processes the request managing CPU memory and processes.
5. **Hardware Layer:** Device drivers interact with physical hardware to retrieve or modify data.

Worked Example:

Tracing the ls command **Problem:** Trace the execution flow of the `ls` command (used to list directory contents) through the Linux architecture components.
Solution:

1. **User Space:** The user types `ls` in the terminal. The terminal emulator captures the input.
2. **Shell:** The Bash shell parses the input finds the `ls` executable and requests execution.
3. **System Call:** The shell uses system calls like `fork()` and `execve()` to create a new process for `ls`. The `ls` program then uses `open()` and `getdents()` to read the directory.
4. **Kernel:** The Linux kernel receives the `getdents()` call accesses the Virtual File System (VFS) and identifies the specific filesystem driver (e.g. ext4).
5. **Hardware:** The storage device driver requests the directory inode data from the physical storage drive. The data is returned back up the stack to be displayed in the terminal.

1.5.3 Distributions

A Linux distribution (distro) is an operating system made from a software collection that is based on the Linux kernel and often a package management system. Common distributions include Ubuntu (Debian-based) CentOS/RHEL (Red Hat-based) and Arch Linux.

CHAPTER 2

Process Management and Inter Process Communication

2.1 Process and Process Management

A process is a program in execution. The process model represents each running program as a sequential process with its own state.

- **Process States:** New, Ready, Running, Waiting (or Blocked), and Terminated.
- **Process Control Block (PCB):** A data structure maintained by the Operating System for every process, containing information such as process state, program counter, CPU registers, and memory-management information.

2.2 Process Scheduling

Scheduling determines which process in the ready queue is allocated to the CPU.

- **Types of Schedulers:** Long-term (Job scheduler), Short-term (CPU scheduler), and Medium-term (Swapping).
- **Criteria:** Maximize CPU Utilization and Throughput; Minimize Turnaround Time, Waiting Time, and Response Time.

Methodology:

Calculating CPU Scheduling Metrics To evaluate scheduling algorithms, use the following fundamental formulas:

1. **Completion Time (CT):** The absolute time at which a process finishes execution.
2. **Turnaround Time (TAT):** The total time taken from submission to completion.

$$TAT = CT - AT$$

3. **Waiting Time (WT):** The total time a process spends waiting in the ready queue.

$$WT = TAT - BT$$

4. **Average Times:** Sum the TAT or WT for all processes and divide by the number of processes.

2.2.1 First Come First Serve Algorithm

FCFS allocates the CPU to the process that requests it first. It is non-preemptive.

Worked Example:

First Come First Serve Evaluation Given the following processes with their Arrival Time (AT) and Burst Time (BT), compute the Average Turnaround Time and Average Waiting Time using FCFS.

Process	AT	BT
P1	0	4
P2	1	3
P3	2	1
P4	3	2

Step 1: Determine the Execution Sequence (Gantt Chart representation) Processes execute in the order they arrive: P1 → P2 → P3 → P4.

P1	P2	P3	P4
0 4	4 7	7 8	8 10

Step 2: Calculate CT, TAT, and WT

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	4	4	$4-0=4$	$4-4=0$
P2	1	3	7	$7-1=6$	$6-3=3$
P3	2	1	8	$8-2=6$	$6-1=5$
P4	3	2	10	$10-3=7$	$7-2=5$

Step 3: Compute Averages

- Average TAT $= (4+6+6+7)/4 = 23/4 = 5.75$ units
- Average WT $= (0+3+5+5)/4 = 13/4 = 3.25$ units

2.2.2 Shortest Job First Algorithm

SJF selects the waiting process with the smallest execution time. We will demonstrate the non-preemptive version.

Worked Example:

Shortest Job First Non Preemptive Evaluation Given the following processes, compute the Average Turnaround Time and Average Waiting Time using Non-Preemptive SJF.

Process	AT	BT
P1	0	6
P2	1	4
P3	2	2
P4	3	3

Step 1: Determine the Execution Sequence At time 0, only P1 is available. It runs to completion at time 6. At time 6, P2, P3, and P4 have arrived. The burst times are 4, 2, and 3 respectively. P3 is selected (shortest BT). P3 runs from 6 to 8. At time 8, P4 is selected (BT=3 is shorter than P2's BT=4). P4 runs from 8 to 11. Finally, P2 runs from 11 to 15.

P1	P3	P4	P2
0 6	6 8	8 11	11 15

Step 2: Calculate CT, TAT, and WT

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	6	6	$6-0=6$	$6-6=0$
P2	1	4	15	$15-1=14$	$14-4=10$
P3	2	2	8	$8-2=6$	$6-2=4$
P4	3	3	11	$11-3=8$	$8-3=5$

Step 3: Compute Averages

- Average TAT $= (6+14+6+8)/4 = 34/4 = 8.5$ units
- Average WT $= (0+10+4+5)/4 = 19/4 = 4.75$ units

2.2.3 Round Robin Algorithm

Round Robin assigns a fixed time quantum to each process in equal portions and in circular order.

Worked Example:

Round Robin Scheduling Evaluation Calculate average TAT and WT for the following processes using Round Robin scheduling with a Time Quantum (TQ) = 2.

