OPERATING SYSTEMS

Compiled by

Dr.Vijay Chandra Jadala

OPERATING SYSTEM

Dr. J. Vijaya Chandra
Dr. V. Naresh
Dr.P.V.R.D.Prasad
Dr.Veeramallu
Dr.B.Vijay Kumar
M.Srikanth
Keerthi Samhitha
B.Pravalika
S.N.V.Jyotshna Kosuru
Y.Bhagya Laxmi

ISBN: XXXXXXXXXXXXXX

Printed @ KLEF

DEDICATION

To my dear friends,

Your encouragement, countless late-night brainstorming sessions, and constructive feedback have been invaluable in shaping the pages of this book. Thank you for believing in me and lending your hearts to my characters and stories.

I humbly offer you this work with the hope that it brings you joy, inspiration, and a momentary escape from reality. Your support and interest in my writing have motivated me to keep putting pen to paper. Without you, the words on these pages would hold little meaning.

Phone:
Email:

TABLE OF CONTENTS

Foreword **11**
Preface **13**
CHAPTER-1 INTRODUCTION TO OPERATING SYSTEMS **17**
Key Operating System Concepts: 18
Common Types Of Operating Systems: 20
History Of Operating Systems 23
Operating System Structure 25
Kernel 25
Types of Kernels 27
CHAPTER-2 UNIX AND SYSTEM CALLS **33**
History of UNIX: 33
System Calls: 34
Important System Calls Used In Os 34
System Call Implementation: 36
Types of System Calls: 37
CHAPTER -3 Process Virtualization **43**
Process Management 43
Process Creation: 45
Process States: 46
API (Application Programming Interface) 47
CHAPTER-4 CPU Scheduling **52**
Scheduling Criteria 52
First-Come, First-Served Scheduling (FCFS) 55
Shortest Job First Scheduling (SJF) 56
Shortest Remaining Time First (SRTF) 57
Priority Scheduling 59
Multilevel Queue Scheduling 60
Multiple Queue Scheduling 61

Multilevel Feedback Queue Scheduling 62
Multiple Processor Scheduling 63
Lottery Scheduling 65

CHAPTER -5 Memory Virtualization 68
Basics of Memory Management 68
Memory API Types of Memory 75
Virtual Memory 78

CHAPTER-6: Introduction To Paging 80
Paging 80
TLB Cache 82
Page Table Design 83
Demand Paging 85
Page Replacement Algorithms 89
Swapping 91
Splitting and Coalescing 94
Thrashing in Operating System 95

CHAPTER-7-Concurrency and Threads 98
Multithreading Models 99
User Threads 99
A) Many-To-One Model 99
B) One-To-One Model 100
C)Many-To-Many Model 101
Thread Libraries 102
Type of Concurrency Processing In Operating System 104

CHAPTER-8: Deadlock Prevention, Detection and Avoidance.... 105
8.1 Introduction 105
8.2 Resource-Allocation Graph 106
8.3 Methods for Handling the Dead Lock 108
8.4 Detecting The Dead Lock 110
Example: Banker’s algorithm 111
8.5 Preventing The Deadlock 113
8.6 Dead lock avoidance 115

8.7 Deadlock recovery 116
Deadlock Avoidance Algorithms: 125
1.Resource-Allocation-Graph Algorithm: 125
2.Banker's Algorithm: 127
Safety Algorithm: 128
Resource-Request Algorithm: 128
Deadlock Detection: 131

CHAPTER 9: Process Synchronization 137
Race Condition: 137
Critical Section: 138
Critical Section Problem in Operating System 139
Mutual Exclusion 140
Solutions for Critical Section Problem 142
Peterson's Solution 142
Classical Synchronization Problem: 144
Dining Philosopher Problem 147
Reader Writer Problem: 149
Semaphores 150
Types of Semaphores 151
Advantages of Semaphores 152
Disadvantages of Semaphores 153

CHAPTER-10 Disk Scheduling Algorithms 155
First-Come-First-Served (FCFS): 156
Shortest Seek Time First (SSTF) 158
SCAN: 160
C-SCAN (Circular SCAN) 163
LOOK 165
C-LOOK (Circular LOOK) 168

CHAPTER-11: Device Management 171
System Architecture 171
Hard Disk Drives 174
A Simple Disk Drive 174

Disk Scheduling 176
Redundant Arrays of Inexpensive Disks 177
Interface and Raid Internals 178
Fault Model 179
Evaluating Raid Performance 180

CHAPTER-12: File Management 183
1. Contiguous Allocation 183
2. Linked List Allocation 184
3. Indexed Allocation 185
File: 186
Inode: 186
Directory: 187
1.Single-Level Directory – 187
Two-Level Directory 188
3.Tree-Structured Directory 189
4.Acyclicgraph Directory 190
5.General graph directory Structure – 191
Disk Failure Modes: 193

CHAPTER-13: Data Integrity and Protection 199
CHAPTER-14: Real Time Operating Systems 205
Real-Time Operating Systems (RTOS) 205
REAL-TIME SCHEDULING 206

FOREWORD

It is with immense delight and admiration that I introduce this exceptional work, "Operating Systems: stands as the bridge between human intention and machine execution." As an industry expert and a colleague of the authors, **Dr. J. Vijaya Chandra**, I had the privilege of witnessing the inception and evolution of this comprehensive book.

During the course of our professional interactions, I have come to deeply appreciate, **Dr. V. Naresh**, profound knowledge and passion for operating systems. His dedication to research, coupled with his innovative approach to problem-solving, has been nothing short of inspiring. As I delved into the pages of this book, it became evident that **Dr.S.Kavitha, Dr.P.V.R.D.Prasad, Dr.Veeramallu** expertise has been distilled into a masterful guide for all enthusiasts of computer science.

“Operating Systems" takes readers on a remarkable journey through the intricate world of modern computing. From process and memory management to virtualization and security, each chapter offers a meticulous exploration of key concepts and cutting-edge technologies. **Dr.B.Vijay Kumar, Mr. M.Srikanth** ability to explain complex topics in a clear and accessible manner ensures that this book will appeal to both seasoned professionals and students alike.

Throughout the writing process, **Keerthi Samhitha, Y.Bhagya Laxmi, B.Pravalika** unwavering commitment to delivering accurate and up-to-date information was evident. His attention to detail and thorough research are a testament to his dedication to the field of operating systems.

As someone who has worked alongside **S.N.V.Jyotshna Kosuru** on various projects, I have witnessed his genuine desire to share knowledge and elevate the collective understanding of operating systems. This book is a manifestation of that desire, and I am confident that it will become an indispensable resource for practitioners, researchers, and educators alike.

I extend my heartfelt congratulations to **BAKKALA SANTHA KUMAR** for producing a work that not only showcases the latest advancements in operating systems but also in stills in readers a deep appreciation for the limitless possibilities of computer science.

To all those embarking on this enlightening journey through the world of operating systems, I invite you to immerse yourself in the wealth of knowledge and insights that “Operating Systems" has to offer. Prepare to be captivated and inspired by the brilliance of this exceptional author.

Phone:
Email:

PREFACE

In the realm of modern computing, the operating system stands as a cornerstone of technological advancement. Like the conductor of a symphony, it orchestrates the harmonious interaction between hardware and software, allowing the digital symphony of our lives to play out. This preface delves into the essence of operating systems, unraveling the layers of complexity that lie beneath the surface.

Imagine a bustling metropolis, with a myriad of tasks and activities occurring simultaneously. Streets, buildings, and infrastructure provide the framework for this bustling ecosystem. Similarly, an operating system lays down the infrastructure upon which applications and processes thrive. It manages the allocation of resources, from the memory that stores our thoughts to the processors that bring them to life. This invisible yet omnipresent force ensures fair allocation, preventing resource gridlock and enabling efficient multitasking.

At its core, the operating system is an enabler of dreams, a facilitator of innovation. It provides a canvas upon which developers paint their creations. Without the operating system, the world of software would be a disjointed collection of tools, devoid of the cohesion needed to power the technologies we rely on daily.

The user interface, a window into this intricate world, bridges the gap between humans and machines. Whether it's a graphical interface with colorful icons or a command-line interface with its rhythmic dance of text, the user interface brings the digital realm closer to our understanding. It transforms complex operations into accessible actions, making computing an experience that transcends barriers.

But with great power comes great responsibility. Security, a paramount concern in our interconnected age, is an integral part of the operating system's role. It guards against digital intruders, fortifying the walls around our data and digital identities. The operating system's vigilance ensures that our personal and sensitive information remains in safe hands.

Through failures and successes, the operating system stands resilient. It handles errors gracefully, recovering from unexpected disruptions and shielding users from the undercurrents of complexity. Just as a ship's captain steers through stormy waters, the operating system guides us through the ever-changing landscape of technology.

As we journey deeper into the digital age, the operating system continues to evolve. From humble beginnings of command-line interfaces, it has morphed into sophisticated graphical environments, virtual realms, and platforms for innovation. The operating system not only adapts to our needs but shapes them, ushering us into new frontiers of possibility.

In this preface, we embark on a voyage through the heart of the operating system. We peel back the layers, unravel the intricacies, and appreciate the role it plays in our digital existence. Just as an explorer maps uncharted territories, we seek to navigate the terrain of operating systems, shedding light on the unsung hero that underpins our technological world. Happy learning!

CHAPTER-1: Introduction to Operating Systems: Operating System Functionalities, Types of Operating Systems, Computer Architecture support to Operating Systems

CHAPTER-2: 2. UNIX and System Calls

CHAPTER-3: Process Virtualization: Processes, Process API code, Direct Execution,

CHAPTER-4: CPU Scheduling: Multi-level Feedback, Lottery Scheduling code, Multiprocessor Scheduling.

CHAPTER-5: Memory Virtualization: Address Spaces, Memory API, Address Translation, Segmentation, Free Space Management, **CHAPTER-6:Introduction to Paging**, Translation Look Aside Buffer, Swapping, Demand Paging, Thrashing, Page replacement algorithms.

CHAPTER-7: Concurrency and Threads

CHAPTER-8: Deadlock: Prevention, Detection and Avoidance Persistence:
CHAPTER-9: Process Synchronization
CHAPTER-10: Disk Scheduling Algorithms
CHAPTER-11: Device Management -I/O Devices, Hard Disk Drives, Redundant Disk Arrays (RAID),
CHAPTER-12: Files and Directories, File System Implementation, Distributed systems, Data Integrity and Protection,
CHAPTER-13: Data Integrity and Protection
CHAPTER-12: Real Time Operating Systems

[**Dr. J. Vijaya Chandra**]

Phone:
Email:

CHAPTER-1 INTRODUCTION TO OPERATING SYSTEMS

INTRODUCTION

An **operating system (OS)** is system software that manages computer hardware and software resources and provides common services for computer programs. The operating system is a vital part of the system software in a computer system; operating systems are an essential part of modern computing. They provide the foundation for all other software, and they make it possible for us to use our computers in a productive and efficient way.

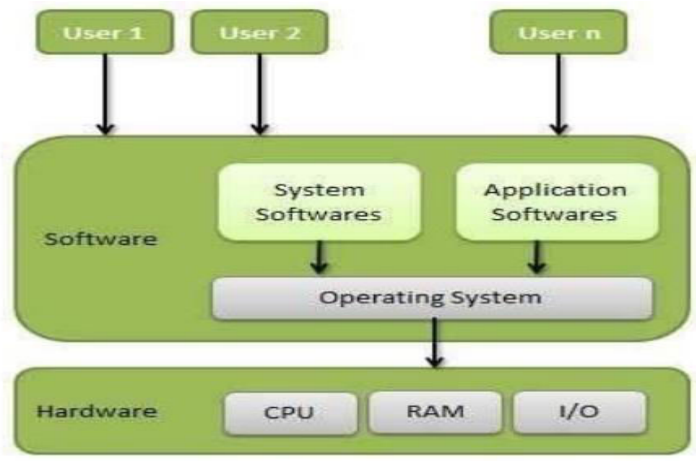


Figure 1. A computer system

A computer system consists of hardware, system programs, and application programs.

Operating system is an interface between user and Hardware, concepts are fundamental principles that fortify the design, functionality, and management of operating systems (OS). These concepts provide a framework for understanding how an operating system works and how it interacts with hardware, software, and users.

Key Operating System Concepts:

Process Management: The OS manages processes, which are executing instances of a program. It allocates CPU time, memory, and other resources to processes, schedules their execution, and provides mechanisms for inter-process communication and synchronization.

Memory Management: This concept deals with the allocation and management of computer memory. The OS is responsible for ensuring that each process has enough memory to execute without interfering with other processes, and it uses techniques like virtual memory to efficiently manage memory space.

File Systems: File systems provide a structured way of organizing and storing data on storage devices such as hard drives. The OS manages file operations like reading, writing, and deleting files and ensures data integrity and security.

Device Management: The OS handles interactions with hardware devices such as printers, keyboards, and disks. It provides device drivers to communicate with hardware and manages input/output (I/O) operations.

CPU Scheduling: As multiple processes compete for CPU time, the OS implements scheduling algorithms to determine which process gets access to the CPU and when. Efficient scheduling improves overall system performance and responsiveness.

Inter-Process Communication (IPC): Operating systems facilitate communication and data sharing between processes using various IPC mechanisms like shared memory, message passing, and semaphores.

Security and Protection: OS concepts include mechanisms to protect system resources and data from unauthorized access, ensuring the security and integrity of the system.

Virtualization: This concept allows multiple virtual machines or containers to run on the same physical hardware, enabling better resource utilization and isolation between applications.

Distributed Systems: Operating systems for distributed environments manage multiple interconnected computers, providing transparent access to resources across the network.

System Calls: These are interfaces that allow user-level applications to request services from the operating system, such as opening files, creating processes, and allocating memory.

Understanding these operating system concepts is essential for developers, system administrators, and anyone working with computers, as it forms the foundation for efficiently using, managing, and troubleshooting operating systems.

The Operating System as a Resource Manager

An operating system (OS) is a software program that manages computer hardware and software resources and provides common services for computer programs. One of the most important tasks of an OS is resource management.

Resources in a computer system can be divided into two main categories:

- Hardware resources: These include the CPU, memory, storage devices, and input/output (I/O) devices.
- Software resources: These include files, processes, and threads.

The OS is responsible for managing these resources in a way that ensures that they are used efficiently and fairly. This includes:

- Allocating resources to programs as needed.
- Ensuring that no one program uses too many resources.
- Preventing programs from interfering with each other.
- Recovering resources that are no longer being used.

The OS uses a variety of techniques to manage resources, such as:

- Scheduling: The OS schedules the execution of programs so that they all get a fair share of the CPU time.
- Memory management: The OS manages the memory in the computer system so that it is used efficiently.
- File management: The OS manages the files in the computer system so that they can be accessed and used by programs.

- Process management: The OS manages the processes in the computer system so that they can run concurrently and without interfering with each other.

Resource management is a complex task, but it is essential for the smooth operation of a computer system. The OS plays a vital role in resource management, and it is responsible for ensuring that resources are used efficiently and fairly.

COMMON TYPES OF OPERATING SYSTEMS:

- **Single-User Operating System:** A single-user operating system is designed to support only one user at a time. It is commonly used on personal computers and workstations where a single user interacts with the system at any given time.
- **Multi-User Operating System:** A multi-user operating system allows multiple users to access and use the system simultaneously. Each user has their own account and can run their processes independently.
- **Batch Processing Operating System:** Batch processing operating systems execute tasks in batches without direct user intervention. Users submit jobs as batches, and the system processes them in sequence without requiring real-time interaction.
- **Real-Time Operating System (RTOS):** RTOS is designed to meet strict timing constraints and provide predictable response times for real-time applications. It is commonly used in embedded systems, industrial automation, and control systems.
- **Time-Sharing Operating System (Multi-tasking OS)**:Time-sharing operating systems enable multiple users or processes to share the CPU's time simultaneously. The CPU switches rapidly between tasks, giving the illusion of concurrent execution.
- **Distributed Operating System:** A distributed operating system runs on a network of interconnected computers and allows them to work together as a single system. Distributed OS provides

transparency to users and applications, making the network appear as a single resource.

- **Network Operating System (NOS)**: A network operating system is specifically designed for managing and coordinating network resources and services. It facilitates file sharing, printer sharing, and centralized user administration in a networked environment.
- **Mobile Operating System:** Mobile operating systems are designed for smartphones, tablets, and other mobile devices. They are optimized for touch-based input and mobile-specific hardware features.
- **Standalone Operating System:** A standalone operating system is designed to work independently on a single computer or device without relying on network resources or connections.
- **Embedded Operating System:** Embedded operating systems are tailored for use in embedded systems with limited resources. They often run on specialized hardware and focus on specific tasks or functions.
- **Virtualization Operating System**: Virtualization OS runs on virtual machines and enables multiple virtualized operating systems to run on a single physical machine, allowing efficient resource utilization and isolation between applications.
- **Hybrid Operating System:** A hybrid operating system combines the characteristics of multiple types of operating systems. For example, modern desktop operating systems often have elements of single-user, multi-user, and time-sharing systems.

The type of operating system used depends on the intended use case, hardware platform, and specific requirements of the applications and users. Different types of operating systems serve various purposes and cater to diverse computing environments.

The Operating system is responsible for:

- **Hardware management**: The operating system manages the computer's hardware resources, such as the CPU, memory, and storage devices. It allocates these resources to programs as needed, and ensures that they are used efficiently.
- **Software management**: The operating system manages software resources, such as files and processes. It provides a way for programs to access files and other resources, and it ensures that programs do not interfere with each other.
- **User interface**: The operating system provides a user interface (UI) that allows users to interact with the computer. The UI can be graphical (GUI) or text-based (TUI).
- **Device drivers**: The operating system provides device drivers for hardware devices, such as printers, scanners, and network cards. Device drivers allow programs to access these devices.

There are many different types of operating systems, each with its own strengths and weaknesses. Some of the most popular operating systems include:

- **Windows**: Windows is the most popular operating system in the world. It is a graphical operating system that is used by millions of people on personal computers, laptops, and tablets.
- **macOS**: macOS is the operating system for Apple's Macintosh computers. It is a graphical operating system that is known for its user-friendliness and stability.
- **Linux:** Linux is a free and open-source operating system that is used by millions of people around the world. It is a powerful operating system that is used for both personal computers and servers.
- **Android:** Android is a mobile operating system that is used by billions of people on smartphones and tablets. It is a powerful and flexible operating system that is known for its customization options.

HISTORY OF OPERATING SYSTEMS

The history of operating systems (OS) is a fascinating journey that spans several decades. Let's take a brief look at the key milestones in the development of operating systems:

- **1940s-1950s:** Early Computers The earliest electronic computers, such as ENIAC and UNIVAC, were programmed using machine code directly. There were no formal operating systems, and tasks were often tedious and time-consuming.
- **1950s-1960s:** Batch Processing Systems As computers became more complex and capable, the concept of batch processing emerged. Users would submit jobs on punched cards, and the computer would process them in batches. Early operating systems like GM-NAA I/O and Fortran Monitor System (FMS) facilitated this process.
- **1960s-1970s:** Time-Sharing Systems and Multiprogramming The introduction of time-sharing systems allowed multiple users to interact with a computer simultaneously. Operating systems like Compatible Time-Sharing System (CTSS) and Multics were developed during this era. Additionally, multiprogramming techniques were implemented to keep the CPU busy by switching between multiple jobs quickly.
- **Late 1960s-1970s:** Unix Unix, developed at Bell Labs by Ken Thompson, Dennis Ritchie, and others, was a groundbreaking operating system introduced in the late 1960s. Unix's design principles emphasized simplicity, modularity, and portability, which made it highly influential and led to its widespread adoption.
- **1980s:** Personal Computers and GUIs The 1980s saw the rise of personal computers. Operating systems like MS-DOS (Microsoft Disk Operating System) and Apple's Mac OS (later known as macOS) gained popularity. Graphical User Interfaces (GUIs) were introduced, making computers more user-friendly, with

examples such as Apple's Macintosh System Software and Microsoft Windows.

- **1990s:** Windows and Linux Microsoft's Windows operating system became dominant in the 1990s, particularly with the release of Windows 95 and subsequent versions. Meanwhile, Linux, an open-source OS inspired by Unix, gained traction and became popular among developers and enthusiasts.
- **2000s:** Mobile OS and Modern Operating Systems The 2000s marked a significant shift towards mobile devices. Operating systems like Palm OS, Symbian, BlackBerry OS, and eventually iOS and Android, dominated the mobile market. In the realm of traditional desktop operating systems, Windows XP, Windows 7, macOS, and various Linux distributions gained prominence.
- **2010s-Present:** Cloud Computing and IoT The 2010s witnessed a surge in cloud computing, where services and applications are hosted on remote servers accessible over the internet. Operating systems like Chrome OS, designed around cloud-based applications, emerged. The Internet of Things (IoT) also led to the development of specialized operating systems for embedded devices.

Current Trends: AI Integration and Virtualization In recent years, there has been a growing trend of integrating artificial intelligence (AI) into operating systems to enhance functionality and user experience. Additionally, virtualization technologies, such as containerization and virtual machines, have become widespread for efficient resource utilization and application deployment.

The history of operating systems is a dynamic and ongoing process, constantly evolving to meet the needs of advancing technology and user demands. It continues to shape the way we interact with computers and the digital world around us

Here are some of the most important milestones in the history of operating systems:

1956: GM-NAA I/O, the first operating system, is developed.

1965: Multics, one of the first multiprogramming OSes, is developed.

1969: Unix, one of the most influential OSes in history, is developed.

1981: Microsoft releases the first version of MS-DOS, which becomes the dominant OS on personal computers.

1984: Apple releases the Macintosh, which is the first personal computer to ship with a GUI.

1991: Linus Torvalds releases the first version of Linux.

2001: Microsoft releases Windows XP, one of the most popular OSes of all time.

2007: Apple releases the iPhone, which is the first smartphone to run a modern OS.

2011: Android, the most popular mobile OS in the world, is released.

OPERATING SYSTEM STRUCTURE

The structure of an operating system refers to its organization and the way its components are designed and interact with each other. Operating systems typically follow a layered or modular architecture to provide abstraction, flexibility, and ease of maintenance.

The key components of an operating system structure include:

KERNEL:

The kernel is the core component of the operating system, residing in privileged mode, directly interacting with hardware, and providing essential services to user programs. It manages system resources, such as memory, CPU, and devices, and enforces security and protection mechanisms. The kernel is responsible for process management, memory management, device drivers, and I/O operations.

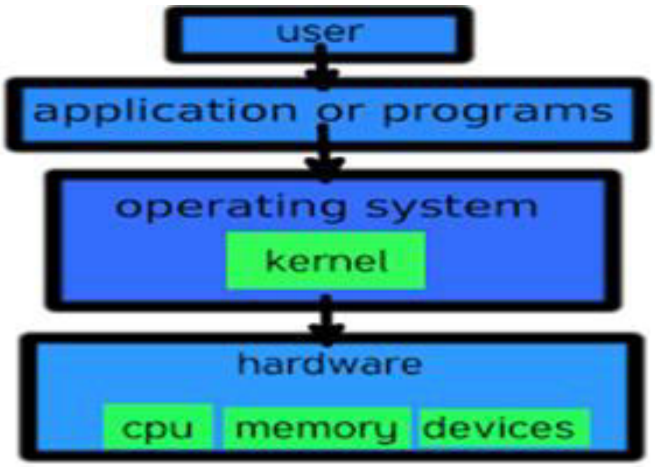


Fig 2: Kernel in operating system

The kernel is a critical component of an operating system that serves as the core or central part of the software. It operates in privileged mode with direct access to hardware and provides essential services and functionalities for the rest of the operating system and user-level applications. The kernel is responsible for managing system resources, ensuring security and protection, and providing an interface for user programs to interact with the underlying hardware.

Here are some key aspects of the kernel:

Core Functions:

- **Process Management:** The kernel manages processes, including their creation, scheduling, termination, and context switching.
- **Memory Management:** It allocates and de-allocates memory for processes and handles memory protection and virtual memory.
- **Device Management:** The kernel communicates with hardware devices through device drivers to facilitate I/O operations.
- **File System Management:** It handles file operations and provides an interface for user programs to access and manage files on storage devices.

Privileged Mode:

The kernel operates in privileged mode (also known as supervisor or kernel mode), which allows it to execute privileged instructions and access hardware resources directly.

User programs run in user mode, which restricts them from performing privileged operations directly.

System Calls:

The kernel provides an interface called system calls through which user-level applications can request services and functionalities from the kernel. When a user program needs to perform a privileged operation (e.g., reading from a file, creating a new process), it makes a system call to the kernel, which then handles the request on its behalf.

Interrupt Handling:

The kernel manages interrupts generated by hardware devices, such as keyboard input, disk I/O, and network communication.

When an interrupt occurs, the kernel interrupts the current execution of a program, handles the interrupt, and resumes the program.

Process Synchronization and Protection:

The kernel ensures that processes are properly synchronized when accessing shared resources to avoid conflicts and data corruption.

It enforces protection mechanisms to prevent unauthorized access to system resources and ensure the security of the system.

Types of Kernels

Monolithic Kernel: - In a monolithic kernel, the kernel and operating system, both run in the same memory, and it is mainly used where security is not a major concern. The result of the monolithic kernel is fastly accessible. But in some situations, like if a device driver has a bug, then there may be chances of a whole system crash.

Microkernel: - A Microkernel is the derived version of the monolithic kernel. In microkernel, the kernel itself can do different jobs, and there is no requirement of an additional GUI.

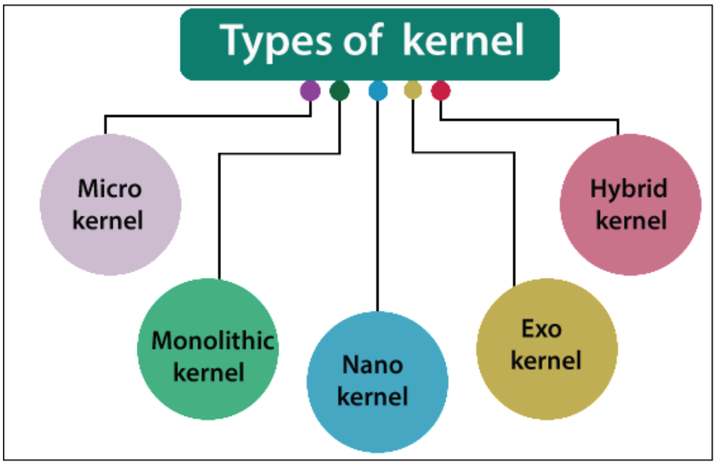


Fig 3: Types of kernels in operating system

Nano kernel: - Nano kernel is the small type of kernel which is responsible for hardware abstraction, but without system services. Nano kernel is used in those cases where most of the functions are set up outside.

Exo kernel: - Exo kernel is responsible for resource handling and process protection. It is used where you are testing out an in house project, and in up-gradation to an efficient kernel type.

Hybrid kernel: - Hybrid kernel is a mixture of microkernel and monolithic kernel. The Hybrid kernel is mostly used in Windows, Apple's macOS. Hybrid kernel moves out the driver and keeps the services of a system inside the kernel.

System Calls in OS:

System calls are an essential interface provided by the operating system that allows user-level applications to request services and access functionalities that are only available in privileged mode (kernel mode). They act as a bridge between user-space applications and the kernel, enabling user programs to interact with the underlying hardware and perform privileged operations. When a user program needs to perform a specific task that requires kernel-level privileges, it makes a system call to request the kernel to perform that task on its behalf.

Here's how the process of a system call typically works:

User Program Initiates a System Call:

A user-level application running in user mode requires certain services from the operating system, such as opening a file, creating a new process, or allocating memory. Since these operations need privileged access to hardware and resources, the user program cannot perform them directly.

Entering Kernel Mode:

To make a system call, the user program must transition from user mode to kernel mode, where the operating system's kernel resides. This transition is typically done through a special instruction or software interrupt provided by the processor.

System Call Number and Arguments:

The user program specifies the desired system call by providing a system call number or identifier. Additionally, any necessary arguments or parameters for the system call are passed to the kernel. These arguments convey the necessary information for the requested operation.

Kernel Service Routine:

The kernel maintains a table of system calls, each associated with a unique identifier or number. When the user program initiates a system call, the kernel looks up the corresponding entry in the system call table based on the system call number provided.

Execution of System Call:

Once the kernel has identified the appropriate system call and its arguments, it executes the corresponding kernel service routine that implements the requested functionality.

Kernel Processing and Return:

The kernel performs the privileged operation on behalf of the user program, using its unrestricted access to hardware and resources.

After the system call is completed, the kernel returns the result or status of the operation to the user program.

Returning to User Mode:

Once the system call is finished, the user program transitions back to user mode, and normal execution of the program resumes.

Examples of common system calls include:

File-related system calls: open(), read(), write(), close()

Process-related system calls: fork(), exec(), wait(), exit()

Memory-related system calls: malloc(), mmap(), munmap()

Network-related system calls: socket(), bind(), send(), recv()

System calls are essential for providing a secure and controlled environment for user programs to interact with the operating system and hardware, ensuring that sensitive operations are performed under the supervision and control of the kernel

Device Drivers in OS:

Device drivers are software components that allow the operating system to communicate with hardware devices, such as printers, disks, and network cards. Each device driver is specific to a particular hardware device and provides an abstraction layer for the kernel to access and control the device.

Device drivers are software components that facilitate communication between the operating system and hardware devices. They serve as intermediaries, enabling the operating system to interact with various peripherals, such as printers, keyboards, graphics cards, network adapters, storage devices, and more. Device drivers are crucial for the proper functioning of hardware devices within a computer system. Here are the key aspects of device drivers:

Purpose of Device Drivers:

Hardware devices have their unique interfaces and communication protocols. The operating system, which is a general-purpose software, needs device drivers to understand and interact with these devices correctly.

Device drivers provide an abstraction layer, shielding the operating system and user-level applications from the complexities of interacting directly with hardware.

Functions of Device Drivers:

Device Initialization: Device drivers are responsible for initializing the hardware devices during system startup or when the device is first detected.

Device Configuration: They configure the device settings and parameters to ensure it operates efficiently within the system.

Data Transfer: Device drivers manage the transfer of data between the operating system's memory and the hardware device. For example, they handle read and write operations to storage devices or network communication.

Interrupt Handling: Device drivers manage hardware interrupts generated by devices to notify the CPU of specific events or data availability.

Error Handling: They handle errors that may occur during device operation and report them to the operating system or user-level applications.

Overall, device drivers play a vital role in enabling the seamless interaction between operating systems and the diverse range of hardware devices used in modern computer systems.

CHAPTER-2 UNIX AND SYSTEM CALLS

HISTORY OF UNIX: -

Multiuser: - Multi user operating system means more than one user shares the same system resources (hard disk, memory, printer, application software etc.,) at the same time.

Multi-tasking: - Another highlight of UNIX is that it is Multitasking, implying that it can carry out more than one job at the same time. Depending on the priority the task, the operating system appropriately allots small time slots to each foreground and background task. Programming

Facility: - UNIX operating system provides shell. Shell works like a programming language. It provides commands and keywords. By running these two, user can prepare efficient program.

Portability: - One of the main reasons for the universal popularity of Unix is that it can be ported to almost any computer system, with only the bare minimum of adoptions to suit the given computer architecture.

Communication: - UNIX provides electronic mail. The communication may be within the network of a single main computer, or between two are more such computer networks. The user can easily exchange mail data, programs through such networks.

Security: - UNIX provides three levels of security to protect data. The first is provided by assigning passwords and login names to individual users ensuring that not anybody can come and have access to your work. At the file level, there are read, write, and execute permissions to each file which decide who can access a particular file, who can modify it and who can execute it. Lastly, there is file encryption. This utility encodes your file into an unreadable format, so that even if someone succeeds in opening it, your secrets are safe.

Open System: - The source code for the UNIX system and not just the executable cede, has been made available to users and programmers. Because of this many people have been able to adapt the UNIX system in different ways.

System Calls: - Programs interact with the kernel through approximately 100 system calls. System calls tell the kernel to carry out various tasks for the program, such as opening a file, writing a file, obtaining. information about a file, executing a program, terminating a process, changing the priority of a process and getting the time of day. Different implementations of Unix system have compatible system calls with each call having the same functionality.

SYSTEM CALLS:

The most common system calls used on Unix system calls, Unix-like, and other POSIX-compliant operating systems are open, read, write, close, wait, exec, fork, exit, and kill.

A system call is a function that a user program uses to ask the operating system for a particular service. User programmers can communicate with the operating system to request its services using the interface that is created by a system call.

System calls serve as the interface between an operating system and a process. System calls can typically be found as assembly language instructions. They are also covered in the manuals that the programmers working at the assembly level use. When a process in user mode needs access to a resource, system calls are typically generated. The resource is then requested from the kernel via a system call.

Important System Calls Used In Os

1. wait ()

In certain systems, a process must wait for another process to finish running before continuing. When a parent process creates a child process, the parent process's execution is suspended until the child process has finished running. With a wait() system call, the parent process is automatically suspended. Control returns to the parent process once the child process has completed its execution.

2. FORK ()

The fork system call in OS is used by processes to make copies of themselves. By using this system, the Call parent process can create a child process, and the parent process's execution will be halted while the child process runs.

3. EXEC ()

When an executable file replaces an earlier executable file within the context of an active process, this system call is executed. The original process identifier is still present even though a new process is not created; instead, the new process replaces the old one's stack, data, head, data, etc.

4. KILL ()

The OS uses the kill () system call to urge processes to terminate by sending them a signal. A kill system call can have different meanings and is not always used to end a process.

5. exit()

An exit system call is used when the programme must be stopped. When the exit system call is used, the resources that the process was using were released.

Why Do You Need System Calls in Operating System?

- System calls are necessary for reading and writing from files.
- System calls are necessary for a file system to add or remove files.
- New processes are created and managed using system calls.
- System calls are required for packet sending and receiving over network connections.
- A system call is required to access hardware devices like scanners and printers.

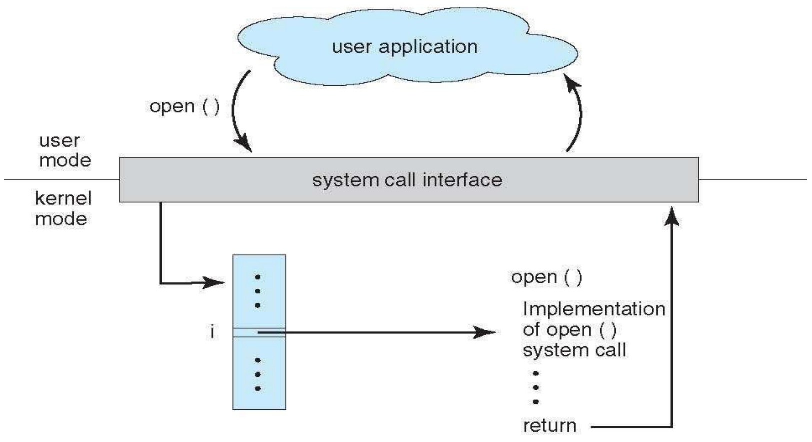
System Call Implementation:

Typically, a number is associated with each system call

- **System-call interface** maintains a table indexed according to these numbers.
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need not know a thing about how the system call is implemented.
- Just needs to obey the API and understand what the OS will do as a result call.
- Most details of OS interface hidden from programmer by API.
- Managed by run-time support library (set of functions built into libraries included with compiler)

System Call -- OS Relationship:

The handling of a user application invoking the open() system call



System Call Parameter Passing:

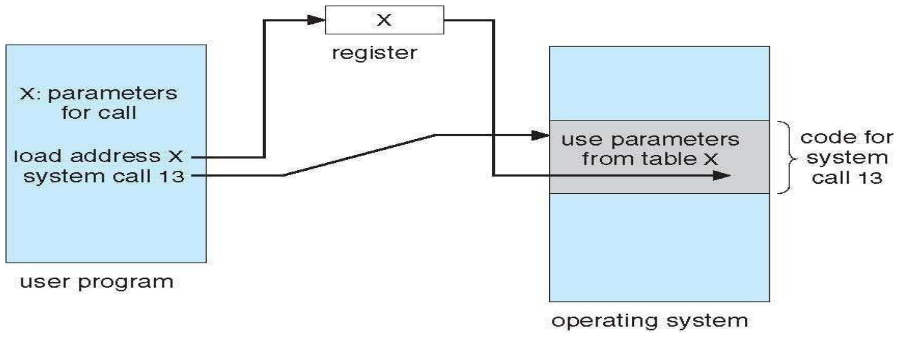
- Often, more information is required than simply identity of desired system call.
- Exact type and amount of information vary according to OS and call.

- **Three** general methods used to pass parameters to the OS.
- **Simplest:** pass the parameters in registers. In some cases, may be more parameters than registers.
- Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register. This approach taken by Linux and Solaris
- Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.
- Block and stack methods do not limit the number or length of parameters being passed.

Parameter Passing via Table:

x points to a block of parameters. x is loaded into a register

A system call is a way for a user program to interface with the operating system. The program requests several services, and the OS responds by invoking a series of system calls to satisfy the request. A system call can be written in assembly language or a high-level language like **C** or **Pascal**. System calls are predefined functions that the operating system may directly invoke if a high-level language is used.



TYPES OF SYSTEM CALLS:

- **Process control**
    - create process, terminate process.
    - end, abort.

- load, execute.
- get process attributes, set process attributes
- wait for time.
- wait event, signal event.
- allocate and free memory
- Dump memory if error
- **Debugger** for determining bugs, single step execution
- **Locks** for managing access to shared data between processes

➢ **File management**

- create file, delete file
- open, close file
- read, write, reposition
- get and set file attributes

➢ **Device management**

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

➢ **Information maintenance**

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

➢ **Communications**

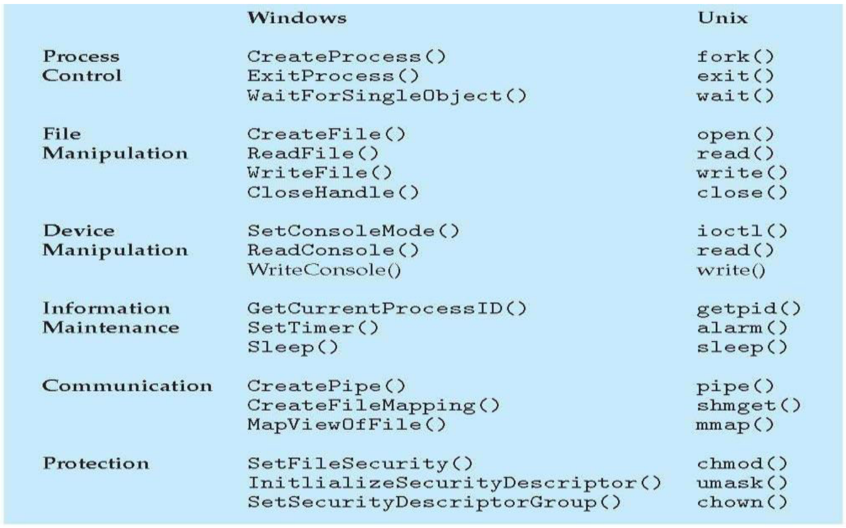
- create, delete communication connection.
- send, receive messages if **message passing model** to **host name** or **process name.**

➢ From **client** to **server**

- **Shared-memory model** create and gain access to memory regions
- transfer status information
- attach and detach remote devices

- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls



	Windows	Unix

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

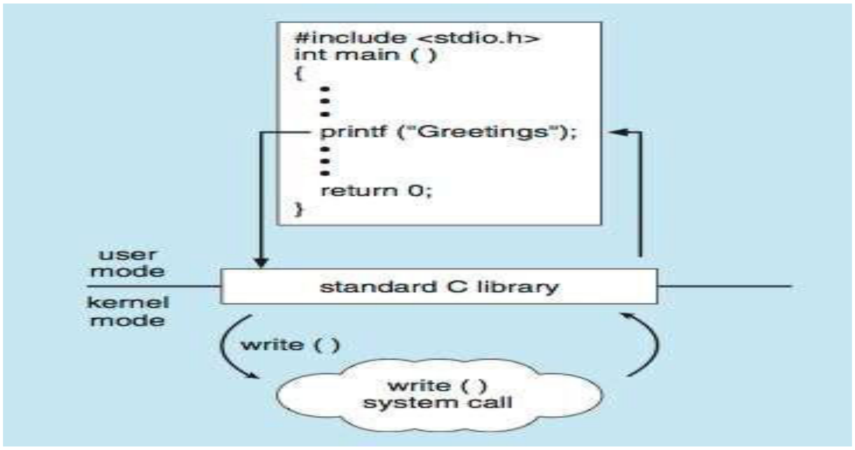
Information Maintenance	GetCurrentProcessID()
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

Protection	SetFileSecurity()
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

Example -- Standard C Library:

C program invoking printf() library call, which calls write() system call



```
#include <stdio.h>
int main ( )
{
   ⋮
   printf ("Greetings");
   ⋮
   return 0;
}
```

File Structure Related System Calls

- The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices.
- To a process then, all input and output operations are synchronous and un buffered.
- All input and output operations start by opening a file using either the "creat()" or "open()" system calls.
- These calls return a file descriptor that identifies the I/O channel.

File descriptors:

- Each UNIX process has 20 file descriptors at it disposal, numbered 0 through 19.

The first three are already opened when the process begins.

- 0: The standard input
- 1: The standard output
- 2: The standard error output

When the parent process forks a process, the child process inherits the file descriptors of the arent.

What is the Purpose of System Calls in OS?

The purpose of system calls serves as the interface between an operating system and a process. System calls can typically be found as assembly language instructions. They are also covered in the manuals that the programmers working at the assembly level use. When a process in user mode needs access to a resource, system calls are typically generated. The resource is then requested from the kernel via a system call.

System calls are required in in following circumstances:

- If a file system needs files to be created or deleted. A system call is also necessary for reading and writing from files.
- development and administration of new procedures.

- System calls are also required for network connections. This also applies to packet sending and receiving.
- A system call is necessary to access hardware such as a printer, scanner, etc.

When a system call is made, the process transitions from user mode to kernel mode. Once the system call has finished running, the control is returned to the user mode process. Sending the kernel, a trap signal causes it to read the system call code from the register and carry out the system call. Process control, file management system calls in OS, device management, information maintenance, and communication are the main types of system calls in OS. Several crucial system calls used by our computer system include wait(), fork(), exec(), kill(), and exit().

How to provide the illusion of many CPUs?

- **CPU virtualizing**
    - The OS can promote the i llusion that many virtual CPUs exist.
    - **Time sharing**: Running one process, then stopping it and running another**.**
    - The potential cost is performance.

CHAPTER -3 PROCESS VIRTUALIZATION

Process Management

This component deals with the creation, scheduling, and termination of processes. It includes the process control block (PCB) data structure for managing process information. The process management component ensures that processes share CPU time fairly and efficiently, and it handles process synchronization and communication**.**

Process management is a crucial aspect of operating systems that involves the creation, scheduling, termination, and coordination of processes. A process represents an executing instance of a computer program, and process management ensures that multiple processes can run concurrently and share system resources efficiently. Here are the key components and functions of process management:

Process Creation:

The operating system is responsible for creating new processes. This can occur in several ways, such as when a user starts an application or when a parent process spawns a child process. During process creation, the operating system allocates a unique process identifier (PID) to each process and sets up a separate address space for the new process, ensuring that each process runs independently.

Process States:

Processes can be in different states during their lifetime, including:

- **New:** The process is being created.
- **Ready:** The process is prepared to run, but the CPU is currently executing another process.
- **Running**: The process is currently being executed by the CPU.

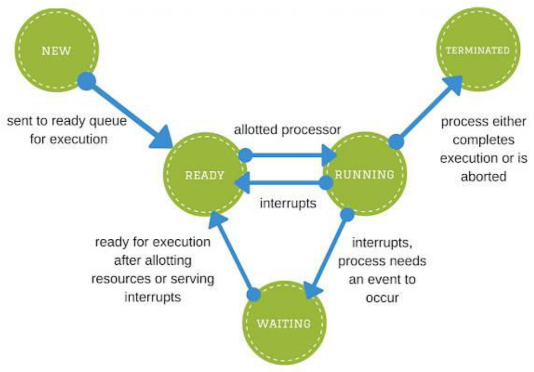

NEW
TERMINATED
sent to ready queue for execution
allotted processor
READY
RUNNING
interrupts
process either completes execution or is aborted
ready for execution after allotting resources or serving interrupts
interrupts, process needs an event to occur
WAITING

Fig 3.1: Process States

- **Blocked (Waiting):** The process is waiting for a specific event or resource (e.g., I/O completion) and cannot proceed until the event occurs or the resource becomes available.
- **Terminated:** The process has finished its execution and is being removed from the system.

The operating system maintains a data structure called the Process Control Block (PCB) for each process. The PCB contains essential information about the process, including process state, program counter, CPU registers, memory allocation, and more. The PCB enables the operating system to manage and track processes efficiently.

Process management is a fundamental function of operating systems, enabling efficient multitasking, resource sharing, and overall system stability. It ensures that multiple processes can execute concurrently without interfering with each other, creating a seamless user experience and efficient utilization of system resources

A Process is a running program. A process is basically a program in execution. The execution of a process must progress in a sequential procedure.

- **Comprising of a process:**
  - Memory (address space)
  - Instructions
  - Data section

- **Registers**
  - Program counter
  - Stack pointer
- **Process API:**
  - These APIs are available on any modern OS.
- **Create**
  - Create a new process to run a program
- **Destroy**
  - Halt a runaway process
- **Wait**
  - Wait for a process to stop running
- **Miscellaneous Control**
  - Some kind of method to suspend a process and then resume it
- **Status**
  - Get some status info about a process

Process Creation:

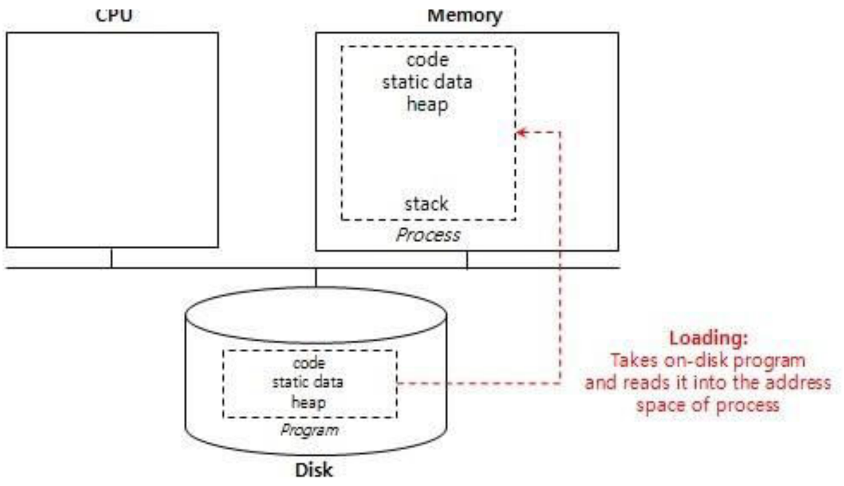
Load a program code into memory , into the address space of the process.

- Programs initially reside on disk in *executable format*.
- OS perform the loading process lazily.
- Loading pieces of code or data only as they are needed during program execution.

1. The program’s run-time **stack** is allocated.
   - Use the stack for *local variables*, *function parameters*, and *return address*.
   - Initialize the stack with arguments ¯ argc and the argv array of main() function
2. The program’s **heap** is created
   - Used for explicitly requested dynamically allocated data.

        - Program request such space by calling malloc() and free it by calling free().
3. The OS do some other initialization tasks. input/output (I/O) setup.
    - Each process by default has three open file descriptors.
    - Standard input, output and error
4. **Start the program** running at the entry point, namely main().
    - The OS *transfers control* of the CPU to the newly-created process.

Loading: From Program to Process:



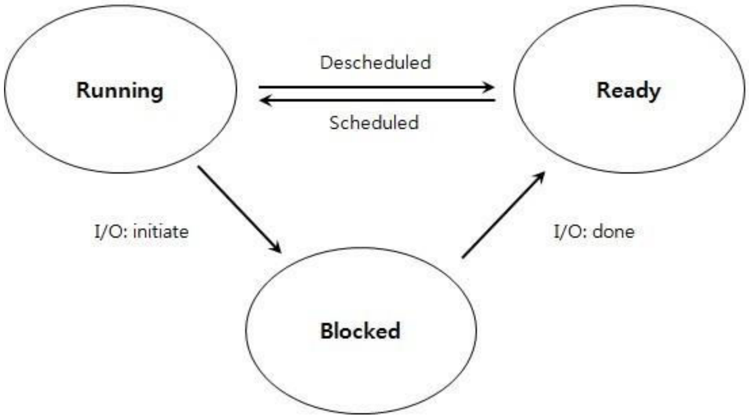
Process States:

A process can be one of three states.

- **Running:** A process is running on a processor.
- **Ready:** A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked:** A process has performed some kind of operation.

- When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition:



Data structures:

- The OS has some key data structures that track various relevant pieces of information.

Process list

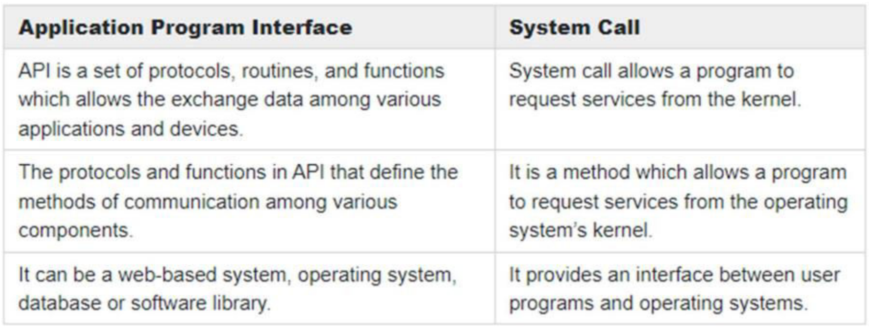
- Ready processes
- Blocked processes
- Current running process

Register context

- PCB(Process Control Block)
- A C-structure that contains information about each process.

API (Application Programming Interface) is used to establish connectivity among devices and applications. However, it is an interface which takes the requests from the user and informs the system about what must be done and returns the response back to the user.

Each process has a name; in most systems, that name is a number known as a **process ID (PID).**



Application Program Interface	System Call
API is a set of protocols, routines, and functions which allows the exchange data among various applications and devices.	System call allows a program to request services from the kernel.
The protocols and functions in API that define the methods of communication among various components.	It is a method which allows a program to request services from the operating system's kernel.
It can be a web-based system, operating system, database or software library.	It provides an interface between user programs and operating systems.

What API does the OS provide to user programs?

- API = Application Programming Interface= functions available to write user programs.
- API provided by OS is a set of "system calls" – System call is a function call into OS code that runs at a higher privilege level of the CPU.
- Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level – Some "blocking" system calls cause the process to be blocked and de-scheduled (e.g., read from disk)

The fork() system call is used to create a new process,

The fork() System Call:

- **Create a new process**
    - The newly-created process has its own copy of the **address space**, **registers**, and **PC**.

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value**:** creation of a child process was unsuccessful.

Zero**:** Returned to the newly created child process.

Positive value**:** Returned to parent or caller. The value contains process ID of newly created child process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}

```

Figure 5.1: **Calling `fork()` (`p1.c`)**

When you run this program (called `p1.c`), you'll see the following:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {              // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}

```

Figure 5.2: **Calling fork() And wait() (p2.c)**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {              // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}

```

Figure 5.3: **Calling fork(), wait(), And exec() (p3.c)**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {              // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}

```

Figure 5.3: **Calling fork(), wait(), And exec() (p3.c)**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4.c)**

CHAPTER-4 CPU SCHEDULING

Scheduling refers to the set of policies and mechanisms that an OS supports for determining the order of execution of the pending jobs and processes. A Scheduler is an operating system program (module) that selects the next job to be admitted for execution.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Scheduling Criteria

A Scheduler algorithm is evaluated against some widely accepted performance criteria.

1. **C.P.U. Utilization:** It is defined as average fraction of time during which CPU is busy, executing either user programs or system modules. The key idea is that if the CPU is busy all the time then the utilization factor of all the components of the system will also be high.
2. **Throughput:** It is defined as the average amount of work completed per unit time. One way to measure throughput is by means of the number of processes that are completed in a unit of time. Please note here that the higher is the number of processes, the more work is apparently being done by the system. Also note that the higher is the throughput better it is. But this approach is not very useful for comparison because this is dependent on the characteristics and the resource requirements of the process being executed. Thus, to compare the throughput of several scheduling algorithms it should feed the process with similar requirements.

3. **Turn around Time (TAT):** It is defined as the total time elapsed from the time the job is submitted (or process is created) to the time the job (or process) is completed. It is the sum of periods spend waiting to get into memory. Waiting in the ready queue, CPU time and I/O operations. So, we can write that – **Turn around Time (TAT) = (Process Finish Time – Process Arrival Time)**Also, note that the lower is the average turnaround time, better it is.
4. **Waiting Time (WT):** It is defined as the total time spent by the job (or process) while waiting in suspended state or ready state, in a multiprogramming environment. So, it is given as **Waiting Time (WT) = (Turn Around Time – Processing Time)**Please note that the lower is the average waiting time, better it is.
5. **Response Time (RT):** This parameter is usually considered for two systems – time sharing and real time operating system. However, its characteristics differ in these two systems. In time sharing system it may be defined as the interval from the time and last character of a command line of a program or transaction is entered to the time the last result appears on the terminal.

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Algorithms based on preemptive scheduling are: Round Robin (RR), Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

Non-preemptive Scheduling is used when a process terminates, or a process switch from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non- preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.

Algorithms based on non-preemptive scheduling are: Shortest Job First (SJF basically non preemptive) and Priority (non-preemptive version), etc.

Preemptive scheduling	Non-preemptive Scheduling
In non-preemptive scheduling, if once a process has been allocated CPU, then the CPU cannot be taken away from that process	In Pre-emptive scheduling, the CPU can be taken away before the completion of the process
No Preference is given when a higher priority job comes	It is useful when a higher priority job comes as here the CPU can be snatched from a lower priority process.
The treatment of all processes is fairer.	The treatment of all processes is not fairer as CPU snatching is done either due to time constraints or due to a higher priority process requests for its execution
It is cheaper scheduling method. For example: FCFS CPU Scheduling algorithm is non-preemptive.	It is costlier scheduling method. For example: Round-Robin

Key Differences Between Preemptive and Non-Preemptive Scheduling:

1. In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes

whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.

3. In Preemptive Scheduling, there is the overhead of switching the process from the ready state to running state, vise-verse and maintaining the ready queue. Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready .
4. In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority must wait for a long, and it may have to starve. in the non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive at, the ready queue, no matter what process is executing currently. Non- preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. Preemptive Scheduling must maintain the integrity of shared data that's why it is cost associative which is not the case with Non-preemptive Scheduling.

First-Come, First-Served Scheduling (FCFS)

It is the simplest of all the scheduling algorithms. **The basic Principle of this algorithm is to allocate the CPU in the order in which the process arrive.** Its implementation involves a ready queue that works in First-in First-out order. When the CPU is free, it is assigned to a process which is in front of the ready queue. Once this process goes into a running state, its PCB is removed from the queue. A FCFS CPU Scheduling algorithms is **non-preemptive** because the CPU has been allocated to a process that keeps the CPU busy until it is released. This usually results in poor performance. It is because of this non-preemption only that the component utilization rate is low and the system throughout is less. Also note here that the shorter jobs may suffer considerable turnaround delays and waiting times when CPU has been allocated to longer jobs.