Process	AT	BT
P1	0	5
P2	1	3
P3	2	1

Step 1: Execute using Ready Queue and Gantt Chart

- Time 0: P1 arrives. Run P1 for 2 units. (P1 remaining: 3)
- Time 2: P2 and P3 have arrived. Queue: P2, P3, P1. Run P2 for 2 units. (P2 remaining: 1)
- Time 4: Queue: P3, P1, P2. Run P3 for 1 unit (finishes).
- Time 5: Queue: P1, P2. Run P1 for 2 units. (P1 remaining: 1)
- Time 7: Queue: P2, P1. Run P2 for 1 unit (finishes).
- Time 8: Queue: P1. Run P1 for 1 unit (finishes).
- Time 9: All completed.

P1	P2	P3	P1	P2	P1
0 2	2 4	4 5	5 7	7 8	8 9

Step 2: Calculate CT, TAT, and WT

Process	AT	BT	CT	TAT	WT
P1	0	5	9	$9-0=9$	$9-5=4$
P2	1	3	8	$8-1=7$	$7-3=4$
P3	2	1	5	$5-2=3$	$3-1=2$

Step 3: Compute Averages

- Average TAT $= (9+7+3)/3 = 19/3 = 6.33$ units
- Average WT $= (4+4+2)/3 = 10/3 = 3.33$ units

2.3 Inter Process Communication

Inter Process Communication (IPC) allows processes to synchronize and share data.

- **Critical Section:** A segment of code where shared resources are accessed.
- **Race Condition:** A situation where the final outcome depends on the sequence of execution.
- **Mutual Exclusion:** Guaranteeing that only one process can enter its critical section at a time.
- **Semaphore:** An integer variable used for signaling among processes.

Methodology:

Semaphore Operations wait and signal Semaphores are accessed via two atomic operations, commonly called `wait()` (or P) and `signal()` (or V).

1. **Wait (P) operation:** Decrements the semaphore value. If the value becomes negative, the process is blocked.

```
wait(S) {
    while (S <= 0); // busy wait
    S--;
}
```

2. **Signal (V) operation:** Increments the semaphore value. If there are blocked processes, one is awakened.

```
signal(S) {
    S++;
}
```

To achieve mutual exclusion on a critical section, a semaphore initialized to 1 (mutex) is used.

Worked Example:

Achieving Mutual Exclusion with Semaphores Show how two processes, P1 and P2, can execute their critical sections without causing a race condition using a binary semaphore named `mutex` initialized to 1.
Step 1: Initialization

```
int mutex = 1;
```

Step 2: Process Execution Pattern Each process must follow this structure:

```
Process Pi {
    wait(mutex);
    // CRITICAL SECTION
    signal(mutex);
    // REMAINDER SECTION
}
```

Step 3: Trace a Concurrent Execution Scenario

- P1 arrives and executes `wait(mutex)`. Since `mutex = 1`, it is decremented to 0, and P1 enters its Critical Section.
- While P1 is in its Critical Section, P2 arrives and executes `wait(mutex)`. Since `mutex = 0`, P2 enters a busy-wait loop (or blocks). Mutual exclusion is preserved.

- P1 finishes its Critical Section and executes `signal(mutex)`, changing `mutex` back to 1.
- P2's `wait` operation now succeeds, decrementing `mutex` to 0, and P2 enters its Critical Section.

2.4 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.

- **Characteristics:** Mutual exclusion, Hold and wait, No preemption, and Circular wait (all four must occur simultaneously).
- **Prevention:** Ensuring at least one of the four characteristics cannot hold.
- **Avoidance:** Dynamically examining the resource allocation state (Banker's Algorithm).
- **Detection and Recovery:** Allowing deadlocks, detecting them, and recovering (e.g., by killing processes).

Methodology:

Bankers Algorithm for Deadlock Avoidance The Banker's Algorithm determines if the system is in a safe state. It requires 3 matrices: **Allocation**, **Max**, and **Available**.

1. **Calculate Need Matrix:**
$$Need[i, j] = Max[i, j] - Allocation[i, j]$$
2. **Initialize Work and Finish:** Let `Work` = `Available` and `Finish`[i] = `false` for all processes i.
3. **Find a runnable process:** Find an index i such that:
   - `Finish`[i] == `false`
   - $Need_i \leq Work$

   If no such i exists, go to Step 5.
4. **Simulate Allocation:**
$$Work = Work + Allocation_i$$
$$\texttt{Finish}[i] = \texttt{true}$$
   Record process P_i in the safe sequence and go back to Step 3.
5. **Check Safety:** If `Finish`[i] == `true` for all i, the system is in a safe state. Otherwise, it is unsafe and a deadlock may occur.