Advantages of FCFS algorithms

1. It is simple algorithm and easy to implement.
2. It is suitable for Batch systems.

Disadvantages of FCFS algorithms

1. The average waiting time is not minimal. Under this scheme, the process that requests the CPU first is allocated the CPU first. When the process enters the ready queue, its Process Control Block (PCB) is linked which keeps the track of all the processes running and waiting. So, the average waiting time under the FCFS policy is very long.
2. It is not suitable for time sharing systems like Unix because in Unix, each user should get the CPU for an equal amount of time interval and this algorithm does not follow this principle.
3. A proper mix of CPU bound and I/O bound jobs is required to achieved good results from FCFS scheduling.

Shortest Job First Scheduling (SJF)

The basic principle of this algorithm is to allocate the CPU to the process with least CPU-burst time. The processes are available in the ready queue. CPU is always assigned to the process with least CPU burst time requirement. **Please note that if the two processes have same CPU burst time, then FCFS is used to break the tie.** This algorithm can be either preemptive or non-preemptive.

In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.

In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Advantages of SJF Scheduling

1. Used for long-term scheduling.
2. Reduces average waiting time.
3. Helpful for batch-type processing where runtimes are known in advance.
4. For long-term scheduling, we can obtain a burst time estimate from the job description.
5. It is necessary to predict the value of the next burst time for Short-Term Scheduling.
6. Optimal with regard to average turnaround time.
7. Since this algorithm gives minimum average waiting time so it is an optimal algorithm

Disadvantages of SJF Scheduling

1. It is necessary to know the job completion time beforehand as it is hard to predict.
2. Used for long-term scheduling in a batch system.
3. Can't implement this algorithm for CPU scheduling for the short term as we can't predict the length of the upcoming CPU burst. It is difficult to know the length of next CPU burst time.
4. Cause extremely long turnaround times or starvation.
5. Knowledge about the runtime length of a process is necessary.
6. Recording the elapsed time results in increased overhead on the processor.
7. It cannot be implemented at the level of short-term CPU Scheduling. Big jobs are waiting for CPU. This results in aging problem.

Shortest Remaining Time First (SRTF)

This algorithm is a preemptive scheduling algorithm. The short-term scheduler always chooses the process that has the shortest remaining processing time. When a new process joins the ready queue, the short-term scheduler compares the remaining time of executing process and new process. If the new process has the least CPU burst time, the scheduler selects that job and allocates CPU else it continues with the old process.

Round Robin (RR) Scheduling algorithm

This is one of the oldest, simplest and widely used algorithms. The basic principle of this algorithm is as follows

Principle – "If there are n-processes in a ready queue and the time quantum is 'q' time intervals, then each process get '1/n' of the CPU time in the chunks of at most 'q' units of time. Each process will have to wait for (n-1) x q time units until its next time quantum comes in"

RR algorithm is like FCFS algorithm but not it is preemptive FCFS scheduling. The round robin scheduling algorithm is primarily used in time-sharing and a multi-user system where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. The preemption takes place after a fixed interval of time called the **time quantum or time slice**. The CPU time is divided into slices. Each process is allocated to small time (usually 10-100 milliseconds) while it executes.

Please note that no process can run for more than one time slice because some other processes are awaiting in the ready queue. If the time quantum expires then process goes to the end of the ready queue to wait for the next allocation. **Also note that if the running process releases a control to OS due to some I/O, then another process is scheduled to run.**

RR Scheduling algorithm utilizes the system resources uniformly. Small process may be executed in a single time-slice giving good response time whereas long processes may require several time slices and thus be forced to pass through ready queue a few times before completion.

The Performance of this **Pure Preemptive algorithm** depends on many factors –

1. **Size of time quantum (q):**

If q is large, then this algorithm becomes same as FCFS algorithm and thus performance degrades. **If q is small,** then the number of context switches increases and q almost equals the time taken to switch the CPU from one process to another. This wastes 50% of overall time. So, q should not be very large and should not be very small also so as to achieve good system performance.

2. **Number of context switches:**

The number of context switches should be too many as it slows down the overall execution of all the processes. **Please remember that the time quantum (q) should be large with respect to the context switch time.** This is done to ensure that a process keeps CPU for a considerable amount of time as compared to the time spent in context switching.

Advantages:

- Every process gets an equal share of the CPU.
- RR is cyclic in nature, so there is no starvation.

Disadvantages:

- Setting the quantum too short increases the overhead and lowers the CPU efficiency but setting it too long may cause a poor response to short processes.
- The average waiting time under the RR policy is often long.
- If time quantum is very high, then RR degrades to FCFS.

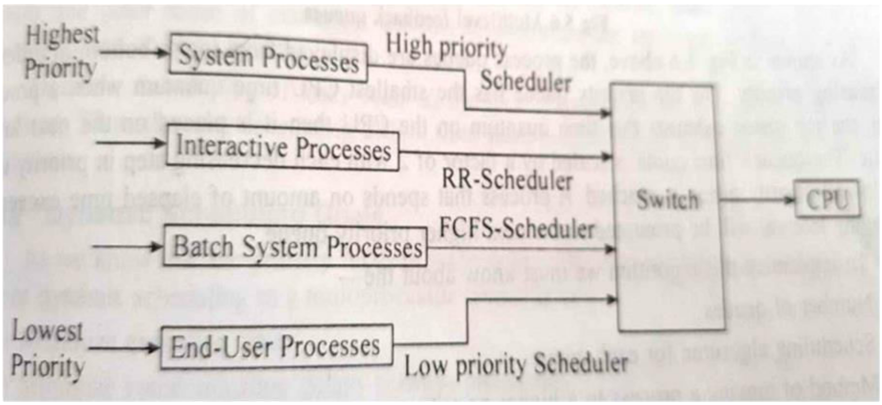
Priority Scheduling (Preemptive and Non-Preemptive) Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

A CPU algorithm that schedules processes based on priority. It used in Operating systems for performing batch processes. If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis. In priority scheduling, a number is assigned to each process that indicates its priority level. Lower the number, higher is the priority. In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy, will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

Multilevel Queue Scheduling When process can be easily classified into different groups then Multilevel Queue Scheduling (MLQ) is applied. MLQ scheduling processes are classified into different groups. For eg., interactive processes (foreground) and batch processes (background) are two types of processes because of their different response time requirements, scheduling needs and priorities. A MLQ scheduling algorithm partitions the ready queue into separate queues. Processes are permanently assigned to each queue. This is done based upon the properties such as memory size or process type. Each queue has its own scheduling algorithm. **For Example,** the interactive queue might be used Round Robin Algorithm. For scheduling while batch queue follows FCFS. A multilevel Queue with five queue is listed below according to order of priority.



Multiple Queue Scheduling

Now, there are different ways of managing queues.

Possibility I: We can assign a time slice to each queue which it can schedule among different processes in its queue. Interactive (or foreground) processes can be assigned 80% of CPU time whereas batch (or background) processes are given 20% of the CPU time.

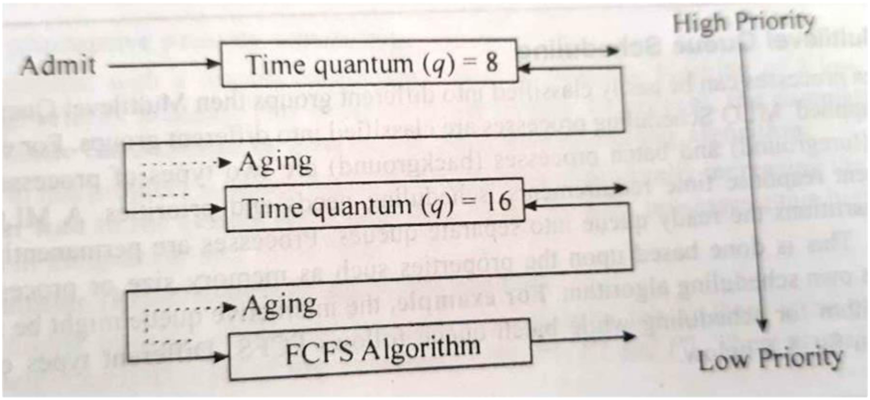
Possibility II: We can execute the high priority queue first. It means that no process in the batch queue could run unless the queue for system process & interactive process were all empty.

Please note that if an interactive process entered the ready queue while a batch process is running, the batch process would be pre-empted.

Advantages of MLQ :- The processes are permanently assigned to a queue when they enter the system. Since processes do not change their foreground or background nature, so there is **low scheduling overhead.**

Disadvantages of MLQ:- Starvation of processes for CPU occurs here as the processes are never allowed to change their queues (inflexibility) and that if one or the other higher priority queues never become empty.

Multilevel Feedback Queue Scheduling – It is an enhancement of multilevel queue scheduling. **The Principle here is that processes can move between the queue now.** In this scheduling, the ready queue is partitioned into multiple queues of different priorities. The processes are assigned to the queues by the system based on their **CPU burst characteristics.** This means that if a process consumes too much of a CPU time, it is placed into a lower priority queue. So, I/O bound and interactive processes stay in the higher priority queues and CPU-bound processes move to the lower priority queues. However, these processes in lower priority queues should be promoted to the next higher priority queues. However, these processes in lower priority queues should be promoted to the next higher priority queue after a suitable time interval. **This technique of moving lower priority processes to the next higher priority queue is known as aging.**



Multi-level Feedback Queues

As shown in the figure above, the process queues are displayed from top to bottom in order to decreasing priority. The top priority queue has the smallest CPU time quantum when a process from the top queue exhausts this time quantum on the CPU then it is placed on the next lower queue.

The queue's time quota is scaled by a factor of 2 with each decreasing step in priority until the lowest priority queue is reached. A process that spends on amount of elapsed time exceeding the aging interval will be promoted to the next higher priority queue.

To implement this algorithm, we must know about the

a) Number of queues
b) Scheduling algorithm for each queue
c) Method of moving a process to a higher priority queue
d) Method of downgrading a process to a lower-priority queue
e) Method of assigning a process to a queue initially

Advantages of MFQs

1. It is a flexible scheduling algorithm as it allows the processes to move between the queues. So, I/O bound processes, do not have to wait long now.
2. Aging can be done. This form of aging prevents starvation.
3. It is the most general CPU scheduling algorithm and can be configured to match a specific system under design.

Disadvantages of MFQs

1. It is a complex scheduling algorithm.
2. The best scheduler is required.
3. Moving the processes around the queue produces more cpu overheads.

Multiple Processor Scheduling

Multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is complex as compared to single processor scheduling. In the multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

The multiple CPUs in the system are in close communication, which shares a common bus, memory, and other peripheral devices. So we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc.

There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. We can use any processor available to run any process in the queue.

Multiprocessor systems may be ***heterogeneous*** (different kinds of CPUs) or *homogenous* (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.

1. **Symmetric Multiprocessing**: It is used where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.
2. **Asymmetric Multiprocessing**: It is used when all the scheduling decisions and I/O processing are handled by a single processor called the Master Server. The other processors execute only the user code. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing.

We have already studied the CPU scheduling algorithms for single CPU systems. What if there are multiple (many) CPUs? Herein, the scheduling problem is more complex, if we consider **Homogeneous systems** i.e.., systems with identical processors then there are certain issues concerning multiprocessor scheduling. Some of issues are given below –

1. With identical processor, **load-sharing** can occur.
2. With a separate queue for each processor, there could be situation where one CPU could be idle with an empty queue while other processor is very busy.

To avoid this situation, we could use a **common ready queue.** All processes enter one queue and are scheduled onto any available processor. In such a scheme, one of the two scheduling approaches may be used.

Case 1: Each processor is **Self-scheduling.** Each processor executes after examining the ready queue and selects a process to be executed. **Please note that we have to take care so that no two processors should select the same process.**

Case 2: One processor is appointed as the scheduler for the other processor, thus creating a master slave structure.

LOTTERY SCHEDULING

Lottery Scheduling is type of process scheduling, somewhat different from other Scheduling. Processes are scheduled in a random manner. Lottery scheduling can be preemptive or non-preemptive. It also solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

In this scheduling every process has some tickets and scheduler picks a random ticket and process having that ticket is the winner and it is executed for a time slice and then another ticket is picked by the scheduler. These tickets represent the share of processes. A process having a higher number of tickets give it more chance to get chosen for execution.

Underlying lottery scheduling is one very basic concept: tickets, which are used to represent the share of a resource that a process (or user or whatever) should receive. The percent of tickets that a process has represents its share of the system resource in question.

Let’s look at an example. Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is for A to receive 75% of the CPU and B the remaining 25%. Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every so often (say, every time slice). Holding a lottery is straightforward: the scheduler must know how many total tickets there are (in our example, there are 100). The scheduler then picks a winning ticket, which is a number from 0 to 991. Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.

Here is an example output of a lottery scheduler's winning tickets:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 12
```

Here is the resulting schedule:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
   B        B                    B     B
```

As you can see from the example, the use of randomness in lottery scheduling leads to a probabilistic correctness in meeting the desired pro portion, but no guarantee. In our example above, B only gets to run 4 out of 20 time slices (20%), instead of the desired 25% allocation. However, the longer these two jobs compete, the more likely they are to achieve the desired percentages.

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
//         get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Figure 9.1: **Lottery Scheduling Decision Code**

Example – If we have two processes A and B having 60 and 40 tickets respectively out of total 100 tickets. CPU share of A is 60% and that of B is 40%.These shares are calculated probabilistically and not deterministically.

Explanation –

1. We have two processes A and B. A has 60 tickets (ticket number 1 to 60) and B have 40 tickets (ticket no. 61 to 100).
2. Scheduler picks a random number from 1 to 100. If the picked no. is from 1 to 60 then A is executed otherwise B is executed.

An example of 10 tickets picked by Scheduler may look like this –

```
Ticket number -  73 82 23 45 32 87 49 39 12 09.
Resulting Schedule -  B  B  A  A  A  B  A  A  A  A.
```

A is executed 7 times and B is executed 3 times. As you can see that A takes 70% of CPU and B takes 30% which is not the same as what we need as we need A to have 60% of CPU and B should have 40% of CPU. This happens because shares are calculated probabilistically but in a long run (i.e when no. of tickets picked is more than 100 or 1000) we can achieve a share percentage of approx. 60 and 40 for A and B respectively.

Ways to manipulate Tickets- Ticket Currency –

Scheduler give a certain number of tickets to different users in a currency and users can give it to there processes in a different currency. E.g. Two users A and B are given 100 and 200 tickets respectively. User A is running two process and give 50 tickets to each in A's own currency. B is running 1 process and gives it all 200 tickets in B's currency. Now at the time of scheduling tickets of each process are converted into global currency i.e A's process will have 50 tickets each and B's process will have 200 tickets and scheduling is done on this basis.

Transfer Tickets – A process can pass its tickets to another process.

Ticket inflation – With this technique a process can temporarily raise or lower the number of tickets its own.

CHAPTER -5 MEMORY VIRTUALIZATION

Basics of Memory Management

- **What does main memory (RAM) contain**? Immediately after booting up, it contains the memory image of the kernel executable, typically in “low memory” or physical memory addresses starting from byte 0. Over time, the kernel allocates the rest of the memory to various running processes. For any process to execute on the CPU, the corresponding instructions and data must be available in main memory. The execution of a process on the CPU generates several requests for memory reads and writes. The memory subsystem of an OS must efficiently handle these requests. Note that contents of main memory may also be cached in the various levels of caches between the CPU and main memory, but the OS largely concerns itself with CPU caches.
- The OS tries to ensure that running processes reside in memory as far as possible. When the system is running low on memory, however, running processes may be moved to a region identified as swap space on a secondary storage device. Swapping processes out leads to a performance penalty and is avoided as far as possible.
- The memory addresses generated by the CPU when accessing the code and data of a running process are called logical addresses. The logical/virtual address space of a process ranges from 0 to a maximum value that depends on the CPU architecture (4GB for 32-bit addresses). Code and data in the program executable are assigned logical addresses in this space by the compiler. On the other hand, the addresses used by the actual memory hardware to locate information are called physical addresses. The physical address space of a system spans from 0 to a maximum value (determined by how much RAM the

machine has). The OS plays a big role in mapping logical addresses of a process to physical addresses.

- When a new process is created, there are two ways of allocating memory to it: contiguous allocation or non-contiguous allocation. With contiguous allocation, the kernel tries to find a contiguous portion of physical memory to accommodate the new process.
- The kernel may use a best-fit or worst-fit or first- fit heuristic to identify a portion of unused memory. The problem with this type of allocation is (external) fragmentation of memory: sometimes, there is enough free memory in the system, but it is fragmented over several locations, so that a contiguous block cannot be found to satisfy a request. To avoid external fragmentation with contiguous allocation, the OS must periodically relocate running processes to other regions of memory.
- With contiguous allocation, the mapping between logical and physical addresses is straightforward. The OS maintains the starting physical address (or base) and the size of the memory image (or limit) for every process it places in memory. This base address value is added to the logical addresses to translate it to a physical address. While contiguous allocation has the benefit of simplicity, it is rarely used in modern operating systems due to the issues around memory fragmentation.
- The most common way of allocating memory to processes in modern operating systems is a type of non-contiguous allocation called paging. The logical address space of every process is divided into fixed-size (e.g., 4KB) chunks called pages. The physical memory is divided into fixed size chunks called frames, which are typically the same size as pages. Every pro cess is allocated some free physical memory frames, and its logical pages are mapped to these physical frames. A page table of a process maintains this mapping from logical page numbers to physical frame numbers.

- The operating system maintains a separate page table for every process it creates. With paging, the issue of external fragmentation is eliminated, because any free frame can be allocated to any process. However, a smaller problem of internal fragmentation (the last page being partially filled) arises.
- Paging adds a level of indirection between logical and physical addressing, and this provides several benefits. There is now a clean separation between logical addresses and physical addresses: the compiler can assign addresses to code and data without worrying about where the program would reside in memory. Paging also makes sharing of memory between processes easier: two logical pages can point to the same physical frame in the page tables. However, paging adds the additional overhead of address translation.
- How are virtual addresses mapped to physical addresses with paging? The virtual address is first split into a page number and an offset within the page. The page number is mapped to the physical frame number by looking up the page table of the process. The physical address is then obtained from the physical frame number and the offset within the frame. Who does this translation from CPU-generated logical addresses to physical addresses? The OS takes the responsibility of constructing and maintaining page tables during the lifecycle of a process; the PCB contains a pointer to the page table.
- The OS also maintains a pointer to the page table of the current process in a special CPU register (CR3 in x86) and updates this pointer during context switches. A specialized piece of hardware called the memory management unit / MMU then uses the page table to translate logical addresses requested by the CPU to physical addresses using the logic described above.
- Segmentation is another way of allocating physical memory to a process. With segmentation, the process memory image is divided into segments corresponding to code, data, stack, and so

on. Each segment is then given a contiguous chunk of physical memory. A segment table stores the mapping between the segments of a process and the base/limit addresses of that segment. Most modern operating systems, however, use paging, or a combination of segmentation and paging. Unix-like operating systems make minimal use of segmentation.

- Segmentation can be used to create a hierarchical address space, where the segment number forms the most significant bits of the logical address. However, Unix-like operating systems mostly use a flat address model on modern CPU architectures in user mode.
- Most C compilers for Linux on x86 today, for example, generate logical addresses from 0 to 4GB, and all segment registers are set to zero. The values of segment registers only change when moving from user mode to kernel mode and vice versa.
- At boot time, the first pieces of code that executes must work only with physical addresses. Once the boot loader executes code to construct page tables and turn the MMU on, the kernel code will start working with virtual addresses. Therefore, the booting process must deal with some complexities arising from this transition.
- On all memory accesses, the memory hardware checks that the memory access is indeed allowed and raises a trap if it detects an illegal access (e.g., user mode process accessing kernel memory or the memory of another process). With contiguous memory allocation, it is easy to check if the memory address generated by the CPU indeed belongs to the process or not: before accessing the memory, one can check if the requested address lies in the range [base, base+limit). With paging, every page page has a set of bits indicating permissions.
- During address translation, the hardware checks that the requested logical address has a corresponding page table entry

with sufficient permissions, and raises a trap if an illegal access is detected.

- With a separation of virtual address space and physical address space in modern operating systems, each process can have a large virtual address space. In fact, the combined virtual address space of all processes can be much larger than the physical memory available on the machine, and logical pages can be mapped to physical frames only on a need basis. This concept is called demand paging and is quite common in modern operating systems. With demand paging, the memory allocated to a process is also called virtual memory, because not all of it corresponds to physical memory in hardware.

Address Space of a Process

- The virtual address space of a process has two main parts: the user part containing the code/data of the process itself, and the kernel code/data. For example, on a 32-bit x86 system, addresses 0-3GB of the virtual address space of a process could contain user data, and addresses 3-4GB could point to the kernel. The page table of every process contains mappings for the user pages and the kernel pages. The kernel page table entries are common for all processes (as there is only one physical copy of the kernel in memory), while the user page table entries are obviously different.
- Note that every physical memory address that is in use will be mapped into the virtual address space of at least one process. That is, the physical memory address will correspond to a virtual address in the page table of some process. Physical memory that is not mapped into the address space of any process is by definition not accessible, since (almost) all accesses to memory go via the MMU. Some physical memory can be mapped multiple times, e.g., kernel code and data is mapped into the virtual address space of every process.
- Why is the kernel mapped into the address space of every process? Having the kernel in every address space makes it easy to execute

kernel code while in kernel mode: one does not have to switch page tables or anything, and executing kernel code is as simple as jumping to a memory location in the kernel part of the address space. Page tables for kernel pages have a special protection bit set, and the CPU must be in kernel mode to access these pages, to protect against rogue processes.

- The user part of the address space contains the executable code of the process and statically allocated data. It also contains a heap for dynamic memory allocation, and a stack, with the heap and stack growing in opposite directions towards each other. Dynamically linked libraries, memory-mapped files, and other such things also form a part of the virtual address space. By assigning a part of the virtual address space to memory mapped files, the data in these files can be accessed just like any other variable in main memory, and not via disk reads and writes. The virtual address space in Linux is divided into memory areas or maps for each of the entities mentioned above.
- The kernel part of the address space contains the kernel code and data. For example, it has various kernel data structures like the list of processes, free pages to allocate to new processes, and so on. The virtual addresses assigned to kernel code and data are the same across all processes.
- One important concept to understand here is that most physical memory will be mapped (at least) twice, once to the kernel part of the address space of processes, and once to the user part of some process. To see why, note that the kernel maintains a list of free frames/pages, which are subsequently allocated to store user process images. Suppose a free frame of size N bytes is assigned a virtual address, say V, by the kernel. Suppose the kernel maintains a 4-byte pointer to this free page, whose value is simply the starting virtual address V of the free page. Even though the kernel only needs this pointer variable to track the page, note that it cannot assign the virtual addresses [V , V + N) to any other variable, because these

addresses refer to the memory in that page, and will be used by the kernel to read/write data into that free page. That is, a free page blocks out a page-sized chunk of the kernel address space. Now, when this page is allocated to a new process, the process will assign a different virtual address range to it (say, [U, U + N)), from the user part of its virtual address space, which will be used by the process to read/write data in user mode. So the same physical frame will also have blocked out another chunk of virtual addresses in the process, this time from the user part. That is, the same physical memory is mapped twice, once into the kernel part of the address space (so that the kernel can refer to it), and once into the user part of the address space of a process (so that the process can refer to it in user mode).

- Is this double consumption of virtual addresses a problem? In architectures where virtual address spaces are much larger than the physical memory, this duplication is not a problem, and it is alright to have one byte of physical memory block out two or more bytes of virtual address space. However, in systems with smaller virtual address spaces (due to smaller number of bits available to store memory addresses in registers), one of the following will happen: either the entire physical memory will not be used (as in the case of xv6), or more commonly, some part of user memory will not be mapped in the kernel address space all the time (as in the case of Linux). That is, once the kernel allocates a free page to a process, it will remove its page table mappings that point to that physical memory and use those freed up virtual addresses to point to something else. Subsequently, this physical memory will only be accessible from the user mode of a process, because only the user virtual addresses point to it in the page table. Such memory is called “high memory” in Linux, and high memory is mapped into the kernel address space (i.e., virtual addresses are allocated from the kernel portion of the virtual memory) only on a need basis.

Memory API Types of Memory

In running a C program, there are two types of memory that are allocated. The first is called **stack memory**, and allocations and deallocations of it are managed implicitly by the compiler for you, the programmer; for this reason, it is sometimes called **automatic memory**. Declaring memory on the stack in C is easy. For example, let's say you need some space in a function func() for an integer, called x. To declare such a piece of memory, you just do something like this:

```
void func()
{
int x; // declares an integer on the stack
...
}
```

The compiler does the rest, making sure to make space on the stack when you call into func(). When you return from the function, the compiler deallocates the memory for you; thus, if you want some information to live beyond the call invocation, you had better not leave that information on the stack.

It is this need for long-lived memory that gets us to the second type of memory, called heap memory,

where all allocations and deallocations are explicitly handled by you, the programmer. A heavy responsibility, no doubt! And certainly, the cause of many bugs. But if you are careful and pay attention, you will use such interfaces correctly and without too much trouble. Here is an example of how one might allocate an integer on the heap:

```
void func()
{
int *x = (int *) malloc(sizeof(int));
...
}
```

A couple of notes about this small code snippet. First, you might notice that both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees your declaration of said pointer (int *x); subsequently, when the program calls malloc(), it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or NULL on failure), which is then stored on the stack for use by the program. Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems.

The malloc() call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL2 . The manual page shows what you need to do to use malloc; type man malloc at the command line and you will see:

```
#include< stdlib.h >
...
void *malloc(size_t size);
```

From this information, you can see that all you need to do is include the header file stdlib.h to use malloc. In fact, you don't really need to even do this, as the C library, which all C programs link with by default, has the code for malloc() inside of it; adding the header just lets the compiler check whether you are calling malloc() correctly (e.g., passing the right number of arguments to it, of the right type). The single parameter malloc() takes is of type size t which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

```
double *d = (double *) malloc(sizeof(double));
```

Wow, that's lot of double-ing! This invocation of malloc() uses the sizeof() operator to request the right amount of space; in C, this is generally thought of as a compile-time operator, meaning that the actual size is known at compile time and thus a number (in this case, 8, for a double) is substituted as the argument to malloc(). For this reason, sizeof() is correctly thought of as an operator and not a function call (a function call would take place at run time). You can also pass in the name of a variable (and not just a type) to sizeof(), but in some cases you may not get the desired results, so be careful. For example, let's look at the following code snippet:

int *x = malloc(10 * sizeof(int)); printf("%d\n", sizeof(x));

In the first line, we've declared space for an array of 10 integers, which is fine and dandy. However, when we use sizeof() in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines). The reason is that in this case, sizeof() thinks we are simply asking how big a pointer to an integer is, not how much memory we have dynamically allocated. However, sometimes sizeof() does work as you might expect: int x[10]; printf("%d\n", sizeof(x)); In this case, there is enough static information for the compiler to know that 40 bytes have been allocated. Another place to be careful is with strings. When declaring space for a string, use the following idiom: malloc(strlen(s) + 1), which gets the length of the string using the function strlen(), and adds 1 to it in order to make room for the end-of-string character. Using sizeof() may lead to trouble here.

You might also notice that malloc() returns a pointer to type void. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a cast; in our example above, the programmer casts the return type of malloc() to a pointer to a double. Casting doesn't really accomplish anything, other than tell the compiler and other programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of malloc(), the programmer is just giving some reassurance; the cast is not needed for the correctness.

The free() Call As it turns out, allocating memory is the easy part of the equation; knowing when, how, and even if to free memory is the hard part. To free heap memory that is no longer in use, programmers simply call

free(): int *x = malloc(10 * sizeof(int)); ... free(x);

The routine takes one argument, a pointer returned by malloc(). Thus, you might notice, the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself.

Other Calls:

There are a few other calls that the memory-allocation library supports. For example, calloc() allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself (see the paragraph on "uninitialized reads" above). The routine realloc() can also be useful, when you've allocated space for something (say, an array), and then need to add something to it: realloc() makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region.

Common Errors

There are a number of common errors that arise in the use of malloc() and free().

1. Forgetting To Allocate Memory
2. Not Allocating Enough Memory
3. Forgetting to Initialize Allocated Memory
4. Forgetting To Free Memory
5. Freeing Memory Before You Are Done With It
6. Freeing Memory Repeatedly
7. Calling free() Incorrectly

VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

Following are the situations, when entire program is not required to load fully.

1. User written error handling routines are used only when an error occurs in the data or computation.
2. Certain options and features of a program may be used rarely.
3. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

The ability to execute a program that is only partially in memory would counter many benefits.

1. Less number of I/O would be needed to load or swap each user program into memory.
2. A program would no longer be constrained by the amount of physical memory that is available.
3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

CHAPTER-6: INTRODUCTION TO PAGING

Paging

- Paging is a memory management technique that allows non-contiguous memory allocations to processes and avoids the problem of external fragmentation. Most modern operating systems use paging. With paging, the physical memory is divided into frames, and the virtual address space of a process is divided into logical pages. Memory is allocated at the granularity of pages. When a process requires a new page, the OS finds a free frame to map this page into and inserts a mapping between the page number and frame number in the page table of the process.
- Suppose the size of the logical address space is 2 m bytes, and the size of the physical memory is 2 n bytes. That is, logical addresses are m bits long, and physical memory addresses are n bits long. Suppose the size of each logical page and physical frame is 2 k bytes. Then, out of the m bits in the virtual address, the most significant m − k bits indicate the page number, and the least significant k bits indicate the offset within the page. A page table of a process contains 2 m−k page table entries (PTEs), one for each logical page of the process. Each PTE stores, among other things, the physical frame number which this page corresponds to. Now, since there are a total of 2 n−k physical frames, it would take n − k bits to identify a physical frame. Thus, each of the PTEs must be at least n−k bits in size, putting the total size of the page table of a process at at least 2 m−k × (n − k) bits. Note that PTEs contain a bit more information than just the physical frame number, e.g., permissions to access the page, so PTEs are slightly longer than n − k bits.
- For example, consider a system with 32-bit virtual addresses (i.e., the virtual address space is of size 2 32 bytes = 4GB). Suppose the size of the physical memory is 64 GB, and pages/frames are of size

4KB each. The total number of pages, and hence the number of PTEs in a page table, is 2 20 for each process. The number of physical frames in the system is 2 24. Therefore, each PTE must be at least 24 bits, or 3 bytes long. Assume that a PTE is 4 bytes long. Therefore, the size of the page table of every process is 2 20 $*$ 4 bytes = 4MB.

- To translate a virtual address to a physical address, the MMU uses the most significant p = m – k bits of the virtual address to offset into the page table and replace the most significant p bits with the f = n − k bits of the physical frame number. Assuming pages and frames are the same size, the lower k bits that indicate the offset in the logical page will still map to the offset in the physical frame.
- Note that a process need not always use up its entire virtual address space, and some logical pages can remain unassigned. In such cases, a bit in the PTE can indicate whether the page is valid or not, or the size of the page table can itself be trimmed to hold only valid pages. Accesses to unmapped/invalid logical addresses will cause the MMU to raise a trap to the OS.
- The OS can sometimes leave certain addresses unmapped on purpose, e.g., at the end of the user stack, to detect processes writing beyond page boundaries. The valid bit in the PTE, and the read/write permissions bits are powerful tools for the OS to enforce memory access control at the granularity of a page.
- A subtle point to note is that when a process is executing on the CPU, memory access requests from the CPU directly go to the memory via the MMU, and the OS is not involved in inspecting 6 or intercepting every request. If the OS wishes to be informed of access to any memory page, the only way to enforce it would be to set appropriate bits in the PTEs which will cause a memory request to trap and reach the kernel. For example, consider how an OS would implement copy on- write during fork. If the OS has only one memory copy for the child and parent processes, how does it know when one of them modifies this common image, so that it can make

a copy? The OS can set all PTEs in the common memory image to read only, so that when either the parent or the child tries to write to the memory, a trap is raised by the MMU. This trap halts the execution of the process, moves it to kernel mode, and transfers control to the kernel. The kernel can then make a copy the memory image, update the page table entries to point to the new image, remove the read only constraint, and restart the halted process.

- How are pages sized? In general, smaller page sizes lead to lesser internal fragmentation (i.e., space being unused in the last page of the process). However, smaller pages also imply greater overhead in storing page tables. Typical pages sizes in modern operating systems are 4-8KB, though systems also support large pages of a few MB in size. Most modern operating systems support multiple page sizes based on the application requirements.
- With smaller virtual address spaces, the page table mappings could fit into a small number of CPU registers. But current page tables cannot be accommodated in registers. Therefore, page tables are stored in memory and only a pointer to the starting (physical) address of a page table is stored in CPU registers (and changed at the time of a context switch). Thus, accessing a byte from memory first requires mapping that byte's virtual address to a physical address, requiring one or more accesses of the page table in memory, before the actual memory access can even begin. This delay of multiple memory accesses imposes an unreasonable overhead. Therefore, most modern systems have a piece of hardware called a **translation look-aside buffer (TLB).**

TLB Cache

A TLB is a small, fast cache that stores the most recently used mappings from virtual to physical addresses. TLB is an associative memory that maps keys (virtual addresses) to values (physical addresses). Most architectures implement the TLB in hardware, and the TLB is transparent to the OS.

That is, the OS only manages the page tables and does not control the TLB, and caching in the TLB is done automatically by the hardware. However, some architectures do provide a software TLB with more complicated features.

- If a virtual address is found in the TLB (a TLB hit), a memory access can be directly made. On the other hand, if a TLB miss occurs, then one or more memory accesses must first be made to the page tables before the actual memory access can be made. A healthy TLB hit ratio is thus essential for good performance. Further, the TLB reach (or the amount of address space it covers) should also be large.
- Let Tc denote the time to check if an address is mapped in the TLB, and let Tm denote the time to access main memory. Further, suppose a TLB miss requires n additional memory accesses to page tables before the actual memory access. Then, the expected time for a memory access in a system with a TLB hit ratio of f is given by $f \times (Tc + Tm) + (1 - f) \times (Tc + (n + 1)Tm)$.
- LRU (least recently used) policy is often used to evict entries from the TLB when it is running out of space. However, some entries (e.g., those corresponding to kernel code) are permanently stored in the TLB. Further, most TLB entries become invalidated after a context switch, because the mappings from virtual to physical addresses are different for different processes. To avoid flushing all TLB entries on a context switch, some TLB designs store a tag of the process ID in the TLB entry, and access only those TLB entries whose tag matches that of the current running process.

Page Table Design

- How are page tables stored in memory? Since the CPU uses just one register to store the starting address of the page table, one way is to store the page table contiguously beginning from the start address. However, it is hard to store page tables as large as 4MB contiguously in memory. Therefore, page tables are themselves split up and stored in pages. Suppose a PTE is of size 2 e bytes and pages are of size 2 k bytes. Then each page can hold 2 k−e PTEs. Now, the 2 m−k page table entries in the inner page table will be spread out over 2m−k 2 k−e = 2m+e−2k pages. And an outer page table stores the frame numbers of the pages of the inner page tables. That is, given the virtual address, the least significant k bits provide an offset in a page, the middle k−e bits will be used to offset into the inner page table to obtain the frame number, and the most significant m + e − 2k bits will be used to index into the outer page table to find the location (frame number) of the inner page table itself. For example, for 32-bit virtual addresses, 4KB pages, and 4-byte page table entries, the most significant 10 bits are used to index into the outer page table, and next 10 bits are used to index into the inner page table, to find the frame number. For this example, the outer page table fits snugly in a page. However, if the outer page table size gets larger, it further may need to be split, and several levels of hierarchy may be required to store the page table. This concept is called hierarchical paging, and quickly gets inefficient for large virtual address spaces (e.g., 64 bit architectures).

- An alternative design for large address spaces is a hashed page table. In this design, the virtual address (key) is hashed to locate its corresponding value entry. For multiple virtual addresses that hash to the same location, a linked list or some such mechanism is used to store all the mappings.
- Another alternative design for page tables in systems with large virtual address spaces is to have a table with entries for every physical frame, and not for every logical page. This is called an **inverted page table**. Every entry of the inverted page table maps a physical frame number to the process identifier, and the logical page number in the address space of that process. To translate a virtual address to a physical address, one would have to search through all the entries in the inverted page table until a matching logical page number is found. So, while inverted page tables save on size, lookup

times are longer. Also, inverted page tables make it hard to share pages between processes.

- Some operating systems use a combination of segmentation and paging. The logical address space is divided into a small number of segments, and the logical address is the combination of the segment number and an offset within that segment. The CPU can only specify the offset part of the logical address, and based on which segment of the memory image it is executing in, the complete logical address can be generated. This logical address is then mapped to the physical address via paging. For example, Linux uses segmentation with paging on architectures that have support for segmentation (e.g., Intel x86). Linux uses a small number of segments: one each of user code, user data, kernel code, kernel data, stack segment, and so on. A segment table stores the starting (logical) addresses of each of these segments in the virtual address space of a process, and implements protection in terms of who can access that segment. For example, access to the kernel segments is allowed only when the CPU is in kernel mode.

Demand Paging

When are physical frames allocated to logical pages? In a simplistic case, all logical pages can be assigned physical frames right at the beginning when a process is created and brought into memory. However, this is wasteful for a number of reasons. For example, a process may access only parts of its memory during a run. Therefore, modern operating systems use a concept called demand paging. With demand paging, a logical page is assigned a physical frame only when the data in the page needs to be accessed. At other times, the data in the page will reside on secondary storage like a hard disk. A bit in the page table indicates whether the page is in memory or not. With demand paging, a process is allocated only as much physical memory as it needs, thereby allowing us to accommodate many more processes in memory.

Suppose the CPU tries to access a certain logical address and the page table shows that the page containing that address has not been mapped into memory. Then a page fault is said to occur. A page fault traps the OS. The process moves into kernel mode. The OS then handles the page fault as follows: a free physical frame is identified, the required page is requested to be read from the disk, and the process that generated the page fault is context switched out.

The CPU then starts running another process. When the data for the page has been read from disk into the free frame, an interrupt occurs, and is handled by whichever process currently has the CPU in its kernel mode. The interrupt service routine in the OS updates the page table of the process that saw the page fault, and marks it as ready to run again. When the CPU scheduler runs the process again, the process starts running at the instruction that generated the page fault, and continues execution normally.

Note that an instruction can be interrupted midway due to a page fault, and will be restarted from the beginning after the page fault has been serviced. Care should be taken in saving the context of a process to ensure that any temporary state that needs to be preserved is indeed preserved as part of the saved context. The instruction set of the underlying architecture must be carefully examined for such issues when adding paging to the OS.

Let Tm be the time to access main memory and Tf be the time to service a page fault (which is typically large, given that a disk access is required). If p is the page fault rate, then the effective memory access time is $(1 - p) \times Tm + p \times Tf$. Systems with high page fault rate see a high memory access time, and hence worse performance.

Note that most operating systems do not allow the kernel code or data to be paged out, for reasons of performance and convenience. For example, the operating system could never recover from a page fault, if the code to handle the page fault itself is paged out, and causes a page fault when accessed.

Some operating systems implement pre-paging, where a set of pages that are likely to be accessed in the future are brought into memory before a page fault occurs. The success of pre- paging depends on how well the page access pattern can be predicted, so as to not bring pages that will never be used into memory.

Note that an inverted page table design will be inefficient when demand paging is used, because one also needs to lookup pages that are not in memory, and an inverted page table does not have entries for such pages. Therefore, operating systems that use inverted page tables must also keep a regular page table around to identify and service page faults.

Demand paging allows for an over-commitment of physical memory, where the aggregate virtual addresses of all processes can be (much) larger than the underlying physical address space. Over-commitment is generally acceptable because processes do not tend to use all their memory all the time. In fact, most memory accesses have a locality of reference, where a process runs in one part of its virtual address space for a while (causing memory accesses in a small number of pages), before moving on to a different part of its address space. Therefore, over-commitment, when done carefully, can lead to improved performance and an ability to support a large number of processes. Operating systems that over-commit memory to processes are said to provide virtual (not physical) memory to processes. Note that virtual addressing does not necessarily imply virtual memory, though most modern operating systems implement both these ideas.

To implement virtual memory, an OS requires two mechanisms. It needs a frame allocation algorithm to decide how many physical frames to allocate to each process. Note that every process needs a certain minimum number of frames to function, and an OS must strive to provide this minimum number of frames. Further, when all physical frames have been allocated, and a process needs a new frame for a page, the OS must take a frame away from one logical page (that is hopefully not being actively used) and use it to satisfy the new request. A page replacement algorithm decides which logical page must be replaced in such cases

A process memory image is initially read from the file system. As the process runs, some pages may be modified and may diverge from the content in the file systems. Such pages that are not backed by a copy in the file system are called dirty pages. When a dirty page is replaced, its contents must be flushed to disk (either to a file or to swap space). Therefore, servicing a page fault may incur two disk operations, one to swap out an unused page to free up a physical frame, and another to read in the content of the new page. The page table maintains a dirty bit for every page, along with a valid bit.

Demand paging (as opposed to anticipatory paging) is a method of virtual memory management. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (i.e., if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages are located in physical memory. This is an example of a lazy loading technique.

Consider a process executing on an operating system that uses demand paging. The average time for a memory access in the system is M units if the corresponding memory page is available in memory, and D units if the memory access causes a page fault. It has been experimental measured that the average time taken for a memory access in the process is X units. Give an expression for the page fault rate experienced by the process?

Given, average time for a memory access = M units if page hits, and average time for a memory access = D units if page fault occurred.
And total/experimental average time taken for a memory access = X units. Let page fault rate is p. Therefore,
Average memory access time = (1 – page fault rate) * memory access time when no page fault
+ Page fault rate * Memory access time when page fault

$\rightarrow X = (1 - p)*M + p*D = M - M*p + p*D$
$\rightarrow\ = M + p(D - M)$

PAGE REPLACEMENT ALGORITHMS

1. **FIFO (First in First Out)** is the simplest page replacement algorithm. With FIFO, logical pages assigned to processes are placed in a FIFO queue. New pages are added at the tail. When a new page must be mapped, the page at the head of the queue is evicted. This way, the page brought into the memory earliest is swapped out. However, this oldest page maybe a piece of code that is heavily used, and may cause a page fault very soon, so FIFO does not always make good choices, and may have a higher than optimal page fault rate. The FIFO policy also suffers from Belady's anomaly, i.e., the page fault rate may not be monotonically decreasing with the total available memory, which would have been the expectation with a sane page replacement algorithm. (Because FIFO doesn't care for popularity of pages, it may so happen that some physical memory sizes lead you to replace a heavily used page, while some don't, resulting in the anomaly.) The FIFO is the simplest page replacement algorithm, the idea behind this is "Replace a page that page is the oldest page of all the pages of the main memory" or "Replace the page that has been in memory longest".

2. What is the optimal page replacement algorithm? One must ideally replace a page that will not be used for the longest time in the future. However, since one cannot look into the future, this optimal algorithm cannot be realized in practice. The optimal page replacement has the lowest page fault rate of all algorithms. The criteria of this algorithm is **"Replace a page that will not be used for the longest period of time".**
3. **The LRU (least recently used)** policy replaces the page that has not be used for the longest time in the past, and is somewhat of an

approximation to the optimal policy. It doesn't suffer from Belady's anomaly (the ordering of least recently used pages won't change based on how much memory you have), and is one of the more popular policies. To implement LRU, one must maintain the time of access of each page, which is an expensive operation if done in software by the kernel. Another way to implement LRU is to store the pages in a stack, move a page to the top of the stack when accessed, and evict from the bottom of the stack. However, this solution also incurs a lot of overhead (changing stack pointers) for every memory access.

LRU is best implemented with some support from the hardware. Most memory hardware can set a reference bit when a page is accessed. The OS clears all reference bits initially, and periodically checks the reference bits. If the bit is set, the OS can know that a page has been accessed since the bit was last cleared, though it wouldn't know when it has been accessed during that period. An LRU-like algorithm can be built using reference bits as follows: the OS samples and clears the reference bit of every page every epoch (say, every few milliseconds), and maintains a history of the reference bit values in the last few epochs in the page table. Using this history of references, the OS can figure out which pages were accessed in the recent epochs and which weren't.

The Criteria of this algorithm is **"Replace a page that has not been used for the longest period of time".** The strategy is "**Page replacement algorithm looking backward in time, rather than forward".**

Finally, several variants of the simple FIFO policy can be created using the hardware support of reference bits. In the second chance algorithm, when the OS needs a free page, it runs through the queue of pages, much like in FIFO. However, if the reference bit is set, the OS skips that page, and clears the bit (for the future). The OS walks through the list of pages, clearing set bits, until a page with no reference bit set is found. The OS keeps this pointer to the list and resumes its search for free pages from the following page for subsequent runs of the algorithm.

In addition to the reference bit, the second chance algorithm can also look at the dirty bit of the page. Pages that have not been recently accessed or written to (both reference and dirty bits are not set) are ideal candidates for replacements, and pages that have both bits set will not be replaced as far as possible.

4. **Least Frequently Used (LFU)** algorithm "**Selects a page for replacement, if the page has not been used often in the past"** or **"Replace page that page has smallest count".** For this algorithm each page maintains a counter, which counter value shows the least count, replace the page. The reason for this selection is that an actively used page should have a large reference count. So don't replace the activity used page.
5. **Most Frequently Used Algorithm (MFU)**, the criteria of this algorithm is replace a page that has the maximum frequency count of all pages. The Implementation of this algorithm is fairly expensive.

Swapping

Swapping is the process of temporarily removing inactive programs from the main memory of a computer system. Swapping is a memory management technique and is used to temporarily remove the inactive programs from the main memory of the computer system. Any process must be in the memory for its execution but can be swapped temporarily out of memory to a backing store and then again brought back into the memory to complete its execution. Swapping is done so that other processes get memory for their execution.

Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

The purpose of the swapping in operating system is to access the data present in the hard disk and bring it to RAM so that the application programs can use it. The thing to remember is that swapping is used only when data is not present in RAM.

Although the process of swapping affects the performance of the system, it helps to run larger and more than one process. This is the reason why swapping is also referred to as memory compaction.

The concept of swapping has divided into two more concepts: Swap-in and Swap-out.

- Swap-out is a method of removing a process from RAM and adding it to the hard disk.
- Swap-in is a method of removing a program from a hard disk and putting it back into the main memory or RAM.

Example: Suppose the user process's size is 2048KB and is a standard hard disk where swapping has a data transfer rate of 1Mbps. Now we will calculate how long it will take to transfer from main memory to secondary memory.

User process size is 2048Kb

Data transfer rate is 1Mbps = 1024 kbps Time = process size / transfer rate

= 2048 / 1024

= 2 seconds

= 2000 milliseconds

Now taking swap-in and swap-out time, the process will take 4000 milliseconds.

Advantages of Swapping

1. It helps the CPU to manage multiple processes within a single main memory.
2. It helps to create and use virtual memory.

Swapping allows the CPU to perform multiple tasks simultaneously. Therefore, processes do not have to wait very long before they are executed.

3. It improves the main memory utilization.

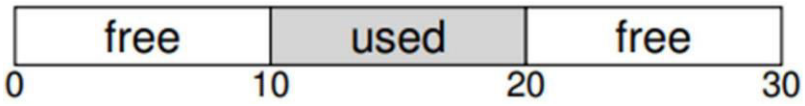
Disadvantages of Swapping

1. If the computer system loses power, the user may lose all information related to the program in case of substantial swapping activity.
2. If the swapping algorithm is not good, the composite method can increase the number of Page Fault and decrease the overall processing performance.

Free Space Management

Managing free space can certainly be easy, as we will see when we discuss the concept of paging. It is easy when the space you are managing is divided into fixed-sized units; in such a case, you just keep a list of these fixed-sized units; when a client requests one of them, return the first entry.

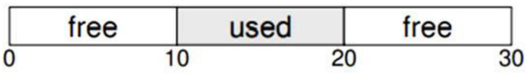
Where free-space management becomes more difficult (and interesting) is when the free space you are managing consists of variable-sized units; this arises in a user-level memory-allocation library (as in **malloc()** and **free()**) and in an OS managing physical memory when using segmentation to implement virtual memory. In either case, the problem that exists is known as external fragmentation: the free space gets chopped into little pieces of different sizes and is thus fragmented; subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.



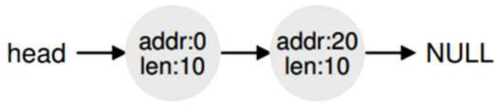
The figure shows an example of this problem. In this case, the total free space available is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free.

Splitting and Coalescing

A free list contains a set of elements that describe the free space still remaining in the heap. Thus, assume the following 30-byte heap:

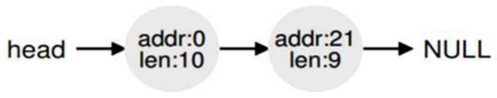


The free list for this heap would have two elements on it. One entry describes the first 10-byte free segment (bytes 0-9), and one entry describes the other free segment (bytes 20-29):



As described above, a request for anything greater than 10 bytes will fail (returning NULL); there just isn't a single contiguous chunk of memory of that size available. A request for exactly that size (10 bytes) could be satisfied easily by either of the free chunks. But what happens if the request is for something *smaller* than 10 bytes?

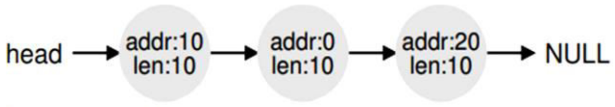
Assume we have a request for just a single byte of memory. In this case, the allocator will perform an action known as **splitting**: it will find a free chunk of memory that can satisfy the request and split it into two. The first chunk it will return to the caller; the second chunk will remain on the list. Thus, in our example above, if a request for 1 byte were made, and the allocator decided to use the second of the two elements on the list to satisfy the request, the call to malloc() would return 20 (the address of the 1-byte allocated region) and the list would end up looking like this:



In the picture, you can see the list basically stays intact; the only change is that the free region now starts at 21 instead of 20, and the length of that free region is now just 9[3]. Thus, the split is commonly used in allocators when requests are smaller than the size of any particular free chunk.

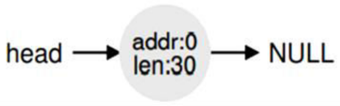
A corollary mechanism found in many allocators is known as **coalescing** of free space. Take our example from above once more (free 10 bytes, used 10 bytes, and another free 10 bytes).

Given this (tiny) heap, what happens when an application calls free(10), thus returning the space in the middle of the heap? If we simply add this free space back into our list without too much thinking, we might end up with a list that looks like this:



Note the problem: while the entire heap is now free, it is seemingly divided into three chunks of 10 bytes each. Thus, if a user requests 20 bytes, a simple list traversal will not find such a free chunk, and return failure.

What allocators do in order to avoid this problem is coalesce free space when a chunk of memory is freed. The idea is simple: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly-freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk. Thus, with coalescing, our final list should look like this:



Indeed, this is what the heap list looked like at first, before any allocations were made. With coalescing, an allocator can better ensure that large free extents are available for the application.

Thrashing in Operating System

In case, if the page fault and swapping happens very frequently at a higher rate, then the operating system has to spend more time swapping these pages. This state in the operating system is termed thrashing. Because of thrashing the CPU utilization is going to be reduced.

Let's understand by an example, if any process does not have the number of frames that it needs to support pages in active use then it will quickly page fault. And at this point, the process must replace some pages. As all the pages of the process are actively in use, it must replace a page that will be needed again right away. Consequently, the process will quickly fault again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity by a process is called thrashing.

During thrashing, the CPU spends less time on some actual productive work spend more time swapping.

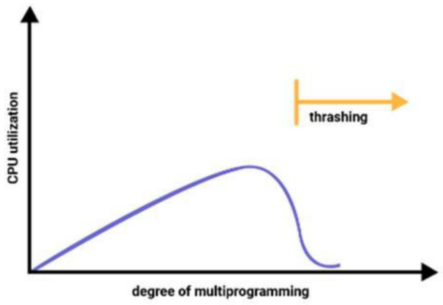


Figure: Thrashing

Causes of Thrashing

Thrashing affects the performance of execution in the Operating system. Also, thrashing results in severe performance problems in the Operating system. When the utilization of CPU is low, then the process scheduling mechanism tries to load many processes into the memory at the same time due to which degree of Multiprogramming can be increased. Now in this situation, there are more processes in the memory as compared to the available number of frames in the memory. Allocation of the limited amount of frames to each process.

Whenever any process with high priority arrives in the memory and if the frame is not freely available at that time then the other process that has occupied the frame is residing in the frame will move to secondary storage and after that this free frame will be allocated to higher priority process. We can also say that as soon as the memory fills up, the process starts spending a lot of time for the required pages to be swapped in. Again the utilization of the CPU becomes low because most of the processes are waiting for pages.

Thus a high degree of multiprogramming and lack of frames are two main causes of thrashing in the Operating system.

Reference:

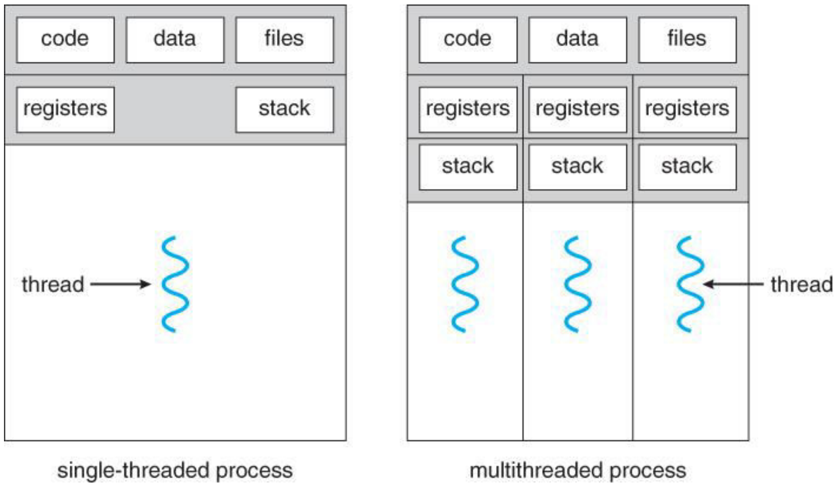
1. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, Arpaci-Dusseau Books, May, (2014).

CHAPTER-7-CONCURRENCY AND THREADS

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g., memory and CPU time, are allocated to it via the operating system.

A thread is a so-called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache.

- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID)
- Traditional (heavy weight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in below figure, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.



Multithreading is a technique that allows for concurrent (simultaneous) execution of two or more parts of a program for maximum utilization of a CPU, as a really basic example, multithreading allows you to write code in one program and listen to music in another. Programs are made up of processes and threads. You can think of it like this:

- A program is an executable file like chrome.exe
- A process is an executing instance of a program. When you double click on the Google Chrome icon on your computer, you start a process which will run the Google Chrome program.
- Thread is the smallest executable unit of a process. A process can have multiple threads with one main thread. In the example, a single thread could be displaying the current tab you're in, and a different thread could be another tab.

Threading is a process where multiple threads run at the same time to increase the efficiency of the processor. Using the Threading API, you can run a main thread and a worker thread simultaneously in your JavaScript application. The Threading API uses kony namespace and the following API elements.

MULTITHREADING MODELS

There are two types of threads to be managed in a modern system: User threads and kernel threads.

User Threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

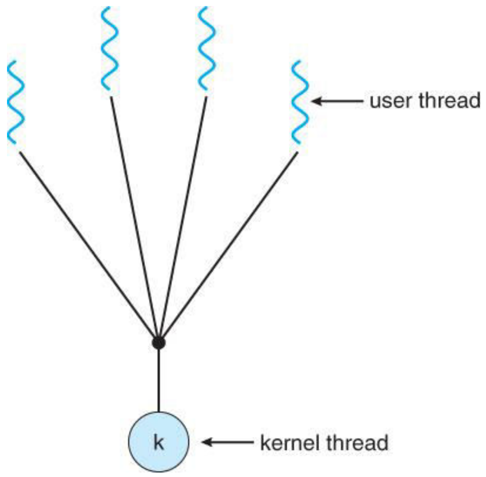
Kernel Threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

A) Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.