Worked Example:

Evaluating Safe State using Bankers Algorithm Consider a system with 5 processes (P_0 to P_4) and 3 resource types (A, B, C). There are 10 instances of A, 5 of B, and 7 of C. Check if the system is in a safe state.
Given State:

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

Step 1: Calculate the Need Matrix (Max - Allocation)

Process	Need (A, B, C)		
P_0	$(7-0=7)$	$(5-1=4)$	$(3-0=3)$
P_1	$(3-2=1)$	$(2-0=2)$	$(2-0=2)$
P_2	$(9-3=6)$	$(0-0=0)$	$(2-2=0)$
P_3	$(2-2=0)$	$(2-1=1)$	$(2-1=1)$
P_4	$(4-0=4)$	$(3-0=3)$	$(3-2=1)$

So, the Need matrix is: $P_0(7,4,3)$, $P_1(1,2,2)$, $P_2(6,0,0)$, $P_3(0,1,1)$, $P_4(4,3,1)$.
Step 2: Execute Safety Algorithm

- **Initial Work:** $= (3,3,2)$. Finish array is all False.
- **Check P_0:** Need $(7,4,3) \leq$ Work $(3,3,2)$? **False**.
- **Check P_1:** Need $(1,2,2) \leq$ Work $(3,3,2)$? **True**.
  Execute P_1: New Work $= (3,3,2) + (2,0,0) = (5,3,2)$. P_1 finishes.
- **Check P_2:** Need $(6,0,0) \leq$ Work $(5,3,2)$? **False**.
- **Check P_3:** Need $(0,1,1) \leq$ Work $(5,3,2)$? **True**.
  Execute P_3: New Work $= (5,3,2) + (2,1,1) = (7,4,3)$. P_3 finishes.
- **Check P_4:** Need $(4,3,1) \leq$ Work $(7,4,3)$? **True**.
  Execute P_4: New Work $= (7,4,3) + (0,0,2) = (7,4,5)$. P_4 finishes.
- **Check P_0 (again):** Need $(7,4,3) \leq$ Work $(7,4,5)$? **True**.
  Execute P_0: New Work $= (7,4,5) + (0,1,0) = (7,5,5)$. P_0 finishes.
- **Check P_2 (again):** Need $(6,0,0) \leq$ Work $(7,5,5)$? **True**.
  Execute P_2: New Work $= (7,5,5) + (3,0,2) = (10,5,7)$. P_2 finishes.

Conclusion: Since all processes finish successfully, the system is in a **Safe State**. The safe sequence is $\langle P_1, P_3, P_4, P_0, P_2 \rangle$.

CHAPTER 3

File Management and Linux File Structure

3.1 File Management

File management is a crucial function of the operating system that handles the storage, retrieval, naming, sharing, and protection of files. In this section, we focus on the quantitative and problem-solving aspects of file systems, including block calculations, allocation efficiency, and access operations.

3.1.1 File Structure Attributes and Types

Every file consists of attributes (name, identifier, type, location, size, protection, time/date) and relies on specific structural organizations. From a computational perspective, protecting files using numerical permission modes is a common operation.

Methodology:

Calculating Numeric File Permissions In Unix/Linux systems file permissions are calculated using a 3-digit octal number representing Owner, Group, and Others.

1. Assign values to permissions: Read (R) = 4, Write (W) = 2, Execute (X) = 1.
2. Sum the values for each category (Owner, Group, Others).
3. Combine the three sums to form the octal permission code.

Worked Example:

Setting File Permissions A file needs the following permissions: Owner can read, write, and execute; Group can read and execute; Others can only read. Calculate the numeric chmod command value.
Solution:

1. **Owner:** Read(4) + Write(2) + Execute(1) = 7
2. **Group:** Read(4) + Execute(1) = 5
3. **Others:** Read(4) = 4

The numeric permission is **754**. The corresponding command would be `chmod 754 filename`.

3.1.2 Directory Structures

Directory structures organize files. Common types include Single-Level, Two-Level, Tree-Structured, and Acyclic-Graph directories. Problem solving often involves calculating the cost of resolving paths or identifying unique files.

Methodology:

Path Resolution and Distance Calculation To calculate the shortest path distance between two nodes (directories/files) in a tree-structured directory:

1. Identify the absolute path of Node A and Node B.
2. Find the Lowest Common Ancestor (LCA) of Node A and Node B.
3. Count the number of edges (steps) from LCA to Node A (D_A) and LCA to Node B (D_B).
4. Total distance = $D_A + D_B$.

Worked Example:

Directory Distance Calculation Given a directory tree where `/home/user1/docs/file.txt` and `/home/user1/pics/img.png` exist. Calculate the minimum path distance between `file.txt` and `img.png`.
Solution:

1. Path A: `/home/user1/docs/file.txt`
2. Path B: `/home/user1/pics/img.png`
3. The Lowest Common Ancestor (LCA) is `/home/user1/`.
4. Distance from LCA to A: `user1` → `docs` → `file.txt` (2 steps).
5. Distance from LCA to B: `user1` → `pics` → `img.png` (2 steps).
6. Total distance = $2 + 2 = 4$ steps.

3.1.3 Access Methods

Files can be accessed sequentially or directly. Direct access involves computing the exact physical block containing the desired logical record.

Methodology:

Direct Access Block Computation For a file structured as fixed-length logical records:

1. Let L be the logical record length in bytes.
2. Let B be the physical block size of the disk in bytes.
3. The number of records per block $R = \lfloor B/L \rfloor$.
4. To access the N-th logical record (0-indexed), the logical block number P is given by: $P = \lfloor N/R \rfloor$
5. The byte offset within that block is: $Offset = (N \pmod R) \times L$

Worked Example:

Computing Physical Block and Offset A disk has a block size of 1024 bytes. A file consists of logical records of length 100 bytes each. Calculate the block number and byte offset to access the 45th record (index 45).
Solution:

1. Block size $B = 1024$ bytes, Record length $L = 100$ bytes.
2. Records per block $R = \lfloor 1024/100 \rfloor = 10$ records/block.
3. Target record index $N = 45$.