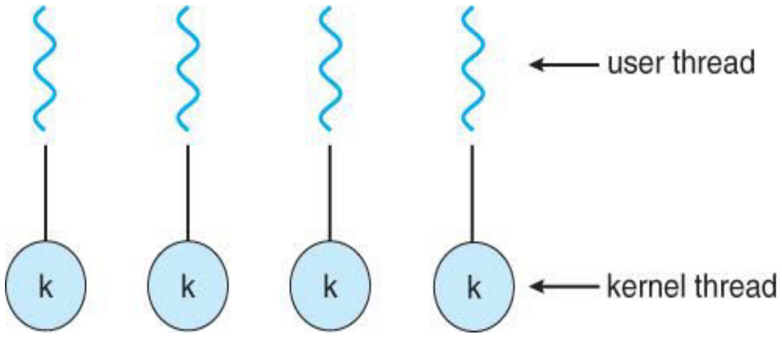
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.



B) One-To-One Model

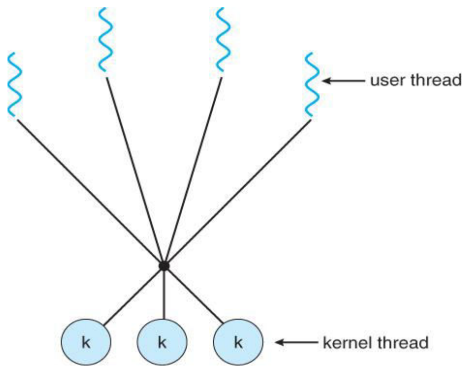
- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.

- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



C)Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors



Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- **There are three main thread libraries in use today:**
    1. **POSIX P threads** - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    2. **Win32 threads** - provided as a kernel-level library on Windows systems.
    3. **Java threads** - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

User Threads

Thread management done by user-level threads library

Three primary thread libraries:

- POSIX Pthreads
- Win32 threads
- Java threads

Kernel Thread

Supported by the Kernel

Examples

- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

Multithreading Models

Many-to-One

Many user-level threads mapped to single kernel thread

Examples:

- Solaris Green Threads
- GNU Portable Threads

Importance of concurrency processing in modern operating systems

Concurrency processing is of paramount importance in modern operating systems due to the following reasons −

Improved performance − With the advent of multi-core processors, modern operating systems can execute multiple threads or processes simultaneously, leading to improved system performance. Concurrency processing enables the operating system to make optimal use of available resources, thereby maximizing system throughput.

Resource utilization − Concurrency processing allows for better utilization of system resources such as CPU, memory, and I/O devices. By executing multiple threads or processes concurrently, the operating system can use idle resources to execute other tasks, leading to better resource utilization.

Enhanced responsiveness − Concurrency processing enables the operating system to handle multiple user requests simultaneously, leading to improved system responsiveness. This is particularly important in applications that require real-time processing, such as online gaming or financial trading applications.

Scalability − Concurrency processing enables the operating system to scale efficiently as the number of users or tasks increases. By executing tasks concurrently, the operating system can handle a larger workload, leading to better system scalability.

Flexibility − Concurrency processing enables the operating system to execute tasks independently, making it easier to manage and maintain the system. This flexibility makes it possible to develop complex applications that require multiple threads or processes without compromising system performance.

Overall, concurrency processing is a critical technology in modern operating systems, enabling them to provide the performance, responsiveness, and scalability required by today's computing environment.

Type of Concurrency Processing In Operating System

There are several types of concurrency processing techniques used in operating systems, including −

Process scheduling − This is the most basic form of concurrency processing, in which the operating system executes multiple processes one after the other, with each process given a time slice to execute before being suspended and replaced by the next process in the queue.

Multi-threading − This involves the use of threads within a process, with each thread executing a different task concurrently. Threads share the same memory space within a process, allowing them to communicate and coordinate with each other easily.

Parallel processing − This involves the use of multiple processors or cores within a system to execute multiple tasks simultaneously. Parallel processing is typically used for computationally intensive tasks, such as scientific simulations or video rendering.

Distributed processing − This involves the use of multiple computers or nodes connected by a network to execute a single task. Distributed processing is typically used for large-scale, data-intensive applications, such as search engines or social networks.

CHAPTER-8: DEADLOCK PREVENTION, DETECTION AND AVOIDANCE

8.1 INTRODUCTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

Deadlock is a critical concept in the field of operating systems, particularly in the context of concurrent and multi-threaded environments. It occurs when two or more processes (or threads) are unable to proceed because each is waiting for the other to release a resource or complete a task, resulting in a standstill where none of the processes can make progress.

To understand deadlock better, let's look at the four necessary conditions for deadlock to occur, often referred to as the "deadlock conditions" or "Coffman conditions":

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode, meaning only one process can use it at a time. When a process holds a resource, no other process can access it until it is released.
2. **Hold and Wait**: A process must be holding at least one resource while waiting to acquire additional resources that are currently held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken away from processes; they can only be released voluntarily by the process holding them.
4. **Circular Wait**: A circular chain of two or more processes exists, where each process is waiting for a resource that is held by another process in the chain.

When all four conditions are met simultaneously, a deadlock situation arises. This can lead to a system becoming unresponsive, and the only way to recover from a deadlock is to abort one or more processes, releasing the resources they were holding. However, deciding which process to abort can be challenging and may lead to data loss or undesirable consequences.

Deadlock avoidance, detection, and recovery are essential techniques used by operating systems to handle or prevent deadlocks from causing system failures. Prevention involves ensuring that at least one of the deadlock conditions is never satisfied. Detection involves periodically examining the state of the system to identify potential deadlocks, while recovery involves taking corrective actions once a deadlock is detected.

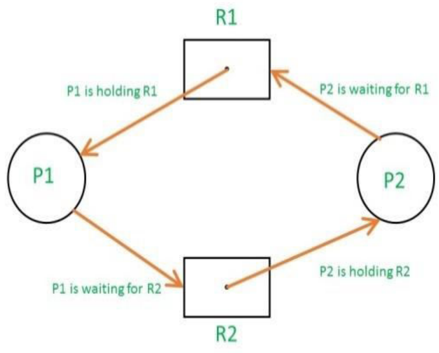
Handling deadlocks effectively is crucial to ensure the stability and reliability of modern operating systems, especially in scenarios where multiple processes are competing for shared resources.

8.2 Resource-Allocation Graph

In operating systems, a resource allocation graph is a graphical representation used to model and analyze the allocation of resources among different processes and the relationships between them. It is a critical tool for understanding resource allocation and detecting potential deadlocks in the system.

A resource allocation graph consists of two types of nodes:

1. Process Nodes: Each process in the system is represented by a process node. It indicates the processes that are currently running or requesting resources.
2. Resource Nodes: Each resource in the system is represented by a resource node. It indicates the instances of resources available in the system.



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

Edges in the graph represent the relationships between processes and resources:

1. **Request Edge:** If a process is requesting a resource, there is a directed edge from the process node to the resource node. This indicates that the process is waiting for that resource to become available.
2. **Assignment Edge:** If a resource is allocated to a process, there is a directed edge from the resource node to the process node. This indicates that the process currently holds that resource.

A resource allocation graph may evolve over time as processes request and release resources. Here are some key points to consider in analyzing the graph:

1. **Deadlock Detection**: A deadlock occurs when there is a cycle in the resource allocation graph. In other words, there is a circular chain of processes waiting for resources that are held by other processes. Detecting such cycles in the graph helps to identify potential deadlocks.
2. **Deadlock Resolution**: Once a deadlock is detected, operating systems can resolve it by employing various deadlock handling strategies, such as resource preemption, process termination, or resource timeout.

3. **Safe State**: If there is no deadlock in the system, it is said to be in a "safe state" where all processes can complete their execution successfully. A state is considered safe if there exists a sequence of process execution that avoids deadlock.
4. **Resource Allocation Policies**: The resource allocation graph can also be used to evaluate the efficiency and effectiveness of resource allocation policies. For example, it can help determine if a resource allocation policy may lead to potential deadlocks under certain conditions.

Overall, the resource allocation graph is a valuable tool for understanding the dynamic behavior of resource allocation and identifying potential issues like deadlocks in operating systems.

8.3 Methods for Handling the Dead Lock

Handling deadlocks in operating systems involves employing various strategies to prevent or resolve deadlock situations. Some common methods for dealing with deadlocks are:

1. **Deadlock Prevention:**
    - **Resource Allocation Order:** One way to prevent deadlocks is by defining a specific order in which resources must be allocated. This can prevent circular wait conditions that lead to deadlocks.
    - **Resource Limitation:** Limiting the maximum number of resources a process can hold at any given time can reduce the likelihood of deadlocks.
2. **Deadlock Avoidance:**
    - **Safe State Check:** The operating system can employ algorithms like the Banker's algorithm to check if a state is safe before allocating resources to a process. If the state is not safe, the resource allocation is deferred until it becomes safe.
    - **Resource Request Check:** Before allocating resources to a process, the operating system can predict if the

allocation will potentially lead to a deadlock. If so, the allocation is postponed or denied.

3. **Deadlock Detection and Recovery:**
    - **Resource Allocation Graph:** As mentioned earlier, the resource allocation graph can be used to detect deadlocks in the system. Once a deadlock is detected, appropriate actions can be taken to resolve it.
    - **Timeout Mechanisms:** A timeout can be set on resource requests. If a process cannot acquire the required resources within a specified time, it releases all its allocated resources and restarts execution from the beginning.
4. **Resource Preemption:**
    - If a process holding certain resources is waiting indefinitely for additional resources held by other processes, the operating system can preempt the resources from the waiting process and allocate them to the waiting process, thereby breaking the potential deadlock.
5. **Process Termination:**
    - In situations where deadlock resolution is difficult or not feasible, the operating system can choose to terminate one or more processes to release the resources they are holding. The terminated processes can be restarted later to continue their tasks.

It's essential to note that each deadlock handling method comes with its advantages and disadvantages. Deadlock prevention and avoidance methods are proactive but may lead to underutilization of resources. On the other hand, deadlock detection and recovery strategies are reactive and incur overhead for deadlock detection routines. The choice of method depends on the system's requirements, resource utilization, and the criticality of the application being executed.

8.4 DETECTING THE DEAD LOCK

Detecting deadlocks in operating systems can be accomplished using various techniques. One of the commonly used methods is the resource allocation graph, as mentioned earlier. Here's a step-by-step guide to detecting deadlocks using the resource allocation graph:

1. **Resource Allocation Graph (RAG) Construction**:
    - Identify all the processes and resources in the system.
    - Represent each process as a node in the graph.
    - Represent each resource as a node in the graph.
    - Add directed edges (arcs) between the process nodes and resource nodes to indicate resource requests and allocations.
2. **Deadlock Detection Algorithm:**
    - Run a deadlock detection algorithm periodically or whenever there is a significant change in resource allocation.
    - One such algorithm is the cycle detection algorithm, which checks if there is a cycle in the resource allocation graph.
    - Another commonly used algorithm is the Banker's algorithm, which checks for the safe sequence to allocate resources and detects if a deadlock exists.
3. **Cycle Detection Algorithm:**
    - Perform a depth-first search (DFS) on the resource allocation graph.
    - Start the DFS from each unmarked process node.
    - While performing the DFS, mark the visited process nodes and follow the arcs (edges) to traverse the graph.
    - If you encounter a marked process node during the DFS, it indicates the presence of a cycle, and thus, a deadlock.

4. **Banker's Algorithm (for Deadlock Avoidance and Detection):**
    - The Banker's algorithm is used to check if the system is in a safe state, i.e., no deadlock exists.
    - It simulates resource allocation and deallocation to determine if a safe sequence can be found to complete all processes.
    - If a safe sequence is found, the system is in a safe state, and there is no deadlock.
    - If a safe sequence cannot be found, it indicates that the system is in an unsafe state, and deadlock is possible.
5. **Deadlock Reporting and Handling:**
    - Once a deadlock is detected, the operating system can take appropriate actions based on the chosen deadlock handling strategy.
    - This may involve resource preemption, process termination, or other methods to resolve the deadlock and restore the system to a functional state.

It's important to note that detecting deadlocks comes with some overhead, as the deadlock detection algorithm needs to be executed periodically or reactively. Therefore, the frequency and impact of deadlock detection routines should be carefully balanced to ensure system performance and responsiveness.

EXAMPLE: BANKER'S ALGORITHM

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm that can also be used for deadlock detection. It is used in systems where multiple processes compete for a finite set of resources. The algorithm keeps track of the available resources, the maximum resources each process may need, and the currently allocated resources.

Here's how the Banker's algorithm works for deadlock detection:

a. **Resource Allocation Data Structures:**
   - The system maintains several data structures:
     - **Available**: An array indicating the number of available instances of each resource type.
     - **Max**: A 2D array representing the maximum number of each resource type that each process may need.
     - **Allocation**: A 2D array representing the number of each resource type currently allocated to each process.

b. **Work and Finish Vectors:**
   - The algorithm also maintains two additional arrays:
     - **Work**: An array initially set to the same values as the **Available** array. It represents the number of available resources after considering the current allocations.
     - **Finish**: A boolean array initially set to **false**, representing the finish state of each process.

c. **Safe Sequence Determination (Deadlock Avoidance):**
   - Before a resource request is granted to a process, the Banker's algorithm checks if the system is in a safe state. A safe state means that there is at least one sequence in which all processes can complete their execution without getting stuck in a deadlock.
   - The algorithm simulates resource allocation and deallocation to check for a safe sequence using the following steps: a. initially, find an index **i** such that **Finish[i]** is **false**, and **Need[i]** (the maximum needed resources - currently allocated resources for process **i**) is less than or equal to **Work**. b. If such an index **i** is found, add the **Allocation[i]** to the **Work** array and mark the process as finished (**Finish[i] = true**). c. Repeat steps a and b until no such index **i** is found, or all processes are marked as finished (**Finish[i]** is **true** for all **i**). d. If all processes are marked as finished, the system is in a

safe state, and the requested resource can be granted. If not, the system is in an unsafe state, and the resource request is denied to avoid potential deadlock.

d. **Deadlock Detection:**
   - If the Banker's algorithm finds that the system is in an unsafe state while granting a resource request, it indicates that a deadlock may occur if the request is allowed.
   - In this case, the system can choose to use other deadlock detection algorithms, such as cycle detection in the resource allocation graph, to confirm the presence of a deadlock.

The Banker's algorithm is a useful tool for detecting potential deadlocks in a system and making informed decisions about resource allocation to avoid deadlock situations.

8.5 PREVENTING THE DEADLOCK

Preventing deadlocks in operating systems involves implementing strategies and policies during system design and resource allocation to ensure that deadlocks cannot occur. Here are some effective methods for preventing deadlocks:

1. **Resource Allocation Order:**
   - Define a global ordering of resources and require processes to request resources in a specific order.
   - This approach prevents circular waits, as processes can only request resources in a way that respects the predefined order.
2. **Resource Limitation:**
   - Set a limit on the maximum number of resources each process can hold simultaneously.
   - By limiting the number of resources that a process can have, the chances of deadlocks are reduced.

3. **Two-Phase Locking:**
    - In concurrent systems, use the two-phase locking protocol to ensure that processes acquire and release locks on resources in a coordinated manner.
    - This method helps prevent deadlocks caused by multiple processes competing for the same resources.
4. **Transaction Serialization:**
    - In database management systems, use transaction serialization techniques to ensure that concurrent transactions do not interfere with each other, reducing the risk of deadlocks.
5. **Spooling:**
    - Implement spooling, where processes use an intermediate storage area (spool) for holding data or results before using a shared resource.
    - Spooling reduces the contention for shared resources and minimizes the chances of deadlocks.
6. **Timeouts and Resource Reclamation:**
    - Set timeouts for resource requests, and if a process cannot acquire the required resources within a specified time, release all its allocated resources and retry later.
    - Reclaim unused or idle resources from processes to avoid resource wastage and increase the availability of resources.
7. **Resource Request Protocol:**
    - Implement a resource request protocol where processes explicitly declare their resource needs and release resources they no longer require.
    - This helps in efficient resource management and reduces the chances of deadlocks.
8. **Deadlock Detection and Recovery:**
    - Although prevention is the ideal approach, incorporating deadlock detection mechanisms is still valuable.

- Detection mechanisms can be used to periodically check for deadlocks and take appropriate actions to resolve them if they occur.

It's essential to carefully design the system and resource allocation mechanisms with deadlock prevention in mind. Prevention techniques may slightly impact system performance and resource utilization, but they are generally preferred over deadlock resolution strategies, which often involve higher overhead and can lead to process termination or resource preemption.

8.6 DEAD LOCK AVOIDANCE

Deadlock avoidance is a different concept and involves making resource allocation decisions in a way that guarantees the system will never enter a deadlock state. The primary goal of deadlock avoidance is to allocate resources safely to processes to prevent any possibility of a deadlock.

One of the well-known techniques for deadlock avoidance is the "Resource Allocation Graph" (RAG) and "Deadlock Avoidance Algorithm." Here's how deadlock avoidance works using these methods:

1. **Resource Allocation Graph (RAG):**
    - The RAG is a graphical representation of the resources and processes in the system, similar to the one used for deadlock detection.
    - The graph consists of process nodes, resource nodes, request edges (representing resource requests), and assignment edges (representing resource allocations).
2. **Deadlock Avoidance Algorithm:**
    - The deadlock avoidance algorithm continually analyzes the resource allocation graph and makes decisions on resource allocation based on certain criteria to ensure deadlock avoidance.
3. **Safe State:**
    - The algorithm maintains the concept of a "safe state," which is a state where there exists a sequence of

resource allocations that allows all processes to complete successfully without encountering a deadlock.

- If the system is in a safe state, resource allocations can proceed.
- If the system is not in a safe state, the algorithm avoids further resource allocations until a safe state is achievable.

4. **Resource Allocation Decisions:**
    - Before allocating resources to a process, the algorithm predicts the potential effects of the allocation on the system's state.
    - If the allocation will lead to an unsafe state (risking a deadlock), the allocation is postponed until it is safe to proceed.
    - If the allocation will maintain or lead to a safe state, the resources are allocated to the requesting process.
5. **Process Prioritization:**
    - The deadlock avoidance algorithm may prioritize processes based on factors such as the number of resources they currently hold, their maximum resource needs, and their priority levels to make optimal resource allocation decisions.

It's important to note that deadlock avoidance requires more sophisticated and dynamic decision-making than deadlock prevention. The advantage of deadlock avoidance is that it allows for more flexible resource allocation, but it also comes with more computational overhead and complexity. Implementing an efficient and effective deadlock avoidance algorithm requires a thorough understanding of the system's resource demands and characteristics.

8.7 **DEADLOCK RECOVERY**

Deadlock recovery is the process of resolving a deadlock after it has occurred in an operating system. When a deadlock is detected or suspected, the system takes appropriate actions to break the deadlock and restore normal functioning. There are several methods for deadlock recovery, each with its advantages and disadvantages. Here are some common deadlock recovery strategies:

1. **Process Termination:**
    - One of the simplest deadlock recovery methods is to terminate one or more processes involved in the deadlock.
    - The terminated processes release their held resources, allowing the remaining processes to continue execution and preventing further deadlocks.
    - The operating system can choose which process(es) to terminate based on certain criteria such as priority or resource usage.
2. **Resource Preemption:**
    - In resource preemption, the operating system forcibly takes resources from one or more processes to break the deadlock.
    - The preempted resources are then allocated to other waiting processes.
    - Preemption can be either partial or complete, depending on the severity of the deadlock and the criticality of the processes involved.
3. **Rollback and Restart:**
    - In some systems, the state of the entire system can be check pointed periodically.
    - When a deadlock is detected, the system can roll back to a previously saved checkpoint and restart the processes from that point, avoiding the conditions that led to the deadlock.
4. **Process Prioritization:**
    - The operating system can prioritize certain processes over others based on their importance or criticality.

- In case of a deadlock, the higher-priority processes are allowed to continue, while lower-priority processes might be terminated or preempted to break the deadlock.

5. **Transaction Abort and Restart:**
    - In database systems, transactions involved in a deadlock can be aborted and then restarted from the beginning.
    - By restarting the transactions, they can potentially proceed differently and avoid the conditions that caused the deadlock.
6. **Resource Manager Intervention:**
    - In distributed systems or systems with multiple resource managers, deadlock recovery may involve intervention from the resource managers.
    - Resource managers may collectively analyze the deadlock and resolve it by releasing or reallocating resources across the system.

It's important to note that deadlock recovery is a reactive approach and typically involves overhead and potential loss of progress. The choice of deadlock recovery strategy depends on the system's requirements, the criticality of the processes involved, and the potential impact on data integrity and system performance. Prevention and avoidance methods are generally preferred over recovery to minimize the occurrence of deadlocks and avoid the need for drastic actions to resolve them.

Deadlocks:

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

System model:

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. A process may utilize a resource in only the following sequence:

1. **Request:** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use.** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release.** The process releases the resource.

Deadlock Characterization:

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** only one process at a time can use a resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock Example:

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
  pthread_mutex_lock(&first_mutex);
  pthread_mutex_lock(&second_mutex);
  /** * Do some work */
  pthread_mutex_unlock(&second_mutex);
```

```
        pthread_mutex_unlock(&first_mutex);
        pthread_exit(0);
    }
    /* thread two runs in this function */
    void *do_work_two(void *param)
    {
        pthread_mutex_lock(&second_mutex);
        pthread_mutex_lock(&first_mutex);
        /** * Do some work */
        pthread_mutex_unlock(&first_mutex);
        pthread_mutex_unlock(&second_mutex);
        pthread_exit(0);
    }
```

Resource-Allocation Graph:

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.

This graph consists of a set of vertices V and a set of edges E.

V is partitioned into two types:

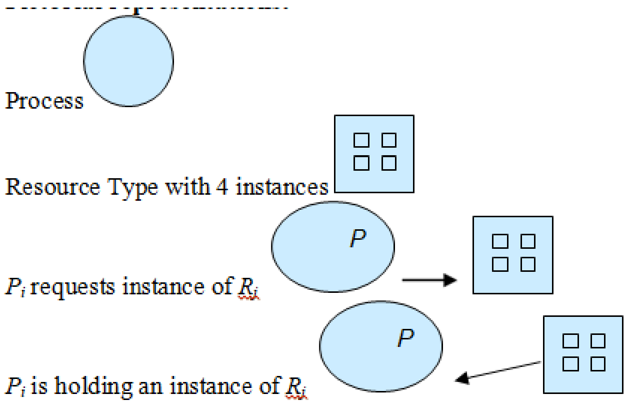
$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

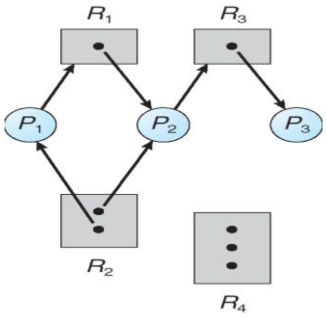
request edge – directed edge $P_i \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow P_i$

Pictorial representations:


Process
Resource Type with 4 instances
P
P_i requests instance of R_j
P
P_i is holding an instance of R_j

Example of a Resource Allocation Graph



The sets P, R, and E: ◦ P = {P1, P2, P3}
◦ R = {R1, R2, R3, R4}
◦ E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

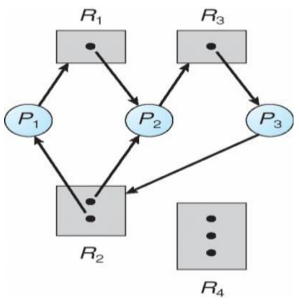
Resource instances:

◦ One instance of resource type R1
◦ Two instances of resource type R2
◦ One instance of resource type R3
◦ Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

Resource Allocation Graph With A Deadlock:

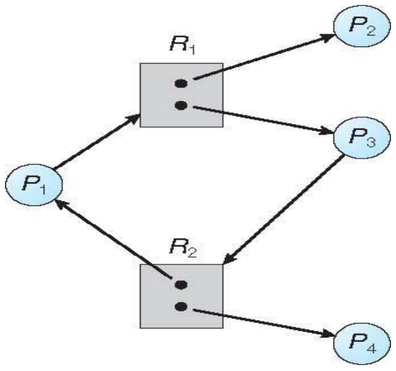


At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

Graph With A Cycle But No Deadlock:



P1 → R1 → P3 → R2 → P1

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

Methods for Handling Deadlocks:

- Ensure that the system will *never* enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention:

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources.
2. **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
Two main disadvantages: Low resource utilization; starvation possible

3. **No Preemption** –
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
    - Preempted resources are added to the list of resources for which the process is waiting.
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
4. **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Example with Lock Ordering:

```
void transaction(Account from, Account to, double amount)
{
mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);
  acquire(lock1);
    acquire(lock2);
      withdraw(from, amount);
      deposit(to, amount);
    release(lock2);
  release(lock1);
}
```
Deadlock is possible if two threads simultaneously invoke the transaction() function, transposing different accounts. That is, one thread might invoke transaction(checking account, savings account, 25); and another might invoke transaction(savings account, checking account, 50);

Deadlock Avoidance:

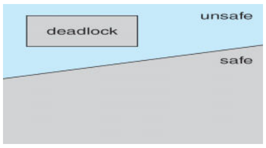
Requires that the system has some additional ***a priori*** information available:

- o Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- o The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- o Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State:

- o When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- o System is in **safe state** if there exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- o That is: If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- o When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate

- When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.
- **Safe, Unsafe, Deadlock State**



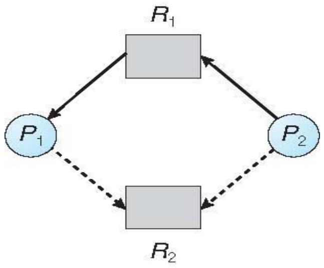
DEADLOCK AVOIDANCE ALGORITHMS:

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

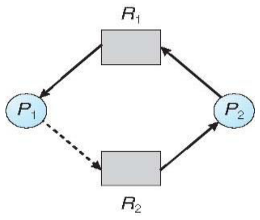
1.Resource-Allocation-Graph Algorithm:

- In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge**.
- A claim edge $Pi \rightarrow Rj$ indicates that process Pi may request resource Rj at some time in the future.
- This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process Pi requests resource Rj , the claim edge $Pi \rightarrow Rj$ is converted to a request edge.
- Similarly, when a resource Rj is released by Pi , the assignment edge $Rj \rightarrow Pi$ is reconverted to a claim edge $Pi \rightarrow Rj$.
- Note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.

- We can relax this condition by allowing a claim edge *Pi* → *Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.
- Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph.
- We check for safety by using a cycle-detection algorithm.
- An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where *n* is the number of processes in the system.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process *Pi* will have to wait for its requests to be satisfied.
- To illustrate this algorithm, we consider the resource-allocation graph



- Suppose that *P*2 requests *R*2. Although *R*2 is currently free, we cannot allocate it to *P*2, since this action will create a cycle in the graph.



- A cycle, as mentioned, indicates that the system is in an unsafe state. If *P*1
- requests *R*2, and *P*2 requests *R*1, then a deadlock will occur.

2.BANKER'S ALGORITHM:

- For Multiple instances of a resource type, we have to use Banker's algorithm.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Data Structures for the Banker's Algorithm:

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where *n* is the number of
processes in the system and *m* is the number of resource types:

- **Available**. A vector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] equals *k,* then *k* instances of resource type *Rj* are available.
- **Max**. An *n* × *m* matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k,* then process *Pi* may request at most *k* instances of resource type *Rj* .
- **Allocation**. An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k,* then process *Pi* is currently allocated *k* instances of resource type *Rj* .

- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***$[i][j]$ equals k, then process *Pi* may need k more instances of resource type *Rj* to complete its task. Note that ***Need***$[i][j]$ equals***Max***$[i][j]$− ***Allocation***$[i][j]$.

Safety Algorithm:

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available*** and ***Finish***$[i]$ = ***false*** for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. ***Finish***$[i]$ == ***false***
   b. ***Needi*** ≤***Work***
   If no such i exists, go to step 4.
3. ***Work*** =***Work*** + ***Allocationi***
   Finish$[i]$ = ***true***
   Go to step 2.
4. If ***Finish***$[i]$ == ***true*** for all i, then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Resource-Request Algorithm:

Let ***Requesti*** be the request vector for process *Pi* . If ***Requesti*** $[\, j]$ == k, then process *Pi* wants k instances of resource type *Rj* . When a request for resources is made by process *Pi* , the following actions are taken:

1. If ***Requesti*** ≤***Needi*** , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Requesti*** ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

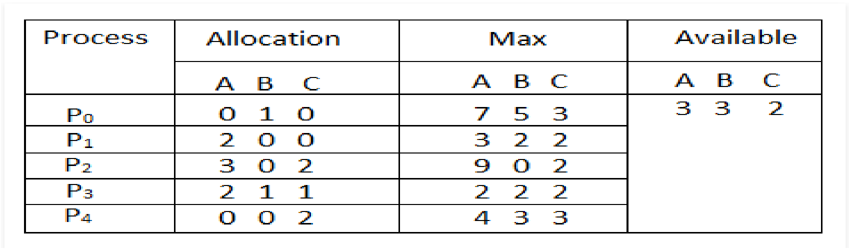
   Available = ***Available–Requesti*** ;

$$\boldsymbol{Allocationi = Allocationi + Requesti}\ ;$$
$$\boldsymbol{Needi = Needi - Requesti}\ ;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for ***Requesti*** **,** and the old resource-allocation state is restored.

Example:

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

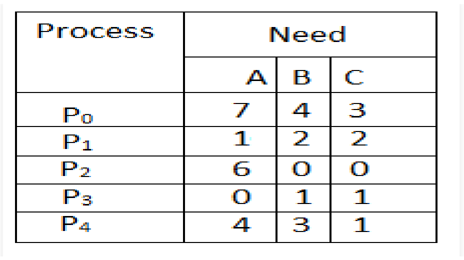


Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Question1. What will be the content of the Need matrix?

Need [i, j] = Max [i, j] – Allocation [i, j]

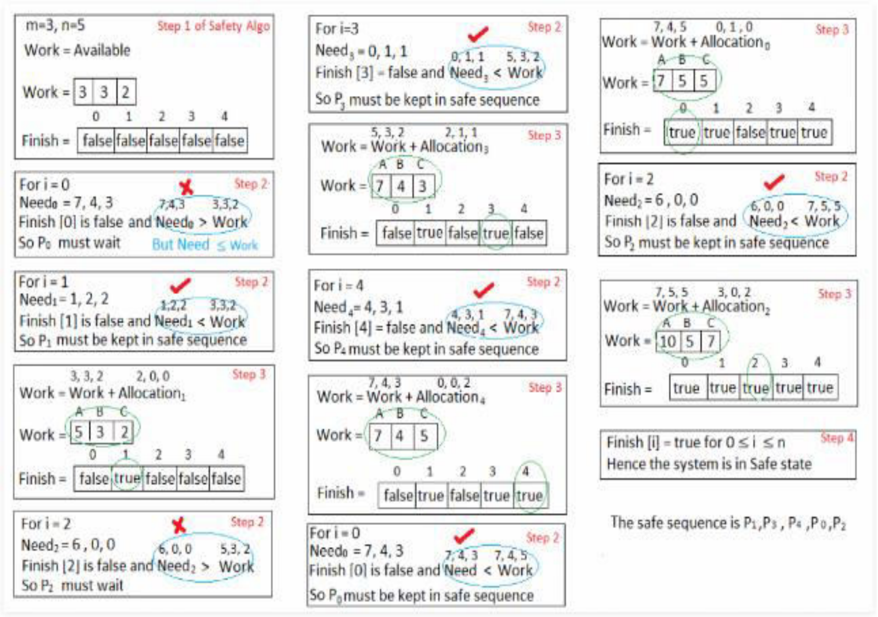
So, the content of Need Matrix is:



Process	Need		
	A	B	C
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

Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,



m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$

So P_1 must be kept in safe sequence

Step 3

$\overset{3,3,2}{Work} = Work + \overset{2,0,0}{Allocation_1}$

Work = | 5 | 3 | 2 | (A B C)

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} > \overset{5,3,2}{Work}$

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,1,1}{Need_3} < \overset{5,3,2}{Work}$

So P_3 must be kept in safe sequence

Step 3

$\overset{5,3,2}{Work} = Work + \overset{2,1,1}{Allocation_3}$

Work = | 7 | 4 | 3 | (A B C)

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $\overset{4,3,1}{Need_4} < \overset{7,4,3}{Work}$

So P_4 must be kept in safe sequence

Step 3

$\overset{7,4,3}{Work} = Work + \overset{0,0,2}{Allocation_4}$

Work = | 7 | 4 | 5 | (A B C)

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need} < \overset{7,4,5}{Work}$

So P_0 must be kept in safe sequence

Step 3

$\overset{7,4,5}{Work} = Work + \overset{0,1,0}{Allocation_0}$

Work = | 7 | 5 | 5 | (A B C)

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0 , 0

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

$\overset{7,5,5}{Work} = Work + \overset{3,0,2}{Allocation_2}$

Work = | 10 | 5 | 7 | (A B C)

	0	1	2	3	4
Finish =	true	true	true	true	true

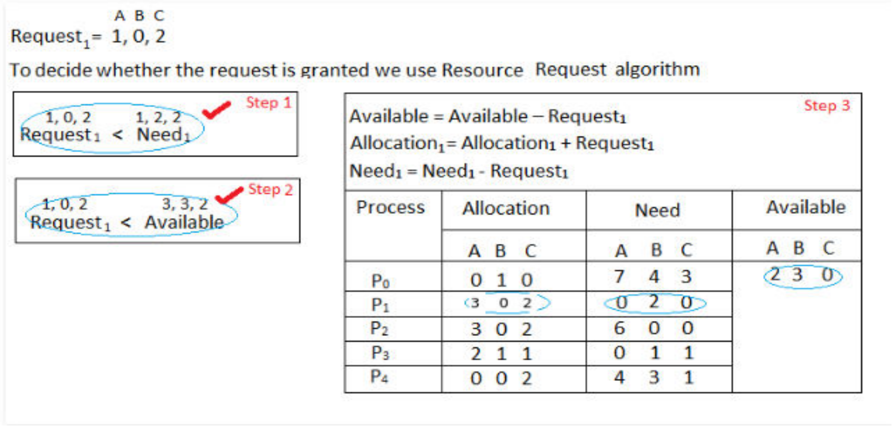
Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?



$Request_1 = \overset{A\ B\ C}{1, 0, 2}$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$\overset{1,0,2}{Request_1} < \overset{1,2,2}{Need_1}$

Step 2

$\overset{1,0,2}{Request_1} < \overset{3,3,2}{Available}$

Step 3

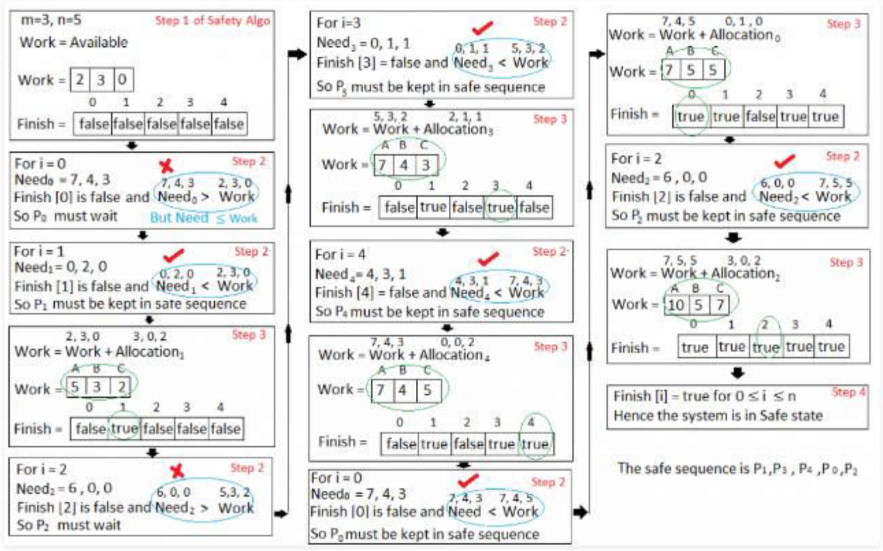
Available = Available – $Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.


m=3, n=5
Step 1 of Safety Algo
Work = Available
Work = 2 3 0
Finish = false false false false false
For i = 0
Step 2
Need0 = 7, 4, 3
Finish [0] is false and Need0 > Work
So P0 must wait
But Need ≤ Work
For i = 1
Need1 = 0, 2, 0
Finish [1] is false and Need1 < Work
So P1 must be kept in safe sequence
Step 3
Work = Work + Allocation1
Work = 5 3 2
Finish = false true false false false
For i = 2
Need2 = 6, 0, 0
Finish [2] is false and Need2 > Work
So P2 must wait
For i=3
Need3 = 0, 1, 1
Finish [3] = false and Need3 < Work
So P3 must be kept in safe sequence
Work = Work + Allocation3
Work = 7 4 3
Finish = false true false true false
For i = 4
Need4 = 4, 3, 1
Finish [4] = false and Need4 < Work
So P4 must be kept in safe sequence
Work = Work + Allocation4
Work = 7 4 5
Finish = false true false true true
For i = 0
Need0 = 7, 4, 3
Finish [0] is false and Need < Work
So P0 must be kept in safe sequence
Work = Work + Allocation0
Work = 7 5 5
Finish = true true false true true
For i = 2
Need2 = 6, 0, 0
Finish [2] is false and Need2 < Work
So P2 must be kept in safe sequence
Work = Work + Allocation2
Work = 10 5 7
Finish = true true true true true
Step 4
Finish [i] = true for 0 ≤ i ≤ n
Hence the system is in Safe state
The safe sequence is P1,P3 , P4 ,P0,P2

Hence the new system state is safe, so we can immediately grant the request for process $\mathbf{P_1}$.

Deadlock Detection:

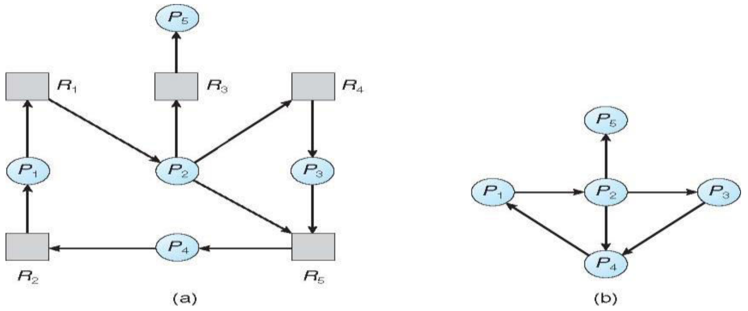
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type:

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.

- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of $\boldsymbol{n}^2$ operations, where $\boldsymbol{n}$ is the number of vertices in the graph
- Nodes are processes
- $\boldsymbol{P_i \rightarrow P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$



Several Instances of Resource type:

- **Available**.Avector of length*m*indicates the number of available resources of each type.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request**. An $n \times m$ matrix indicates the current request of each process. If ***Request***[*i*][*j*] equals *k,* then process *Pi* is requesting *k* more instances of resource type *Rj* .

Detection Algorithm:

1.**Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:**

(a) *Work* = *Available*

(b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = *false*; otherwise, *Finish*[i] = *true*

2.Find an index *i* such that both:

(a) *Finish*[*i*] == *false*

(b) $Request_i \leq Work$

If no such *i* exists, go to step 4

3.*Work* = *Work* + *Allocation*$_i$

Finish[*i*] = *true*

go to step 2

4.If *Finish[i]* == *false*, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Example of Detection algorithm:

we consider a system with five processes *P*0 through *P*4 and three resource types *A, B,* and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances. Suppose that, at time *T*0, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*P*0, *P*2, *P*3, *P*1, *P*4> results in ***Finish***[*i*] == *true* for all *i.*

Suppose now that process *P*2 makes one additional request for an instance of type *C*. The ***Request*** matrix is modified as follows:

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

Detection-Algorithm Usage:

- When, and how often, to invoke the detection algorithm depends on:
  - How often is a deadlock likely to occur?
  - How many processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken. The no of processes involved in deadlock cycle may grow.
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock:

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system **recover** from the deadlock automatically. There are two options for breaking a deadlock.
- One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

- Abort all deadlocked processes – great expensive
- Abort one process at a time until the deadlock cycle is eliminated (minimum cost)
- In which order should we choose to abort?
  1. Priority of the process

2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

CHAPTER 9: PROCESS SYNCHRONIZATION

Process synchronization is essential when two or more cooperative processes interact with each other. If their order of execution is not properly managed, conflicts can arise, leading to incorrect outputs. Cooperative processes are those that can influence or be influenced by the execution of other processes. To ensure the appropriate order of execution for these processes, synchronization mechanisms are employed. These mechanisms enable coordination and orderly interaction among processes, preventing issues like data inconsistency and conflicts. Some common synchronization techniques are used to achieve this coordination.

RACE CONDITION:

The behavior or output of a program becomes dependent on the relative timing of events, such as the sequence in which numerous processes or threads are run, in an operating system (OS) or in any concurrent computing system. It occurs when two or more processes or threads access crucial or shared resources without proper coordination, resulting in unpredictable and unwanted outcomes.

As the processes or threads are "competing" to access the shared resource, the term "race" underscores the unpredictable nature of the outcome. Depending on how quickly each process or thread executes, a race condition's outcome may change each time the program is run.

Race conditions can lead to various issues, such as:

- **Data Corruption:** When multiple processes or threads try to write to a shared resource simultaneously, data integrity can be compromised. The final value of the shared data may be incorrect or inconsistent.

- **Deadlocks:** In certain cases, race conditions can lead to deadlock situations where processes or threads are stuck waiting for resources indefinitely.
- **Resource Starvation:** Some processes might get stuck in an infinite loop trying to access a shared resource, preventing other processes from executing and causing resource starvation.
- **Inappropriate Outputs:** Race conditions can lead to unexpected and incorrect results being produced, affecting the overall correctness and reliability of the program.

CRITICAL SECTION:

A critical section is a specific area of code or a region of a program where common resources (such variables, data structures, or devices) are accessed and modified by several processes or threads (in concurrent programming and operating systems). The concept of the critical section is used to guarantee that only one process or thread can run this portion at once, avoiding race situations and preserving data consistency.

The critical section problem occurs in systems with multiple processes or threads because multiple concurrent processes or threads may attempt to access the same shared resources at the same time. This can result in difficulties like data corruption, race situations, and other concurrency-related concerns if it is not controlled effectively.

To ensure mutual exclusion in the critical section, synchronization mechanisms are used. Common synchronization techniques include:

A mutex (also known as mutual exclusion) is a lock that only permits one process or thread to access the crucial portion at once. A process must obtain the mutex before proceeding to the vital area. The requesting process will be stopped until the mutex is released if it is already being held by another process.

An integer variable called **a semaphore** is used to manage access to shared resources. It can be utilized to permit just a specified number of threads or processes to enter the vital region at once.

A mutex, condition variables, and shared data are all combined into a monitor, which is a higher-level synchronization architecture. It guarantees that only one process or thread can carry out **the monitor** operations at once, making it simple to synchronize access to shared resources.

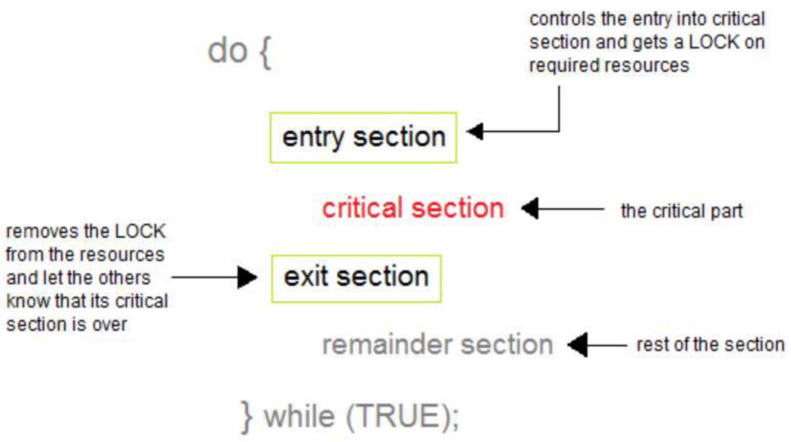
Critical Section Problem in Operating System

The section of a program known as the Critical Section attempts to access shared resources. Any resource in a computer, such as a memory region, data structure, CPU, or any IO device, could constitute that resource.

The operating system has trouble authorizing and preventing processes from entering the critical part because more than one process cannot operate in the critical section at once.

A set of procedures that can guarantee that the Race condition among the processes will never exist are designed using the critical section problem.

Our main objective is to resolve the critical section problem in order to synchronize the cooperative processes. In order to satisfy the following requirements, a solution must be offered.


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

A Critical Section is a section of code that must be run in one atomic operation that accesses shared variables. It means that just one process must be doing its crucial portion at any given time in a group of processes that are cooperating. Any subsequent processes must wait until the first one is finished before beginning their key sections. The **wait()** and **signal()** functions are primarily responsible for controlling the entry into and exit from the crucial region.

Section : Entry

- The method primarily asks for entry into the important part in this section.

Section : Exit

- The crucial section comes after this one.
- The remedy for the Critical Section Issue

The following three criteria must be met by a solution to the critical section problem:

MUTUAL EXCLUSION

Only one process at a time can be in its critical phase out of a set of cooperating processes.

Progress

Any one of these threads must be permitted to enter a process' critical part if there is no process in that section already and one or more threads desire to execute that section.

Bounded Waiting

There is a restriction on how many other processes can enter their crucial sector once a process requests access to it, before this process's request is approved. The system must thus give the process permission to enter its crucial portion once the limit has been reached.

Lock Variable

The simplest method of synchronization is this one. This software mechanism is one that has been set up in user mode. It is possible to utilize this busy waiting method for more than two processes.

An Lock variable lock is utilized in this method. There are only two potential lock values: 0 and 1. The crucial sector is either unoccupied or occupied depending on the lock value, which ranges from 0 to 1.

A procedure that wishes to enter the crucial section first verifies the lock variable's value. If it is 0, it waits; if it is 1, it sets the lock value to 1 and enters the crucial section.

```
Entry Section →
While (lock! = 0);
Lock = 1;
//Critical Section
Exit Section →
Lock =0;
```

The Pseudo Code has three components, as can be seen by looking at it. Sections for entry, critical thinking, and departure.

The lock variable's initial value is 0. The process enters the entrance section and verifies the condition specified in the while loop when it has to enter the crucial part.

The while loop implies that the process will wait indefinitely until the lock value is 1 before proceeding. The process will enter the crucial section by setting the lock variable to 1 because the critical section is empty the first time.

When the process leaves the crucial area, it resets the value of the lock to 0 in the exit section.

Mutual Exclusion

In some circumstances, the lock variable mechanism may not offer mutual exclusion. Looking at the program's pseudocode from the Operating System's perspective, or its Assembly code, will help to better explain this. The Code will now be translated into assembly language.

Test Set Lock Mechanism

the lock's adjustable mechanism On occasion, the process examines the lock variable's previous value and reaches the crucial area. This makes it possible for many processes to enter the crucial area. The code provided in part two of the following section can, however, be used in place of the code presented in part one. Although the method is unaffected, we may ensure mutual exclusion somewhat but not entirely by doing this.

The value of Lock is put into the local register R0 and set to 1 in the revised version of the code.

But in step 3, the prior lock value—which is now stored in R0—is contrasted with 0.

Section 1	Section 2
1. Load Lock, R0 2. CMP R0, #0 3. JNZ step1 4. store #1, Lock	1. Load Lock, R0 2. Store #1, Lock 3. CMP R0, #0 4. JNZ step 1

Solutions for Critical Section Problem

Process synchronization depends heavily on the vital part, hence the issue has to be fixed.

The following are a few popular approaches to the critical section problem:

Peterson's Solution

This is a well-known, software-based fix for crucial section issues. The reason it is termed Peterson's solution is because computer scientist Peterson created it.

With this approach, anytime a process is running in a critical condition, the other process merely runs the remaining code, and vice versa. This technique also aids in ensuring that only one process may be active in the crucial region at any given moment.

All three requirements are maintained by this answer:

- The comfort of Mutual Exclusion comes from the fact that only one process can ever access the crucial area.

- A process outside the crucial area cannot prevent other processes from approaching the critical section, which is another consoling aspect of progress.
- Because every process has an equal opportunity to access the Critical section, bounded waiting is guaranteed.

Peterson's solution is a software-based solution to the critical section problem in operating systems. The critical section problem is a synchronization problem that arises when multiple processes need to access a shared resource. If the processes access the shared resource at the same time, it can lead to data corruption or other problems.

Peterson's solution solves the critical section problem by using two shared variables:

* **`flag[i]`:** A boolean variable that indicates whether process `i` is interested in entering the critical section.

*** `turn`:** An integer variable that indicates which process has the right to enter the critical section.

The following pseudocode shows how Peterson's solution works:

```
procedure critical_section(i)
  flag[i] := true
  turn := j
  while flag[j] and turn = j do
    continue
  // critical section
  flag[i] := false
end procedure
```

The first step is for process `i` to set `flag[i]` to `true`. This indicates that process `i` is interested in entering the critical section. Then, process `i` sets `turn` to the identifier of the other process. This ensures that only one process can enter the critical section at a time.

Next, process `i` enters a loop. In the loop, process `i` checks if `flag[j]` is `true` and if `turn` is equal to `j`. If both of these conditions are true, then it means that process `j` is also interested in entering the critical section. In this case, process `i` simply continues to loop.

If `flag[j]` is not `true` or if `turn` is not equal to `j`, then process `i` can enter the critical section. Once process `i` is in the critical section, it can access the shared resource.

When process `i` is finished with the critical section, it sets `flag[i]` to `false`. This indicates that process `i` is no longer interested in entering the critical section.

Peterson's solution is a simple and elegant solution to the critical section problem. It is also deadlock-free, meaning that no two processes will ever be deadlocked waiting to enter the critical section.

However, Peterson's solution does have some limitations. It can only be used for two processes, and it can be inefficient in some cases. Despite these limitations, Peterson's solution is a valuable tool for understanding and solving the critical section problem.

A mutex lock

Due to the difficulty of implementing the synchronization hardware solution, Mutex Locks, a rigid software technique, was developed. According to this method, a LOCK is gained over the vital resources that are updated and utilized inside the crucial portion of code in the entrance part, and it is released in the exit section.

No other process may access the resource since it is locked while a process is completing its crucial portion.

Classical Synchronization Problem:

Classical synchronization problems are a set of problems that arise in concurrent programming when multiple processes need to access a shared resource. These problems can lead to data corruption or other errors if they are not solved correctly.

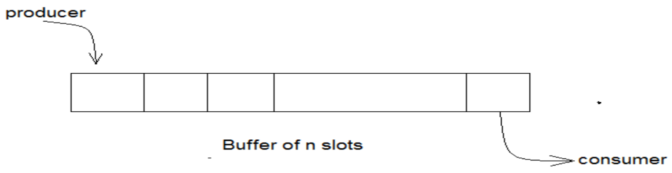
Some of the most well-known classical synchronization problems include:

- **The critical section problem:** This problem arises when multiple processes need to access a shared resource, but only one process can access the resource at a time.

- **The producer-consumer problem:** This problem arises when one process (the producer) produces data and another process (the consumer) consumes data. The producer and consumer must coordinate their access to the shared data so that the producer does not produce more data than the consumer can consume.
- **The readers-writers problem:** This problem arises when there are multiple processes that need to access a shared resource, but some processes only need to read the resource and others need to write to the resource. The readers and writers must coordinate their access to the shared resource so that the readers do not interfere with the writers.
- **The dining philosopher's problem:** This problem is a classic example of a deadlock. Five philosophers are sitting around a table, and each philosopher has a chopstick in each hand. A philosopher can only eat if they have both chopsticks. If all five philosophers try to pick up their chopsticks at the same time, they will all deadlock.
- **Bounded Buffer Problem**: One of the most well-known synchronization issues is the bounded buffer problem, often known as the producer-consumer problem. Before we move on to the answer and the computer code, let's first clarify the issue at hand.

Problem Statement:

Each slot in the buffer, which has n slots, may hold one unit of data. Producer and consumer, two processes that use the buffer, are now active.



A producer attempts to insert data into a buffer slot that is empty. Data is attempted to be removed from a full slot in the buffer by a consumer. As you should have figured by now, if those two procedures are run simultaneously, the outcome won't be what was anticipated.

Solution for Bounded Buffer Problem / Producer Consumer Problem:

```
class Producer(threading.Thread):
    def __init__(self, queue):
        super().__init__()
        self.queue = queue

    def run(self):
        for i in range(10):
            # Produce an item
            item = i
            print("Producer produced:", item)
            # Enqueue the item
            self.queue.put(item)

class Consumer(threading.Thread):
    def __init__(self, queue):
        super().__init__()
        self.queue = queue

    def run(self):
        for i in range(10):
            # Dequeue an item
            item = self.queue.get()
            print("Consumer consumed:", item)

# Create a queue
queue = threading.Queue()

# Create a producer and consumer
producer = Producer(queue)
```

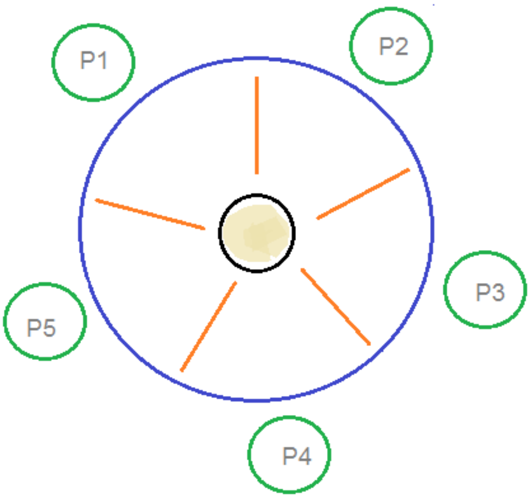
```
consumer = Consumer(queue)

# Start the producer and consumer
producer.start()
consumer.start()

# Wait for the producer and consumer to finish
producer.join()
consumer.join()
```

Dining Philosopher Problem

Another well-known synchronization problem that is used to assess circumstances in which it is necessary to distribute resources across several processes is the dining philosophers problem.

Problem Statement:

Imagine a dining table with a circle in the middle and five philosophers seated there. As seen in the illustration below, the dinner table contains five chopsticks and a bowl of rice in the center.



A philosopher is either eating or thinking at any given time. A philosopher uses one chopstick from their left and one from their right when he wants to eat. A philosopher places both chopsticks back where they were when he wants to reflect.

Solution for Dining Philosophers Problem:

```
# Create semaphores for the chopsticks
chopsticks = [threading.Semaphore(1) for _ in range(5)]

# Create philosophers
philosophers = []
for i in range(5):
    philosopher = threading.Thread(target=philosophize, args=(i,
chopsticks))
    philosophers.append(philosopher)

# Start the philosophers
for philosopher in philosophers:
    philosopher.start()

# Wait for the philosophers to finish
for philosopher in philosophers:
    philosopher.join()

def philosophize(i, chopsticks):
    while True:
        # Think
        print("Philosopher {} is thinking.".format(i))
        time.sleep(random.random())

        # Try to acquire the chopsticks
        left_chopstick = chopsticks[i]
        right_chopstick = chopsticks[(i + 1) % 5]
        if left_chopstick.acquire(blocking=True)  and
right_chopstick.acquire(blocking=True):  # Eat
            print("Philosopher {} is eating.".format(i))
            time.sleep(random.random())

            # Release the chopsticks
            left_chopstick.release()
            right_chopstick.release()
        else:
            # Unable to acquire chopsticks, keep thinking
            print("Philosopher {} is unable to acquire chopsticks, keep thinking.".format(i))
```

Reader Writer Problem:

Another typical synchronization issue is the readers-writers dilemma. This issue has several variations, one of which is discussed here.

Problem Statement:

There is a common resource that several processes should access. In this context, there are two different sorts of processes. Both of them read and write. The shared resource can be used simultaneously by any number of readers, but only one writer can make changes to it. No other process may access the resource while a writer is writing data to it. If there are more readers accessing the resource than zero at any given moment, the writer cannot update the resource.

Solution for Reader Writer Problem:

```
# Create a semaphore for the shared resource
mutex = threading.Semaphore(1)
# Create readers and writers
readers = []
writers = []
# Start the readers and writers
for i in range(10):
    if i % 2 == 0:
        reader = threading.Thread(target=read, args=(i,))
        readers.append(reader)
    else:
        writer = threading.Thread(target=write, args=(i,))
        writers.append(writer)
for reader in readers:
    reader.start()
for writer in writers:
    writer.start()
def read(i):
    # Acquire the mutex
    mutex.acquire()
    # Do the reading
    print("Reader {} is reading.".format(i))
    time.sleep(random.random())
    # Release the mutex
    mutex.release()
def write(i):
    # Acquire the mutex
    mutex.acquire()
    # Do the writing
    print("Writer {} is writing.".format(i))
    time.sleep(random.random())
    # Release the mutex
    mutex.release()
```

SEMAPHORES

By leveraging the value of a straightforward integer variable to synchronize the advancement of interacting processes, Dijkstra presented a novel and very significant method for controlling concurrent processes in 1965. We refer to this integer variable as a semaphore. Therefore, it primarily functions as a synchronization tool and can only be accessed via the low-standard atomic operations wait and signal, which are denoted by P(S) and V(S), respectively.

The semaphore, which is used by all threads and has the actions wait and signal, can only contain a non-negative integer value, and it works as follows:

```
P(S): if S >= 1 then S := S - 1
      else <block and enqueue the process>;
V(S): if <some process is blocked on the queue>
        then <unblock a process>
      else S := S + 1;
```

Wait and signal are traditionally defined as follows:

When the value of its input S becomes non-negative (greater than or equal to 1), this procedure decreases its value. You may regulate the entry of a job into the important section primarily with the aid of this operation. If the value is negative or 0, no operation is carried out. The wait() operation is also known as a P(S) operation since it was initially referred to as a P operation.

Specifications of semaphores

It is straightforward and always has an integer value that is not zero.

- Uses a variety of procedures.
- May contain a variety of distinct crucial sections and semaphores.
- There are distinct access semaphores for every crucial part.
- Can, if desired, allow several processes to enter the crucial area at once.

Types of Semaphores

There are two types

1. Binary semaphore
2. Counting semaphore

1.Binary Semaphore

It is frequently referred to as a Mutex since it is a unique kind of semaphore that is utilized to perform mutual exclusion. When a program is being run, a binary semaphore only accepts the values 0 and 1. It is initialized to 1. In a binary semaphore, the signal action succeeds when the semaphore's value is 0, while the wait operation only functions if the semaphore's value is 1. Using binary semaphores is simpler than using counting semaphores.

2.Counting Semaphore

Bounded concurrency is implemented using these. The Counting semaphores have an unbounded sphere of operation. These can be applied to limit access to a resource that has a limited number of instances. The number of resources that are accessible in this situation is indicated by the semaphore count. The semaphore count is automatically increased when resources are added, and decremented when resources are withdrawn. There is no mutual exclusion in counting semaphore.