4. Block number $P = \lfloor 45/10 \rfloor = 4$. (The record is in the 5th block, logical index 4).
5. Offset within block = $(45 \pmod{10}) \times 100 = 5 \times 100 = 500$ bytes.

The 45th record is located in Block index 4 starting at an offset of 500 bytes.

3.1.4 Allocation Methods

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

Methodology:

Disk Access Count for Allocation Methods To read the i-th logical block of a file (where blocks are 0-indexed):

1. **Contiguous Allocation:** Requires 1 disk access. The block is at physical address $Start + i$.
2. **Linked Allocation:** Requires $i + 1$ disk accesses. Must traverse the linked list from the first block to the i-th block.
3. **Indexed Allocation:** Requires 2 disk accesses. One to read the index block (assuming it is not cached) and one to read the data block.

Worked Example:

Evaluating Allocation Methods A file consists of 100 blocks. Assume the directory is already in memory. How many disk I/O operations are required to read the 50th logical block (index 50) for Contiguous Linked and Indexed allocation methods?
Solution:

1. **Contiguous:** Direct calculation gives the address. Requires **1** disk access.
2. **Linked:** To reach block 50, we must read block 0, 1, 2... up to 50. This requires $50 + 1 =$ **51** disk accesses.
3. **Indexed:** Read the index block (1 access) + read the target data block (1 access). Total = **2** disk accesses.

3.2 Linux File System

The Linux file system follows a unified hierarchical structure. Features include support for multiple file system types (ext3, ext4, xfs) and a robust permission model. Everything in Linux is treated as a file, including hardware devices.

3.2.1 Structure Features and Types

At the core of the Linux file system is the **inode** (index node). Every file is represented by an inode, which stores metadata and pointers to the physical data blocks. The inode pointer structure is a fundamental concept for computing maximum file sizes.

Methodology:

Computing Maximum File Size via Inodes A standard Linux inode contains 12 direct pointers, 1 single indirect pointer, 1 double indirect pointer, and 1 triple indirect pointer.

1. Let block size be B bytes and pointer size be P bytes.
2. Number of pointers per block $N = B/P$.

3. Data from direct pointers $= 12 \times B$.
4. Data from single indirect $= N \times B$.
5. Data from double indirect $= N^2 \times B$.
6. Data from triple indirect $= N^3 \times B$.
7. Total maximum file size = Sum of all the above data capacities.

Worked Example:

Maximum Linux File Size Calculation Consider a Linux file system with a block size of 4 KB (4096 bytes) and a disk pointer size of 4 bytes. Calculate the maximum file size supported by this file system using the standard inode structure.

Solution:

1. Block size $B = 4$ KB. Pointer size $P = 4$ bytes.
2. Pointers per block $N = 4096/4 = 1024$.
3. **Direct pointers capacity:** 12×4 KB $= 48$ KB.
4. **Single indirect capacity:** 1024×4 KB $= 4096$ KB $= 4$ MB.
5. **Double indirect capacity:** $1024 \times 1024 \times 4$ KB $= 1048576 \times 4$ KB $= 4$ GB.
6. **Triple indirect capacity:** $1024 \times 1024 \times 1024 \times 4$ KB $= 4$ TB.
7. **Total File Size:** 48 KB $+ 4$ MB $+ 4$ GB $+ 4$ TB ≈ 4.004 TB.

The maximum file size is approximately 4 TB.

CHAPTER 4
Security and Protection

4.1 Security in Operating system

The operating system must ensure the security of user data and system resources. Computational approaches in security involve authentication algorithms, threat mitigation procedures, and policy enforcement checks.

4.1.1 System Authentication

Authentication verifies the identity of users and processes. A common computational technique is the use of one-way hash functions combined with a cryptographic salt to securely store and evaluate passwords.

Methodology:

One Way Hash Authentication Algorithm **Step 1:** Read the user credentials (Username and Password).
Step 2: Retrieve the stored `Salt` for the Username from the system database.
Step 3: Concatenate the `Salt` and the `Password` to form the combined string: `Salt + Password`.
Step 4: Apply a cryptographic hash function H to compute the current hash value: H(`Salt + Password`).
Step 5: Compare the computed hash value with the stored hash value in the database. If they match exactly, authentication is successful; otherwise, access is denied.

Worked Example:

Password Verification Process **Problem:** A system stores passwords using a simplified hashing function that assigns a numerical value to each letter (A=1, B=2, ..., Z=26) and sums them up, adding the sum of the salt characters. Given: Stored Salt: "XY" Stored Hash: 75 Provided Password: "OS" Verify if the authentication should succeed using the algorithmic steps.
Solution: Step 1: Calculate the numerical value of the Salt.
Value of X = 24
Value of Y = 25
Salt Value = $24 + 25 = 49$
Step 2: Calculate the numerical value of the provided Password.
Value of O = 15
Value of S = 19
Password Value = $15 + 19 = 34$
Step 3: Compute the Hash.
Hash = Salt Value + Password Value
Hash = $49 + 34 = 83$
Step 4: Compare with Stored Hash.
Computed Hash (83) $\neq$ Stored Hash (75).
Result: The calculated hash does not match the stored hash. Authentication fails.

4.1.2 Program Threats and System Threats

Security measures must quickly identify and mitigate unauthorized access and resource exploitation. Program threats include malicious code embedded in applications (viruses and Trojans), while system threats abuse

network services (worms and Denial of Service attacks).

Methodology:

Threat Mitigation Algorithm **Step 1:** Monitor system resource allocation metrics continuously (CPU usage, memory consumption, network requests).
Step 2: Compare current metrics against baseline thresholds to identify statistical anomalies (e.g. excessive network requests indicate a Denial of Service).
Step 3: If a program threat is mathematically identified (e.g. unauthorized file modification signature match), terminate the offending process ID immediately.
Step 4: If a system threat is identified (e.g. port flooding), drop packets and block the source IP address mathematically at the firewall interface.

Worked Example:

Identifying and Mitigating a DoS Attack **Problem:** A web server has a baseline network traffic of 500 requests per minute. The system threshold for an anomaly is configured as 3× the baseline. Over a 3-minute monitoring window, the server records: Minute 1: 520 requests Minute 2: 1800 requests Minute 3: 2500 requests (90% originating from IP 192.168.1.10) Identify the threat and specify mitigation steps.
Solution: Step 1: Check against mathematical thresholds.
Threshold $= 3 \times 500 = 1500$ requests per minute.
Minute 1 (520) < 1500 (Normal Traffic)
Minute 2 (1800) > 1500 (Anomaly Detected)
Minute 3 (2500) > 1500 (Anomaly Detected)
Step 2: Identify the threat type.
The rapid geometric surge in network traffic far exceeding the threshold without user intervention indicates a Denial of Service (DoS) attack, classified as a system threat.
Step 3: Apply mitigation logic.
Analyze the traffic source: $0.90 \times 2500 = 2250$ requests in Minute 3 came from IP 192.168.1.10.
Action: Update firewall rule to block IP 192.168.1.10 and terminate existing connections to free resources.

4.1.3 Operating System Security Policies and Procedures

Operating systems enforce security policies to algorithmically control access to objects (files, hardware) by subjects (users, processes). A widespread mathematical model for enforcing confidentiality policies is the Mandatory Access Control (MAC) based on security clearances.

Methodology:

Mandatory Access Control Policy Check **Step 1:** Assign a numeric security clearance level $C(S)$ to each Subject S.
Step 2: Assign a numeric classification level $L(O)$ to each Object O.
Step 3: To allow a Read operation, verify that $C(S) \geq L(O)$. This enforces "No Read Up".
Step 4: To allow a Write operation, verify that $C(S) \leq L(O)$. This enforces "No Write Down".

Worked Example:

Applying Security Clearances **Problem:** An OS uses a security policy with three hierarchical levels: Top Secret (3), Secret (2), and Confidential (1). Subject A has a clearance of Secret (2). Object X is classified as Top Secret (3). Object Y is classified as Confidential (1). Determine whether Subject A is granted Read or Write access to Object X and Object Y.
Solution: Evaluate Object X (Classification Level = 3):
Read Request: Check if $C(A) \geq L(X) \Rightarrow 2 \geq 3$. This is False. Read is DENIED.
Write Request: Check if $C(A) \leq L(X) \Rightarrow 2 \leq 3$. This is True. Write is ALLOWED.

Evaluate Object Y (Classification Level = 1):
Read Request: Check if $C(A) \geq L(Y) \Rightarrow 2 \geq 1$. This is True. Read is ALLOWED.
Write Request: Check if $C(A) \leq L(Y) \Rightarrow 2 \leq 1$. This is False. Write is DENIED.

4.2 Protection

Protection refers to the strict mechanisms for controlling the access of programs, processes, or users to the resources defined by a computer system.

4.2.1 Protection Domain

A system consists of a collection of objects (hardware or software). Each object has a unique identifier and can be accessed through specific operations. A protection domain is a computational boundary specifying the resources that a process may access and the precise operations it may invoke.

Methodology:

Defining Protection Domains **Step 1:** Define domains $D_1, D_2, \ldots, D_n$.
Step 2: Define objects $O_1, O_2, \ldots, O_m$.
Step 3: Specify a set of access rights for each domain. An access right is represented computationally as an ordered pair $\langle$object-name, rights-set$\rangle$.
Step 4: Map a running process to a domain to rigidly determine its permissible operations on all system objects.

Worked Example:

Evaluating Domain Access Rights **Problem:** Given the following protection domains: $D_1 = \{\langle F_1, \{read, write\}\rangle, \langle F_2, \{read\}\rangle\}$ $D_2 = \{\langle F_2, \{read, execute\}\rangle, \langle F_3, \{read, write, execute\}\rangle\}$ Process P_A executes in D_1 and Process P_B executes in D_2. Which process is allowed to execute file F_2? Can Process P_A write to file F_3?
Solution: Step 1: Check F_2 execution privileges.
In domain D_1, the rights set for F_2 is $\{read\}$. Execute is not present.
In domain D_2, the rights set for F_2 is $\{read, execute\}$. Execute is present.
Therefore, Process P_B (running in D_2) is allowed to execute F_2.
Step 2: Check P_A write privileges on F_3.
Process P_A operates strictly in D_1.
Scanning D_1, it does not contain any access right tuple for F_3.
Therefore, P_A cannot write to F_3 (access is denied by default).