```
# Create a semaphore
semaphore = threading.Semaphore(1)
# Define a function that uses the semaphore
def critical_section():
   # Acquire the semaphore
   semaphore.acquire()
   # Do something in the critical section
   print("Critical section")
   # Release the semaphore
   semaphore.release()
# Start the function in a thread
thread = threading.Thread(target=critical_section)
thread.start()
# Wait for the thread to finish
thread.join()
```

Advantages of Semaphores

- The management of resources is flexible because to semaphores.
- Semaphores are machine-independent and should be used in the microkernel's machine-independent code.
- Multiple processes cannot enter the crucial part thanks to semaphores.

- They provide simultaneous access from many threads to the crucial part.
- Semaphores are substantially more effective than certain other synchronization techniques because they rigorously adhere to the mutual exclusion principle.
- No resources are spent in semaphores due to busy waiting since processor time isn't wasted checking whether a condition is met before allowing a process to access the vital area.

Disadvantages of Semaphores

- The possibility of priority inversion, wherein low priority processes access the vital portion first and high priority processes access the critical area later, is one of the main drawbacks of semaphores.
- The Wait and Signal actions must be carried out in the appropriate sequence in order to prevent deadlocks in the semaphore.
- It is not possible to employ semaphores on a wide scale since doing so results in a loss of modularity because the wait() and signal() procedures prevent the system's organized layout from being created.
- Their usage is merely by tradition and not by law.
- A procedure might get permanently blocked if used improperly. Deadlock describes such a circumstance. In the upcoming classes, we will thoroughly examine deadlocks.

CHAPTER-10 DISK SCHEDULING ALGORITHMS

An essential part of the operating system is disk scheduling which controls how data is accessed and moved on a computer's hard disk drive (HDD) or solid-state drive (SSD). The disk scheduler chooses which requests are handled first when a process needs to read or write data to or from the disk. By lowering disk arm movement and seek time, the performance of the disk will be optimized, increasing system effectiveness.

There are various disk scheduling techniques and each has benefits and drawbacks of its own.

- **First-Come-First-Served (FCFS):** The most basic disk scheduling method handles requests in order as they are received. Although simple to construct, it may result in low performance because of long seek durations brought on by dispersed disk access.
- **Shortest Seek Time First (SSTF):** The request that is closest to the current disk head position is chosen by this algorithm. It seeks to reduce time, which enhances overall disk performance. However, for some requests that are far from the current head position, it might lead to starvation.
- **SCAN:** Also referred to as the elevator algorithm, SCAN advances the disk head in a single direction, responding to requests until it reaches the disk's edge. Then it turns around and responds to queries going the other way. While this helps minimize starvation, it could result in longer response times for requests sent at the disk's outer borders.
- **C-SCAN:** Similar to SCAN, C-SCAN advances the head in one way, but when it reaches the end of the disk, it immediately bounces back to the other end. As a result, it takes less time to fulfill requests for the disk's extreme music.

- **LOOK:** A better form of SCAN is called LOOK, which scans only up until the final request in a given direction before turning around to avoid moving to the disk's edge when there are no more requests in that direction.
- **C-LOOK:** Similar to C-SCAN, C-LOOK scans only until the last request in a particular direction and then jumps back to the other end. Which reduces the seek times for requests and enhances performance.

The unique use case and the workload parameters will determine which disk scheduling algorithm is used. Modern operating systems frequently employ sophisticated disk scheduling techniques to optimize disk access and enhance overall system performance since different algorithms may perform better in specific contexts.

First-Come-First-Served (FCFS): The most straightforward disk scheduling technique is first-come-first-served (FCFS), which treats requests as they come in. Regardless of where they are on the disk or how long it will take to complete them, the recommendations are handled in the order they were received. Let's walk through an example to see how FCFS works in practice.

Consider a disk with 200 tracks (0–199) with the following track numbers in the request queue: 93, 176, 42, 148, 27, 14,180. The read/write head is currently located at position 55. Using FCFS scheduling, we must now determine the total number of read/write head track movements.

To calculate the total number of track movements of the read/write head using the First-Come-First-Serve (FCFS) scheduling algorithm, we need to follow the disk request queue in the order they appear and calculate the distance between consecutive tracks. The current position of the read/write head is 55.

- Disk Tracks: 0 – 199
- Request Queue: 93, 176, 42, 148, 27, 14, 180
- Current Head Position: 55

Step 1: Calculate the distance between the current head position (55) and the first request (93).

Request	Distance Traveled
93	38

Step 2: Calculate the distance between each consecutive request.

Request	Distance Traveled
93	38
176	83
42	134
148	106
27	121
14	13
180	166

Step 3: Sum up the distances traveled to get the total head movements.
Total Head Movements = 38 + 83 + 134 + 106 + 121 + 13 + 166 = 661

So, the total number of track movements of the read/write head using FCFS scheduling is 661.

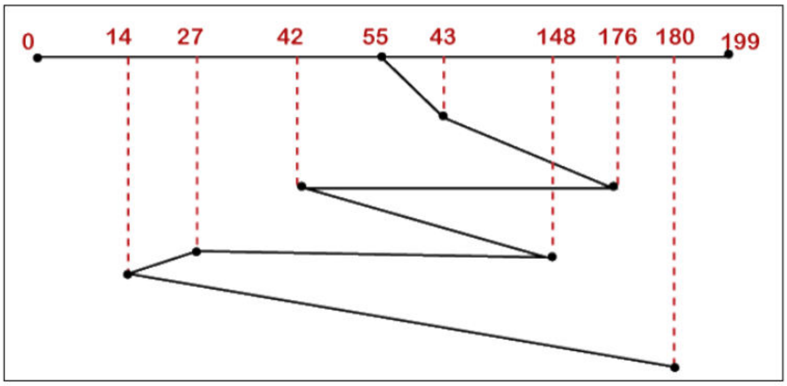


Figure: FCFS Disk Scheduling

In the above example, FCFS handles the requests in order as they come in. As a result, the disk head might need to travel far between recommendations, which could increase the seek time. Performance may suffer due to inefficient disk access, mainly if numerous requests have dispersed track numbers. FCFS is considered a naive algorithm and is rarely used in real-world scenarios where better-performing disk scheduling algorithms are available.

Shortest Seek Time First (SSTF): A disk scheduling algorithm called Shortest Seek Time First (SSTF) chooses the request with the least seek time from the disk head's present position. The time it takes the disk head to move from its current position to the track where the desired data is located is known as the seek time. By prioritizing requests close to where the disk head is currently located, SSTF seek times are reduced, and disk access performance is improved.
Let's use an illustration to clarify SSTF:

Think about a disk with 200 tracks (0–199). Tracks 82, 170, 43, 140, 24, 16, 190, and 82 are in the request queue, correspondingly. The read/write head is currently located at position 50.

- Disk Tracks: 0 – 199
- Request Queue: 82, 170, 43, 140, 24, 16, 190
- Current Head Position: 50

Step 1: Calculate the distance between the current head position (50) and each track in the request queue.

Request	**Distance to Current Head (Absolute Value)**
82	32
170	120
43	7
140	90
24	26
16	34

190	140

Step 2: Choose the track with the shortest seek time from the above table.

Request	Distance to Current Head (Absolute Value)
43	7

Step 3: Move the head to the chosen track (43).

Step 4: Remove the processed request from the queue.

Updated Request Queue: 82, 170, 140, 24, 16, 190

Step 5: Repeat Steps 1 to 4 until the request queue becomes empty.

Request	Distance to Current Head (Absolute Value)
140	97

Move the head to track 140.

Updated Request Queue: 82, 170, 24, 16, 190

Request	Distance to Current Head (Absolute Value)
24	76

Move the head to track 24.

Updated Request Queue: 82, 170, 16, 190

Request	Distance to Current Head (Absolute Value)
16	8

Move the head to track 16.

Updated Request Queue: 82, 170, 190

Request	Distance to Current Head (Absolute Value)
190	174

Move the head to track 190.

Updated Request Queue: 82, 170

Request	Distance to Current Head (Absolute Value)
170	80

Move the head to track 170.

Updated Request Queue: 82

Request	Distance to Current Head (Absolute Value)
82	88

Move the head to track 82.

Step 6: All requests have been processed, and the queue is empty.

The total number of track movements of the read/write head using SSTF disk scheduling is 7 + 97 + 76 + 8 + 174 + 80 + 88 = 530.

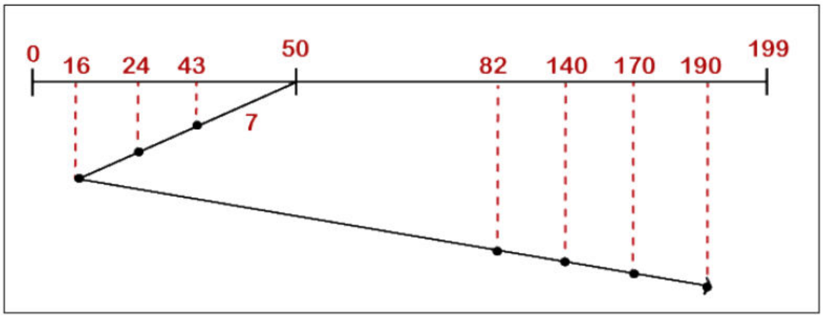


Figure: SSTF Disk Scheduling

In this example, SSTF reduces the total seek time compared to FCFS. The algorithm prioritizes requests closer to the disk head's current position, resulting in efficient disk access and improved performance. However, it's worth noting that SSTF may need more support for requests far away from the current head position. Other disk scheduling algorithms like SCAN, C-SCAN, LOOK, or C-LOOK may be used to mitigate this issue.

SCAN: A disk scheduling algorithm called SCAN, also called the elevator algorithm, moves the disk head in a single direction while responding to requests as they come in until it hits the disk's edge. As soon as it reaches the finish, it turns around and begins processing requests in the opposite direction. The back-and-forth motion resembles an elevator's action, hence the name "SCAN." SCAN is generally made to shorten seek times and prevent starvation of requests at one end of the disk.

Let's understand SCAN with an example:

Consider a disk with 200 tracks (0–199), and the track numbers 93, 176, 42, 148, 27, 14, and 180, respectively, are included in the request queue. The read/write head is currently at position 55 and pointing toward the larger value. Utilizing SCAN disk scheduling, determine the total number of cylinders that the head has moved.

- Disk Tracks: 0 – 199
- Request Queue: 93, 176, 42, 148, 27, 14, 180
- Current Head Position: 55
- Direction: Towards larger values

Step 1: Sort the request queue in ascending order.

Sorted Request Queue: 14, 27, 42, 93, 148, 176, 180

Step 2: Determine the closest requests to the current head position in the scanning direction.

In this case, the closest requests toward larger values are 93, 148, 176, and 180.

Step 3: Move the head towards the first closest request (93).

Request	**Distance Traveled**
93	38

Step 4: Move the head towards the second closest request (148).

Request	**Distance Traveled**
93	38
148	55

Step 5: Move the head towards the third closest request (176).

Request	**Distance Traveled**
93	38
148	55
176	28

Step 6: Move the head towards the fourth closest request (180).

Request	**Distance Traveled**
93	38
148	55
176	28
180	4

Step 7: The head has reached the end of the disk (track 199). Now, it changes direction.

Step 8: Move the head towards the remaining requests (42, 27, and 14) in the opposite direction.

Request	Distance Traveled
93	38
148	55
176	28
180	4
42	138
27	13
14	13

Step 9: All requests have been serviced.

The total number of track movements of the read/write head using SCAN disk scheduling is 38 + 55 + 28 + 4 + 138 + 13 + 13 = 289.

So, the total number of cylinders moved by the head using SCAN disk scheduling is 289.

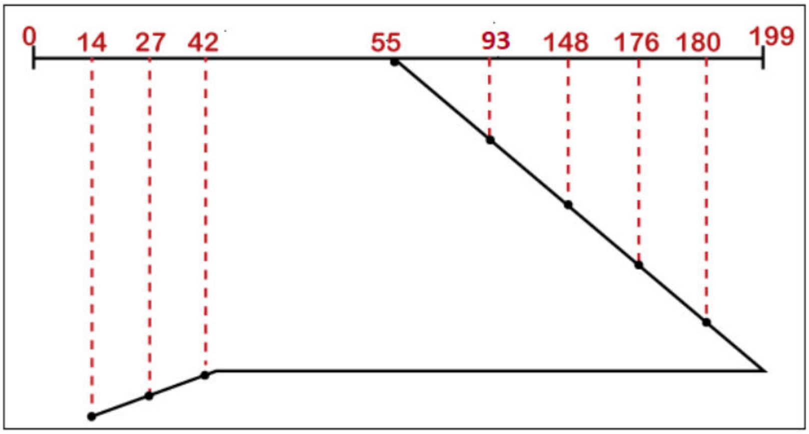


Figure: SCAN Disk Scheduling

SCAN offers a balanced disk access pattern in this instance. It effectively decreases the average seek time by handling queries in both directions. However, because SCAN handles requests linearly, it may still result in comparatively longer response times for demands placed at the disk's edges.

Variations of SCAN, such as LOOK and C-SCAN, have been developed to improve performance further. LOOK performs a similar movement to SCAN but only goes as far as the last request in a particular direction before reversing. At the same time, C-SCAN jumps back to the other end of the disk without servicing any demands on the return path, avoiding unnecessary movement to the other end of the disk. These variations address some of the limitations of the basic SCAN algorithm.

C-SCAN (CIRCULAR SCAN): A disk scheduling technique called C-SCAN (Circular SCAN) outperforms the SCAN algorithm by speeding up handling requests placed near the disk's edge. The disk head is moved by C-SCAN in a single direction, responding to queries as it goes until it reaches the disk's end. Conversely, C-SCAN leaps back to the opposite end of the disk without attending to any requests on the return path, unlike SCAN, which reverses direction. Cheating is prevented by ensuring that queries at one end of the disk do not experience lengthy wait times.

Let's understand C-SCAN with an example:

Consider that a disk has 200 tracks (0–199), and the tracks in the request queue are 82, 170, 43, 140, 24, 16, and 190, respectively. The R/W head is currently 50 and moving toward the larger value. Utilizing C-SCAN disk scheduling, determine the overall number of cylinders driven by the head.

- Disk Tracks: 0 – 199
- Request Queue: 82, 170, 43, 140, 24, 16, 190
- Current Head Position: 50
- Direction: Towards larger values

Step 1: Sort the request queue in ascending order.

Sorted Request Queue: 16, 24, 43, 82, 140, 170, 190

Step 2: Determine the closest requests to the current head position in the scanning direction.

In this case, the closest requests toward larger values are 82, 140, 170, and 190.

Step 3: Move the head towards the first closest request (82).

Request	Distance Traveled
82	32

Step 4: Move the head towards the second closest request (140).

Request	Distance Traveled
82	32
140	58

Step 5: Move the head towards the third closest request (170).

Request	Distance Traveled
82	32
140	58
170	30

Step 6: Move the head towards the fourth closest request (190).

Request	Distance Traveled
82	32
140	58
170	30
190	20

Step 7: The head has reached the end of the disk (track 199). It returns to the starting track (track 0) without servicing any requests.

Step 8: Move the head towards the remaining requests (43, 24, and 16) in the same direction.

Request	Distance Traveled
82	32
140	58
170	30
190	20
43	147
24	19
16	8

Step 9: All requests have been serviced.

The total number of track movements of the read/write head using C-SCAN disk scheduling is 32 + 58 + 30 + 20 + 147 + 19 + 8 = 314.

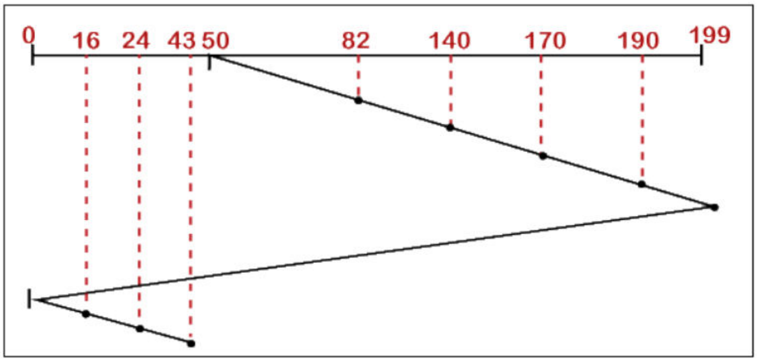


Figure: C-SCAN Disk Scheduling

So, the total number of cylinders moved by the head using C-SCAN disk scheduling is 314.

In this case, C-SCAN offers a more evenly distributed disk access pattern than SCAN. Installing moving on to the other end after reaching the end of the disk avoids the lengthy wait periods for requests located there. C-SCAN thus shortens the typical seek time and improves overall disk access performance.

LOOK: A disk scheduling method called LOOK is a variant of SCAN. When there are no more requests in the current direction, LOOK switches directions, moving the disk head in the opposite direction from SCAN, which reverses direction after reaching the end of the disk, which results in more effective disk access by preventing pointless travel to the end of the disk when no ongoing requests exist.

Let's understand LOOK with an example:

Think of a disk with 200 tracks (0–199). Tracks 82, 170, 43, 140, 24, 16, 190, and 82 are in the request queue, correspondingly. The read/write head is currently located at position 50. Towards the higher value is the direction. Utilizing look disk scheduling, determine the overall number of cylinders moved by the head.

- Disk Tracks: 0 – 199
- Request Queue: 82, 170, 43, 140, 24, 16, 190
- Current Head Position: 50
- Direction: Towards larger values

Step 1: Sort the request queue in ascending order.

Sorted Request Queue: 16, 24, 43, 82, 140, 170, 190

Step 2: Determine the closest requests to the current head position in the scanning direction.

In this case, the closest requests toward larger values are 82, 140, 170, and 190.

Step 3: Move the head towards the first closest request (82).

Request	**Distance Traveled**
82	32

Step 4: Move the head towards the second closest request (140).

Request	**Distance Traveled**
82	32
140	58

Step 5: Move the head towards the third closest request (170).

Request	**Distance Traveled**
82	32
140	58
170	30

Step 6: Move the head towards the fourth closest request (190).

Request	**Distance Traveled**
82	32
140	58
170	30
190	20

Step 7: All requests have been addressed in the scanning direction. The head moves back to the starting position (track 0) without servicing any requests.

Step 8: Move the head towards the remaining requests (43, 24, and 16) in the same direction.

Request	Distance Traveled
82	32
140	58
170	30
190	20
43	147
24	19
16	8

Step 9: All recommendations have been serviced.

The total number of track movements of the read/write head using LOOK disk scheduling is 32 + 58 + 30 + 20 + 147 + 19 + 8 = 314.

So, the total number of cylinders moved by the head using LOOK disk scheduling is 314.

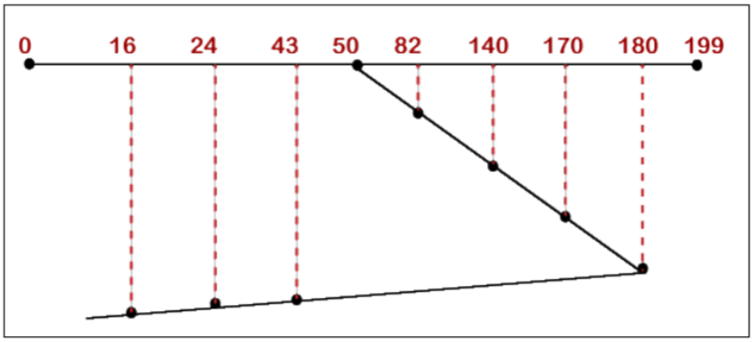


Figure: Look Disk Scheduling

In this case, LOOK offers a disk access pattern that is more effective than SCAN. It reduces the average seek time and enhances overall disk access performance by avoiding needless movement to the end of the disk when there are no more requests in a specific direction. Due to its ability to balance fulfilling demands from both ends of the disk and preventing any one request from becoming starved for resources, LOOK is a well-liked disk scheduling method.

C-LOOK (CIRCULAR LOOK): Compared to LOOK, the disk scheduling algorithm C-LOOK (Circular LOOK) offers better-balanced disk access and decreases the average seek time. Like LOOK, C-LOOK moves the disk head in a single direction while responding to queries. When there are no more requests in the current order, C-LOOK hops back to the opposite end of the disk rather than reversing direction like LOOK does, skipping any demands on the return path. Cheating is prevented by ensuring that queries at one end of the disk do not experience lengthy wait times.

Let's understand C-LOOK with an example:

Think about a disk with 200 tracks (0–199). The tracks with 93, 176, 42, 148, 27, and 14,183 are in the request queue. The R/W head is currently in position 55. The movement is in the direction of the higher value. Utilizing c-look disk scheduling, determine the total number of cylinders moved by the head.

- Disk Tracks: 0 – 199
- Request Queue: 93, 176, 42, 148, 27, 14, 183
- Current Head Position: 55
- Direction: Towards larger values

Step 1: Sort the request queue in ascending order.

Sorted Request Queue: 14, 27, 42, 93, 148, 176, 183

Step 2: Determine the closest requests to the current head position in the scanning direction.

In this case, the closest requests toward larger values are 93, 148, 176, and 183.

Step 3: Move the head towards the first closest request (93).

Request	Distance Traveled
93	38

Step 4: Move the head towards the second closest request (148).

Request	Distance Traveled
93	38
148	55

Step 5: Move the head towards the third closest request (176).

Request	Distance Traveled
93	38
148	55
176	28

Step 6: Move the head towards the fourth closest request (183).

Request	Distance Traveled
93	38
148	55
176	28
183	7

Step 7: All requests have been addressed in the scanning direction. The head moves back to the starting position (track 0) without servicing any requests.

Step 8: Move the head towards the remaining requests (14, 27, 42) in the same direction.

Request	Distance Traveled
93	38
148	55
176	28
183	7
14	169
27	13
42	15

Step 9: All requests have been serviced.

The total number of track movements of the read/write head using C-LOOK disk scheduling is 38 + 55 + 28 + 7 + 169 + 13 + 15 = 325.

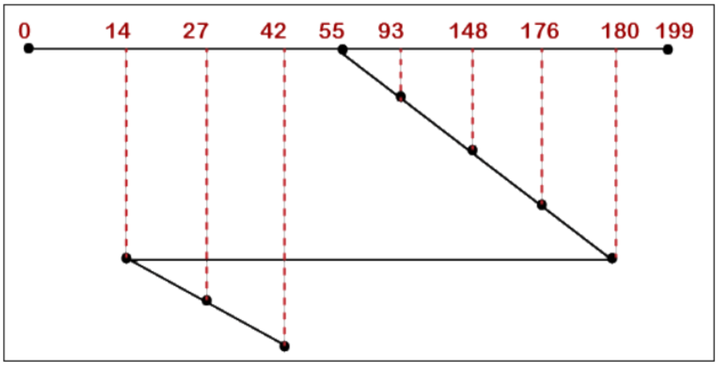


Figure: C-Look Disk Scheduling

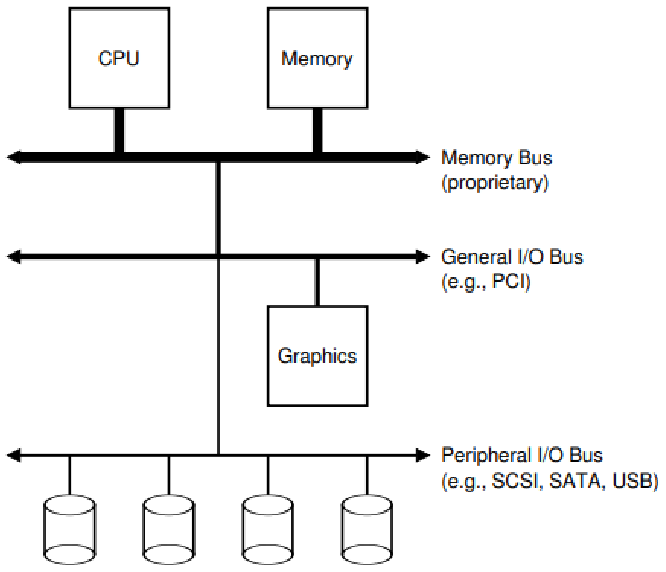
So, the total number of cylinders moved by the head using C-LOOK disk scheduling is 325. Compared to LOOK and other algorithms like SCAN or C-SCAN, C-LOOK offers a more evenly distributed disk access pattern. Installing moving on to the other end after reaching the end of the disk avoids the lengthy wait periods for requests located there. C-LOOK thus decreases the typical seek time and improves overall disk access performance.

CHAPTER-11: DEVICE MANAGEMENT

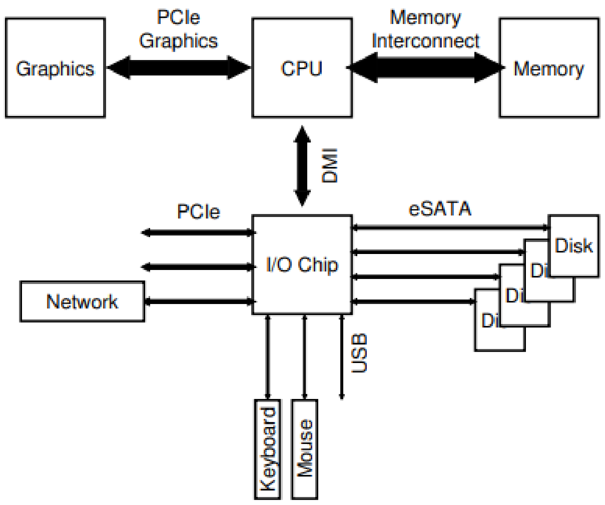
Before delving into the main content of this part of the book (on persistence), we first introduce the concept of an input/output (I/O) device and show how the operating system might interact with such an entity. I/O is quite critical to computer systems, of course; imagine a program without any input (it produces the same result each time); now imagine a program with no output (what was the purpose of it running?). Clearly, for computer systems to be interesting, both input and output are required. And thus, our general problem

SYSTEM ARCHITECTURE

The picture shows a single CPU attached to the main memory of the system via some kind of memory bus or interconnect. Some devices are connected to the system via a general I/O bus, which in many modern systems would be PCI (or one of its many derivatives); graphics and some other higher-performance I/O devices might be found here. Finally, even lower down are one or more of what we call a peripheral bus, such as SCSI, SATA, or USB. These connect slow devices to the system, including disks, mice, and keyboards. One question you might ask is: why do we need a hierarchical structure like this? Put simply: physics, and cost. The faster a bus is, the shorter it must be; thus, a high-performance memory bus does not have much room to plug devices and such into it. In addition, engineering a bus for high performance is quite costly. Thus, system designers have adopted this hierarchical approach, where components that demand high performance (such as the graphics card) are nearer the CPU. Lower performance components are further away.

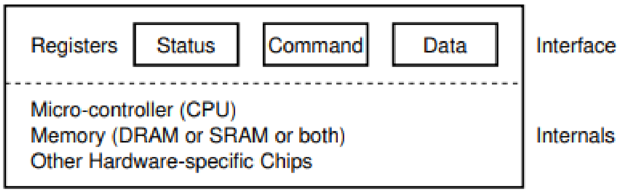


Modern systems increasingly use specialized chipsets and faster point-to-point interconnects to improve performance. Figure 36.2 (page 3) shows an approximate diagram of Intel's Z270 Chipset [H17]. Along the top, the CPU connects most closely to the memory system, but also has a high-performance connection to the graphics card (and thus, the display) to enable gaming (oh, the horror!) and other graphicsintensive applications. The CPU connects to an I/O chip via Intel's proprietary DMI (Direct Media Interface), and the rest of the devices connect to this chip via a number of different interconnects. On the right, one or more hard drives connect to the system via the eSATA interface; ATA (the AT Attachment, in reference to providing connection to the IBM PC AT), then SATA (for Serial ATA), and now eSATA (for external SATA) represent an evolution of storage interfaces over the past decades, with each step forward increasing performance to keep pace with modern storage devices. Below the I/O chip are a number of USB (Universal Serial Bus) connections, which in this depiction enable a keyboard and mouse to be attached to the computer. On many modern systems, USB is used for low performance devices such as these. Finally, on the left, other higher performance devices can be connected.