4.2.2 Access Control List

To comprehensively model protection, an Access Matrix is created using rows to represent domains and columns to represent objects. For practical implementation and efficiency, this matrix is decoupled into Access Control Lists (extracted by column) or Capability Lists (extracted by row).

Methodology:

Access Matrix Implementation **Step 1:** Construct the Access Matrix M, where entry $M[i, j]$ mathematically defines the set of operations Domain D_i can perform on Object O_j.
Step 2: To generate an **Access Control List (ACL)** for a given object O_j, iterate sequentially through its column. Create a list of paired entities $\langle$Domain, Rights$\rangle$ for all non-empty matrix entries.
Step 3: To generate a **Capability List** for a given domain D_i, iterate sequentially through its row. Create a list of paired entities $\langle$Object, Rights$\rangle$ for all non-empty matrix entries.

Worked Example:

Matrix Conversion to ACL and Capabilities **Problem:** Given the following Access Matrix for Domains D_1, D_2, D_3 and Files F_1, F_2, F_3:

Domain	F1	F2	F3
D1	read		read, write
D2		execute	read
D3	read, write	read	

Compute the precise Access Control List (ACL) for each file and the Capability List for each domain.
Solution:
Step 1: Extract Access Control Lists (column-wise)
For **F1**: Scan the column under F1.
ACL(F1) = $\{\langle D_1, \{read\}\rangle, \langle D_3, \{read, write\}\rangle\}$
For **F2**: Scan the column under F2.
ACL(F2) = $\{\langle D_2, \{execute\}\rangle, \langle D_3, \{read\}\rangle\}$
For **F3**: Scan the column under F3.
ACL(F3) = $\{\langle D_1, \{read, write\}\rangle, \langle D_2, \{read\}\rangle\}$
Step 2: Extract Capability Lists (row-wise)
For **D1**: Scan the row corresponding to D1.
Capability(D1) = $\{\langle F_1, \{read\}\rangle, \langle F_3, \{read, write\}\rangle\}$
For **D2**: Scan the row corresponding to D2.
Capability(D2) = $\{\langle F_2, \{execute\}\rangle, \langle F_3, \{read\}\rangle\}$
For **D3**: Scan the row corresponding to D3.
Capability(D3) = $\{\langle F_1, \{read, write\}\rangle, \langle F_2, \{read\}\rangle\}$

CHAPTER 5

Linux commands and shell programming

5.1 Installation of Linux Operating System

Methodology:

Steps for Linux OS Installation The installation of a Linux distribution generally follows these standard computational and procedural steps:

1. **Download the ISO Image:** Obtain the desired Linux distribution ISO file from its official repository.
2. **Create a Bootable Media:** Flash the ISO onto a USB drive using a tool such as Rufus or dd.
3. **Boot from USB:** Modify the system BIOS/UEFI boot order to prioritize the bootable USB.
4. **Start the Installer:** Boot into the live environment and launch the graphical or text-based installation wizard.
5. **Disk Partitioning:** Allocate disk space. Create necessary partitions such as root (`/`), swap, and home (`/home`).
6. **User Configuration:** Define the root password and create a standard user account with a username and password. Set the hostname.
7. **Finalize Installation:** Allow the installer to copy files and configure the bootloader (like GRUB). Reboot and remove the installation media.

Worked Example:

Installing Ubuntu as a Virtual Machine **Problem:** Outline the steps to install Ubuntu Linux on a virtual machine using virtualization software.
Solution:

1. Open the virtualization software (e.g. VirtualBox) and click **New** to create a VM.
2. Name the VM and select "Linux" and "Ubuntu" as the type and version.
3. Allocate at least 2048 MB of RAM.
4. Create a Virtual Hard Disk of at least 20 GB.
5. Go to the VM **Settings** > **Storage**, click the optical drive, and select the downloaded Ubuntu ISO file.
6. Start the VM to boot from the ISO. Select "Install Ubuntu".
7. Follow the on-screen prompts for language selection, keyboard layout, and disk partitioning (select "Erase disk and install Ubuntu" for simplicity in a VM).
8. Enter user details and wait for the installation to finish. Restart the VM to access the new Linux system.

5.2 Basic commands: calendar date and others

Methodology:

Basic Utilities Syntax Linux provides fundamental commands for basic operations.

- `cal`: Displays the calendar.
  - Syntax: `cal [month] [year]`
- `date`: Displays or sets the system date and time.
  - Syntax: `date [+format]`
- `echo`: Prints text or variables to standard output.
  - Syntax: `echo [string]`
- `clear`: Clears the terminal screen.

Worked Example:

Using Calendar and Date Commands **Problem:** Display the calendar for August 2023 and display the current date in `YYYY-MM-DD` format.
Solution:

1. To display the calendar for a specific month and year, execute the `cal` command with the month (8) and year (2023) as arguments:

   ```
   cal 8 2023
   ```

2. To format the date, use the `date` command with the `+` format specifier. For `YYYY-MM-DD`, use `%Y-%m-%d`:

   ```
   date "+%Y-%m-%d"
   ```

5.3 Editing files with vi vim gedit and gcc

Methodology:

Editors and C Compiler Editing configuration files and source code requires command-line or graphical text editors.

1. **vi and vim:** Powerful command-line text editors operating in multiple modes:
   - **Command Mode:** Default mode for navigation and deletion.
   - **Insert Mode:** Reached by pressing `i`. Used to type text. Press `Esc` to exit.
   - **Ex Mode:** Reached by pressing `:` from Command mode. Used for saving (`:w`) and quitting (`:q` or `:wq`).
2. **gedit:** A graphical text editor used in desktop environments. Launch using `gedit filename.txt`.
3. **gcc:** The GNU Compiler Collection used to compile C and C++ programs.
   - Syntax: `gcc source.c -o executable_name`

Worked Example:

Writing and Compiling a C Program **Problem:** Write a "Hello World" C program using the `vim` editor and compile it using `gcc`.
Solution:

1. Open a new file in vim:

```
vim hello.c
```

2. Press `i` to enter Insert mode and write the C code:

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

3. Press `Esc` to return to Command mode.
4. Type `:wq` and press `Enter` to save the file and quit vim.
5. Compile the program using gcc:

```
gcc hello.c -o hello
```

6. Execute the compiled binary:

```
./hello
```

5.4 Linux Super user commands

Methodology:

Administrative Commands These commands are essential for managing user sessions and administrative privileges:

- `su` (Substitute User): Switches the current user session to another user. `su -` switches to the root user environment.
- `loginname` / `logname`: Prints the login name of the current user.
- `exit`: Terminates the current shell session or logs out the user.
- `whoami`: Displays the effective username of the current user.
- `hostname`: Displays or modifies the system's network name.

Worked Example:

Switching Users and Checking Hostname **Problem:** Check your current username switch to the root user environment check the hostname and exit.
Solution:

1. Identify the current effective user:

```
whoami
```

2. Switch to the root user (requires the root password):

```
su -
```

3. Verify that the switch was successful:

```
whoami
```

(The output will now be `root`)

4. Display the system's hostname:

```
hostname
```

5. Exit the root session and return to the normal user:

```
exit
```

5.5 File and directory commands

Methodology:

Managing Files and Directories Linux relies on a hierarchical file system. The following commands manage navigation and file manipulation:

- `pwd`: Print Working Directory. Shows the absolute path of the current directory.
- `ls`: Lists directory contents. Use `ls -l` for a detailed listing.
- `cd`: Change Directory. Use `cd ..` to move to the parent directory.
- `mkdir` / `rmdir`: Make or remove empty directories.
- `cp`: Copies files. Use `-r` to copy directories recursively.
- `mv`: Moves or renames files and directories.
- `rm`: Removes files. Use `-r` for directories and `-f` to force removal.
- `cat`: Concatenates files and prints them to standard output.

Worked Example:

Directory Navigation and File Manipulation **Problem:** Create a directory named `Backup` copy a file named `config.cfg` into it rename the copied file to `old_config.cfg` and display its contents.
Solution:

1. Create the directory:

```
mkdir Backup
```

2. Copy the file into the new directory:

```
cp config.cfg Backup/
```

3. Navigate into the directory:

```
cd Backup
```

4. Rename the copied file:

```
mv config.cfg old_config.cfg
```

5. Display the file's contents:

```
cat old_config.cfg
```

5.6 Process management commands

Methodology:

Controlling Processes Processes are executing instances of programs. They can be monitored and controlled using these commands:

- `ps`: Reports a snapshot of current processes. `ps aux` lists all processes for all users.
- `top`: Provides a dynamic real-time view of running system processes.
- `kill`: Sends a signal to a specific Process ID (PID). `kill -9 PID` forcefully terminates a process.
- `bg`: Resumes a suspended job and runs it in the background.
- `fg`: Brings a background or suspended job to the foreground.
- `jobs`: Lists the active jobs running or suspended in the current terminal session.

Worked Example:

Terminating an Unresponsive Process **Problem:** A program named `loop.sh` is consuming too much CPU. Find its Process ID (PID) and terminate it forcefully.
Solution:

1. List all active processes and filter for the specific script using `grep`:

```
ps aux | grep loop.sh
```

2. Identify the PID from the output (typically the second column). Assume the PID is `2048`.
3. Send the SIGKILL signal to forcefully terminate the process:

```
kill -9 2048
```

4. Verify termination by listing the processes again.

5.7 Shell scripting basics

Methodology:

Writing and Executing a Shell Script A shell script automates sequences of commands.

1. **Shebang:** Start the script with `#!/bin/bash` to specify the Bash interpreter.
2. **Variables:** Define variables without spaces around the equal sign: `NAME="John"`. Access the value using `$NAME`.
3. **Input and Output:** Use the `echo` command to print text and the `read` command to capture user input.
4. **Permissions:** Scripts must be made executable using the `chmod` command before running them directly.

Worked Example:

Basic Interactive Shell Script **Problem:** Write a shell script that prompts the user for their name and age and then prints a summary message.
Solution:

1. Create a file named `info.sh` and add the script code:

```
#!/bin/bash
echo "Enter your name:"
read name
echo "Enter your age:"
read age
echo "Summary: Your name is $name and you are $age years old."
```

2. Make the script executable:

```
chmod +x info.sh
```

3. Execute the script from the current directory:

```
./info.sh
```

5.8 Control structures in shell scripting

Methodology:

Conditional and Loop Statements Shell scripts utilize control structures to execute logic based on conditions and to iterate over data.

- **If-Else Statement:** Evaluates a condition inside brackets `[ ]`.

```
if [ condition ]; then
    # executed if true
else
    # executed if false
fi
```

- **For Loop:** Iterates over a sequence or list of items.

```
for var in list; do
    # iterative commands
done
```

- **While Loop:** Continues execution as long as the condition is true.

```
while [ condition ]; do
    # commands
done
```

- **Case Statement:** A cleaner alternative to multiple if-elif statements.

```
case $variable in
    pattern1) commands ;;
    pattern2) commands ;;
esac
```

Worked Example:

Checking Number Parity using If-Else **Problem:** Write a shell script to determine whether a given integer is even or odd.
Solution:

1. Create the script `parity.sh`:

```
#!/bin/bash
echo "Enter an integer:"
read num

if [ $((num % 2)) -eq 0 ]; then
    echo "$num is an Even number."
else
    echo "$num is an Odd number."
fi
```

2. Execute the script. If the user enters `10`, the condition `$((10 % 2)) -eq 0` evaluates to true, outputting `10 is an Even number.`

Worked Example:

Printing a Multiplication Table using a For Loop **Problem:** Write a shell script to print the multiplication table of a user-provided number using a for loop.
Solution:

1. Create the script `table.sh`:

```
#!/bin/bash
echo "Enter a number for the multiplication table:"
read n
echo "Multiplication Table for $n:"
for i in {1..10}; do
    res=$((n * i))
    echo "$n x $i = $res"
done
```

2. Execute the script and enter `5`. The loop iterates `i` from 1 to 10, computing `res` and printing lines such as `5 x 1 = 5`.

Worked Example:

Countdown using a While Loop **Problem:** Write a shell script that counts down from 5 to 1 using a while loop.
Solution:

1. Create the script `countdown.sh`:

```
#!/bin/bash
counter=5
while [ $counter -gt 0 ]; do
    echo "Countdown: $counter"
    counter=$((counter - 1))
done
echo "Liftoff!"
```

2. Run the script. It prints 5, 4, 3, 2, 1 sequentially, ending with "Liftoff!".

Worked Example:

Menu Selection using Case Statement **Problem:** Write a shell script that displays a menu and prints a message based on the user's choice using a case statement.
Solution:

1. Create the script `menu.sh`:

```
#!/bin/bash
echo "Select an option: a or b"
read choice

case $choice in
```

```
        a)
            echo "You selected Option A."
            ;;
        b)
            echo "You selected Option B."
            ;;
        *)
            echo "Invalid selection."
            ;;
    esac
```

2. Run the script. If the user types `a`, the first pattern matches and executes the corresponding echo command.

Appendix: Key Formulae

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

CPU Scheduling

These formulae are fundamental to evaluating the performance of various CPU scheduling algorithms.

- **Turnaround Time (TAT):** The total time taken between the submission of a process and its completion.

  $$TAT = CT - AT$$

  Where CT is the Completion Time and AT is the Arrival Time.

- **Waiting Time (WT):** The total time a process spends waiting in the ready queue.

  $$WT = TAT - BT$$

  Where BT is the Burst Time (execution time) of the process.

Deadlock Avoidance (Banker's Algorithm)

These relationships are used in resource allocation and safety state calculations to prevent deadlocks.

- **Need Matrix Calculation:** Determines the remaining resources a process might request.

  $$Need[i, j] = Max[i, j] - Allocation[i, j]$$

  Where Max is the maximum resource requirement and $Allocation$ is the currently allocated resources for process i and resource type j.

- **Work Update:** During the safety check, once a process finishes, its allocated resources are added back to the available pool ($Work$).

  $$Work = Work + Allocation_i$$

- **Process Completion Flag:** Indicates that process i can safely finish.

  $$\texttt{Finish}[i] = \texttt{true}$$

Memory Management

Mathematical relationships used in paging and memory allocation schemes.

- **Number of Elements / Pages (N):**

  $$N = B/P$$

  Where B is the total block or memory size and P is the size of a single page or frame.

- **Page Number Calculation (P):** Used to determine the page number from a given logical address or offset.

  $$P = \lfloor N/R \rfloor$$

  Where N is the target address and R is the size of the page/region.

File System and Protection

Formal notation used to define access control and capabilities within a system.

- **Access Control Domain (D):** A set of access rights for various objects (like files).

  $$D_i = \{\langle F_x, \{rights\}\rangle, \dots\}$$

  For example, a domain defining read/write access to files:

  $$D_1 = \{\langle F_1, \{read, write\}\rangle, \langle F_2, \{read\}\rangle\}$$