A Canonical Device

Let us now look at a canonical device (not a real one) and use this device to drive our understanding of some of the machinery required to make device interaction efficient. From Figure 36.3 (page 4), we can see that a device has two important components. The first is the hardware interface it presents to the rest of the system. Just like a piece of software, hardware must also present some kind of interface that allows the system software to control its operation. Thus, all devices have some specified interface and protocol for typical interaction. The second part of any device is its internal structure. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Very simple devices will have one or a few hardware chips to implement their functionality; more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done. For example, modern RAID controllers might consist of hundreds of thousands of lines of firmware (i.e., software within a hardware device) to implement its functionality.



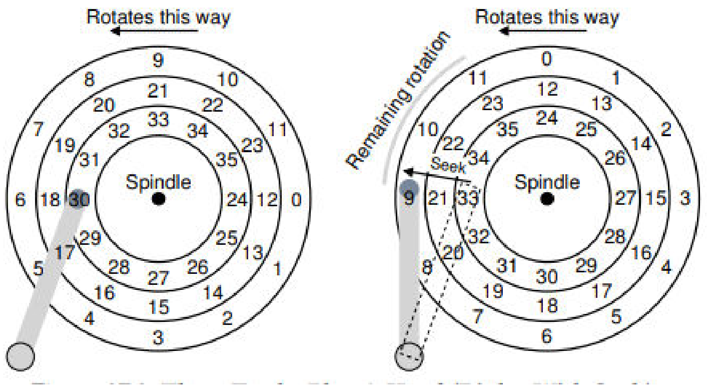
Hard Disk Drives

The last chapter introduced the general concept of an I/O device and showed you how the OS might interact with such a beast. In this chapter, we dive into more detail about one device in particular: the hard disk drive. These drives have been the main form of persistent data storage in computer systems for decades and much of the development of file system technology (coming soon) is predicated on their behavior. Thus, it is worth understanding the details of a disk's operation before building the file system software that manages it.

A Simple Disk Drive

Let's understand how disks work by building up a model one track at a time. Assume we have a simple disk with a single track (Figure 37.1). This track has just 12 sectors, each of which is 512 bytes in size (our typical sector size, recall) and addressed therefore by the numbers 0 through 11. The single platter we have here rotates around the spindle, to which a motor is attached. Of course, the track by itself isn't too interesting; we want to be able to read or write those sectors, and thus we need a disk head, attached to a disk arm, as we now see (Figure 37.2). In the figure, the disk head, attached to the end of the arm, is positioned over sector 6, and the surface is rotating counterclockwise.

Single-track Latency: The Rotational Delay To understand how a request would be processed on our simple, onetrack disk, imagine we now receive a request to read block 0. How should the disk service this request? In our simple disk, the disk doesn't have to do much. It must just wait for the desired sector to rotate under the disk head. This wait happens often enough in modern drives and is an important enough component of I/O service time, that it has a special name: rotational delay (sometimes rotation delay, though that sounds weird). In the example, if the full rotational delay is R, the disk has to incur a rotational delay of about R 2 to wait for 0 to come under the read/write head (if we start at 6). A worst-case request on this single track would be to sector 5, causing nearly a full rotational delay in order to service such a request. Multiple Tracks: Seek Time So far our disk just has a single track, which is not too realistic; modern disks of course have many millions. Let's thus look at an ever-so-slightly more realistic disk surface, this one with three tracks (Figure 37.3, left). In the figure, the head is currently positioned over the innermost track (which contains sectors 24 through 35); the next track over contains the next set of sectors.



I/O Time: Doing the Math

Now that we have an abstract model of the disk, we can use a little analysis to better understand disk performance. In particular, we can now represent I/O time as the sum of three major components:

TI/O = Tseek + Trotation + Ttransfer (37.1) Note that the rate of I/O (RI/O), which is often more easily used forcomparison between drives (as we will do below), is easily computed from the time. Simply divide the size of the transfer by the time it took: RI/O = SizeT ransfer TI/O (37.2) To get a better feel for I/O time, let us perform the following calculation. Assume there are two workloads we are interested in. The first, known as the random workload, issues small (e.g., 4KB) reads to random locations on the disk. Random workloads are common in many important applications, including database management systems. The second, known as the sequential workload, simply reads a large number of sectors consecutively from the disk, without jumping around. Sequential access patterns are quite common and thus important as well. To understand the difference in performance between random and sequential workloads, we need to make a few assumptions about the disk drive first. Let's look at a couple of modern disks from Seagate. The first, known as the Cheetah 15K.5 [S09b], is a high-performance SCSI drive. The second, the Barracuda [S09a], is a drive built for capacity. Details on both are found in Figure 37.5. As you can see, the drives have quite different characteristics, and in many ways nicely summarize two important components of the disk drive market. The first is the "high performance" drive market, where drives are engineered to spin as fast as possible, deliver low seek times, and transfer data quickly.

Disk Scheduling

Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/Os issued to the disk. More specifically, given a set of I/O requests, the disk scheduler examines the requests and decides which one to schedule next [SCO90, JW91]. Unlike job scheduling, where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a "job" (i.e., disk request) will take. By estimating the seek and possible rotational delay of a request, the disk scheduler can know how long each request will take, and thus (greedily) pick the one that will take the least time to service first. Thus, the disk scheduler will try to follow the principle of SJF (shortest job first) in its operation.

SSTF: Shortest Seek Time First One early disk scheduling approach is known as shortest-seek-time-first (SSTF) (also called shortest-seek-first or SSF). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would then issue the request to 21 first, wait for it to complete, and then issue the request to 2 (Figure 37.7). SSTF works well in this example, seeking to the middle track first and then the outer track. However, SSTF is not a panacea, for the following reasons. First, the drive geometry is not available to the host OS; rather, it sees an array of blocks. Fortunately, this problem is rather easily fixed. Instead of SSTF, an OS can simply implement nearest-block-first (NBF), which schedules the request with the nearest block address next.

The second problem is more fundamental: starvation. Imagine in our example above if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach.

Redundant Arrays of Inexpensive Disks

When we use a disk, we sometimes wish it to be faster; I/O operations are slow and thus can be the bottleneck for the entire system. When we use a disk, we sometimes wish it to be larger; more and more data is being put online and thus our disks are getting fuller and fuller. When we use a disk, we sometimes wish for it to be more reliable; when a disk fails, if our data isn't backed up, all that valuable data is gone.

To introduce the Redundant Array of Inexpensive Disks better known as RAID a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system. The term was introduced in the late 1980s by a group of researchers at U.C. Berkeley (led by Professors David Patterson and Randy Katz and then student Garth Gibson); it was around this time that many different researchers simultaneously arrived upon the basic idea of using multiple disks to build a better storage system Externally, a RAID looks like a disk: a group of blocks one can read or write. Internally, the RAID is a complex beast, consisting of multiple disks, memory (both volatile and non-), and one or more processors to manage the system. A hardware RAID is very much like a computer system, specialized for the task of managing a group of disks. RAIDs offer a number of advantages over a single disk. One advantage is performance. Using multiple disks in parallel can greatly speed up I/O times. Another benefit is capacity. Large data sets demand large disks. Finally, RAIDs can improve reliability; spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of redundancy, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

RAIDs provide these advantages transparently to systems that use them, i.e., a RAID just looks like a big disk to the host system. The beauty of transparency, of course, is that it enables one to simply replace a disk with a RAID and not change a single line of software; the operating system and client applications continue to operate without modification. In this manner, transparency greatly improves the deployability of RAID, enabling users and administrators to put a RAID to use without worries of software compatibility. We now discuss some of the important aspects of RAIDs. We begin with the interface, fault model, and then discuss how one can evaluate a RAID design along three important axes: capacity, reliability, and performance. We then discuss a number of other issues that are important to RAID design and implementation.

Interface and Raid Internals

To a file system above, a RAID looks like a big, (hopefully) fast, and (hopefully) reliable disk. Just as with a single disk, it presents itself as a linear array of blocks, each of which can be read or written by the file system (or other client). When a file system issues a logical I/O request to the RAID, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more physical I/Os to do so. The exact nature of these physical I/Os depends on the RAID level, as we will discuss in detail below. However, as a simple example, consider a RAID that keeps two copies of each block (each one on a separate disk); when writing to such a mirrored RAID system, the RAID will have to perform two physical I/Os for every one logical I/O it is issued. A RAID system is often built as a separate hardware box, with a standard connection (e.g., SCSI, or SATA) to a host. Internally, however, RAIDs are fairly complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases, non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations (useful in some RAID levels, as we will also see below). At a high level, a RAID is very much a specialized computer system: it has a processor, memory, and disks; however, instead of running applications, it runs specialized software designed to operate the RAID.

Fault Model

To understand RAID and compare different approaches, we must have a fault model in mind. RAIDs are designed to detect and recover from certain kinds of disk faults; thus, knowing exactly which faults to expect is critical in arriving upon a working design. The first fault model we will assume is quite simple and has been called the fail-stop fault model [S84]. In this model, a disk can be in exactly one of two states: working or failed. With a working disk, all blocks can be read or written. In contrast, when a disk has failed, we assume it is permanently lost. One critical aspect of the fail-stop model is what it assumes about fault detection. Specifically, when a disk has failed, we assume that this is easily detected. For example, in a RAID array, we would assume that the RAID controller hardware (or software) can immediately observe when a disk has failed. Thus, for now, we do not have to worry about more complex “silent” failures such as disk corruption. We also do not have to worry about a single block becoming inaccessible upon an otherwise working disk (sometimes called a latent sector error). We will consider these more complex (and unfortunately, more realistic) disk faults later.

How To Evaluate A RAID

As we will soon see, there are a number of different approaches to building a RAID. Each of these approaches has different characteristics which are worth evaluating, in order to understand their strengths and weaknesses. Specifically, we will evaluate each RAID design along three axes. The first axis is capacity; given a set of N disks each with B blocks, how much useful capacity is available to clients of the RAID? Without redundancy, the answer is N ·B; in contrast, if we have a system that keeps two copies of each block (called mirroring), we obtain a useful capacity of (N · B)/2. Different schemes (e.g., parity-based ones) tend to fall in between. The second axis of evaluation is reliability. How many disk faults can the given design tolerate? In alignment with our fault model, we assume only that an entire disk can fail; in later chapters (i.e., on data integrity), we’ll think about how to handle more complex failure modes.

Finally, the third axis is performance. Performance is somewhat challenging to evaluate, because it depends heavily on the workload presented to the disk array. Thus, before evaluating performance, we will first present a set of typical workloads that one should consider. We now consider three important RAID designs: RAID Level 0 (striping), RAID Level 1 (mirroring), and RAID Levels 4/5 (parity-based redundancy). The naming of each of these designs as a “level” stems from the pioneering work of Patterson, Gibson, and Katz at Berkeley.

Evaluating Raid Performance

In analyzing RAID performance, one can consider two different performance metrics. The first is single-request latency. Understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation. The second is steady-state throughput of the RAID, i.e., the total bandwidth of many concurrent requests. Because RAIDs are often used in high-performance environments, the steady-state bandwidth is critical, and thus will be the main focus of our analyses. To understand throughput in more detail, we need to put forth some workloads of interest. We will assume, for this discussion, that there are two types of workloads: sequential and random. With a sequential workload, we assume that requests to the array come in large contiguous chunks; for example, a request (or series of requests) that accesses 1 MB of data, starting at block x and ending at block (x+1 MB), would be deemed sequential. Sequential workloads are common in many environments (think of searching through a large file for a keyword), and thus are considered important. For random workloads, we assume that each request is rather small, and that each request is to a different random location on disk. For example, a random stream of requests may first access 4KB at logical address 10, then at logical address 550,000, then at 20,100, and so forth. Some important workloads, such as transactional workloads on a database management system (DBMS), exhibit this type of access pattern, and thus it is considered an important workload. Of course, real workloads are not so simple, and often have a mix of sequential and random-seeming components as well as behaviors inbetween the two. For simplicity, we just consider these two possibilities. As you can tell, sequential and random workloads will result in widely different performance characteristics from a disk. With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. With random access, just the opposite is true: most time is spent seeking and waiting for rotation and relatively little time is spent transferring data. To capture this difference in our analysis, we will assume that a disk can transfer data at S MB/s under a sequential workload, and R MB/s when under a random workload. In general, S is much greater than R (i.e., $S \gg R$).

CHAPTER-12: FILE MANAGEMENT

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide:

- Efficient disk space utilization.
- Fast access to the file blocks.

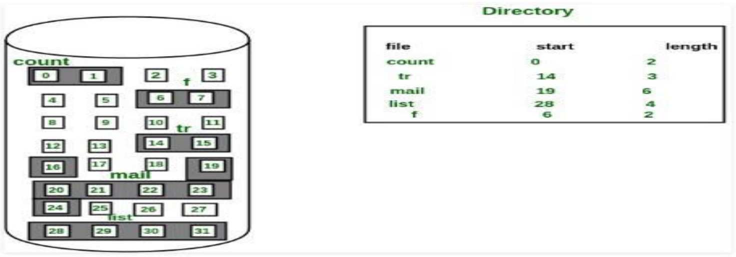
All the three methods have their own advantages and disadvantages as discussed below:

1. Contiguous Allocation

In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.* This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.
The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

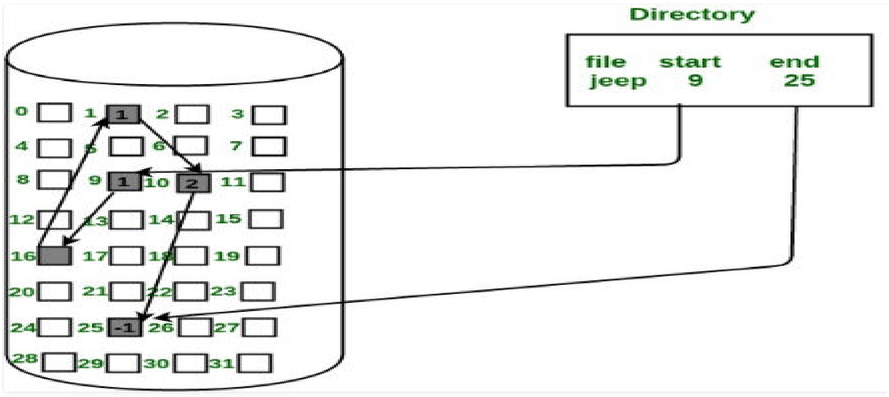
- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Advantages:

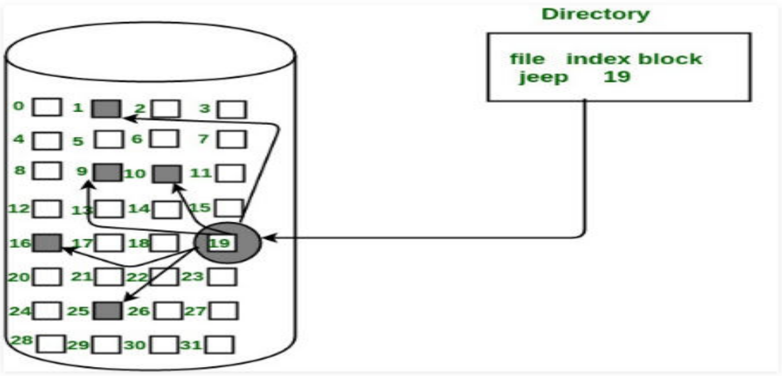
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:



Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

FILE:

A file is a named collection of related information that is recorded on secondary storage.

File attributes: Name, identifier, type, location, size, protection, time date and user identification

INODE:

In Unix based operating system each file is indexed by an **Inode**. Inode are special disk blocks they are created when the file system is created. The number of Inode limits the total number of files/directories that can be stored in the file system.

The Inode contains the following information:

- Administrative information (permissions, timestamps, etc).
- A number of direct blocks (typically 12) that contains to the first 12 blocks of the files.
- A single indirect pointer that points to a disk block which in turn is used as an index block, if the file is too big to be indexed entirely by the direct blocks.
- A double indirect pointer that points to a disk block which is a collection of pointers to disk blocks which are index blocks, used

if the file is too big to be indexed by the direct and single indirect blocks.

- A triple indirect pointer that points to an index block of index blocks of index blocks.

Inode Total Size:

- Number of disk block address possible to store in 1 disk block = (Disk Block Size / Disk Block Address).
- Small files need only the direct blocks, so there is little waste in space or extra disk reads in those cases. Medium sized files may use indirect blocks. Only large files make use of the double or triple indirect blocks, and that is reasonable since those files are large anyway. The disk is now broken into two different types of blocks: **Inode and Data Blocks**.
- There must be some way to determine where the Inodes are, and to keep track of free Inodes and disk blocks. This is done by a **Superblock**. Superblock is located at a fixed position in the file system. The Superblock is usually r

Directory:

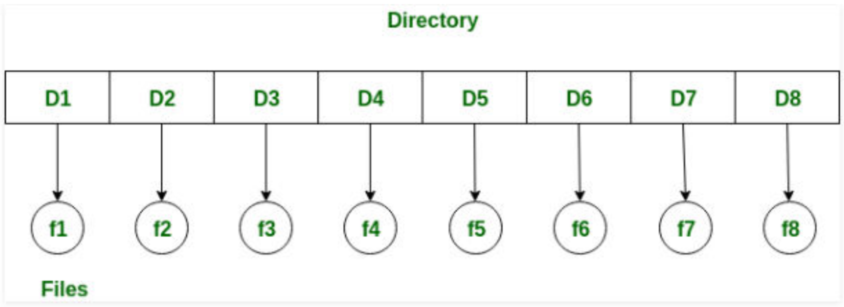
A **directory** is a container that is used to contain folders and file. It organizes files and folders into a hierarchical manner.

There are several logical structures of a directory, these are given below:

1.Single-Level Directory –

Single level directory is simplest directory structure. In it all files are contained in same directory which make it easy to support and understand.

A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. Since all the files are in the same directory, they must have the unique name. If two users call their dataset test, then the unique name rule violated.



Advantages:

- Since it is a single directory, so its implementation is very easy.
- If files are smaller in size, searching will faster.
- The operations like file creation, searching, deletion, updating are very easy in such a directory structure.

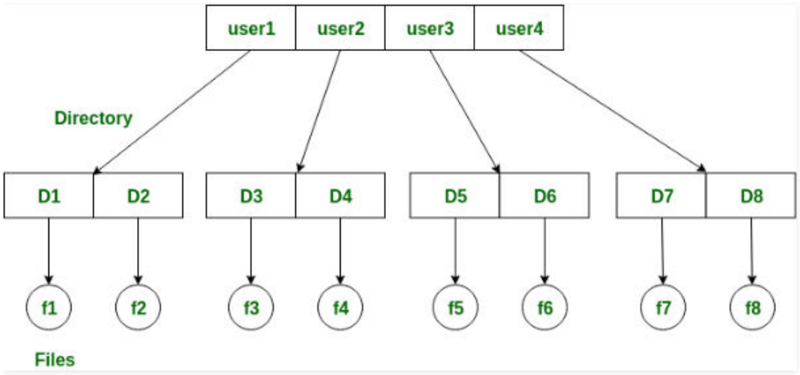
Disadvantages:

- There may chance of name collision because two files can not have the same name.
- Searching will become time taking if directory will large.
- In this can not group the same type of files together.

Two-Level Directory

As we have seen, a single level directory often leads to confusion of files names among different users. the solution to this problem is to create a separate directory for each user.

In the two-level directory structure, each user has there own *user files directory (UFD)*. The UFDs has similar structures, but each lists only the files of a single user. system's *master file directory (MFD)* is searches whenever a new user id=s logged in. The MFD is indexed by username or account number, and each entry points to the UFD for that user.



Advantages:

- We can give full path like /User-name/directory-name/.
- Different users can have same directory as well as file name.
- Searching of files become more easy due to path name and user-grouping.

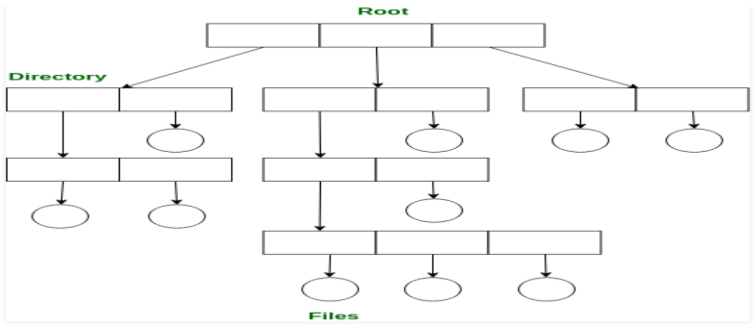
Disadvantages:

- A user is not allowed to share files with other users.
- Still it is not scalable, two files of the same type cannot be grouped together in the same user.

3.Tree-Structured Directory

3. **Tree-structured directory –**

   Once we have seen a two-level directory as a tree of height 2, the natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows the user to create there own subdirectories and to organize on their files accordingly.



A tree structure is the most common directory structure. The tree has a root directory, and every file in the system have a unique path.

Advantages:

- Very generalize, since full path name can be given.
- Very scalable, the probability of name collision is less.
- Searching becomes very easy, we can use both absolute path as well as relative.

Disadvantages:

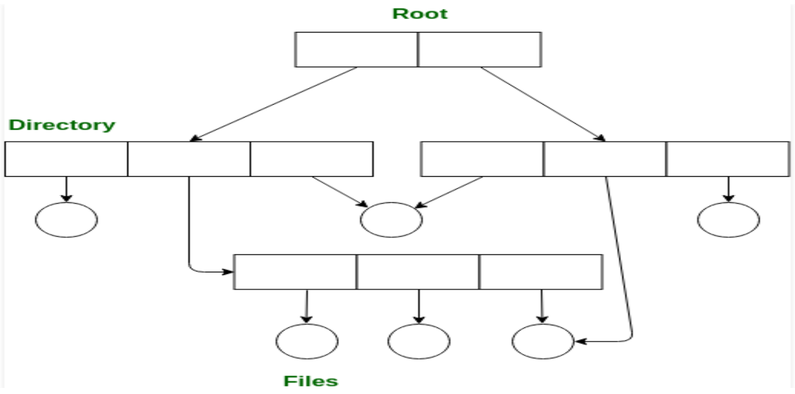
- Every file does not fit into the hierarchical model, files may be saved into multiple directories.
- We can not share files.
- It is inefficient, because accessing a file may go under multiple directories.

4.Acyclicgraph Directory

An acyclic graph is a graph with no cycle and allows to share subdirectories and files. The same file or subdirectories may be in two different directories. It is a natural generalization of the tree-structured directory.

It is used in the situation like when two programmers are working on a joint project and they need to access files. The associated files are stored in a subdirectory, separating them from other projects and files of other programmers, since they are working on a joint project so they want the subdirectories to be into their own directories. The common subdirectories should be shared. So here we use Acyclic directories.

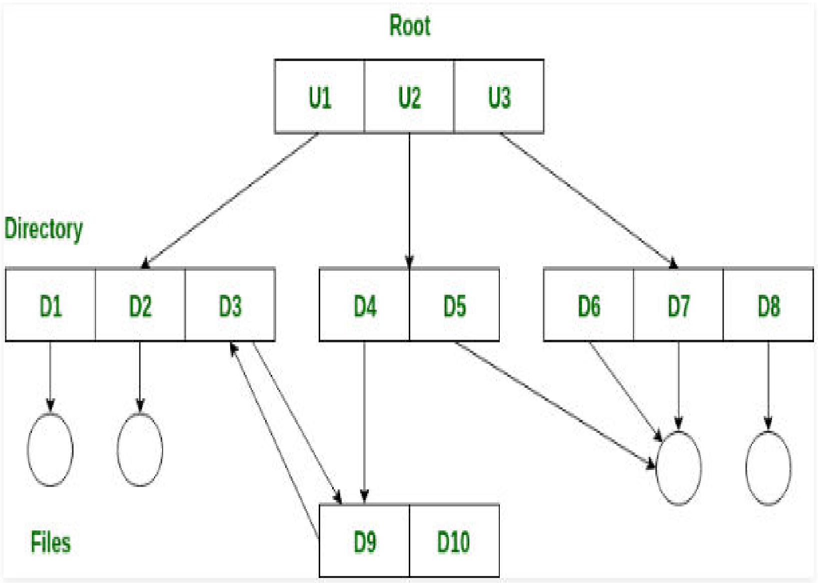
It is the point to note that shared file is not the same as copy file . If any programmer makes some changes in the subdirectory it will reflect in both subdirectories.



5.General graph directory Structure –

In general graph directory structure, cycles are allowed within a directory structure where multiple directories can be derived from more than one parent directory.

The main problem with this kind of directory structure is to calculate total size or space that has been taken by the files and directories.



Problems on disk:

1. Consider a disk pack with the following specifications- 16 surfaces, 128 tracks per surface, 256 sectors per track and 512 bytes per sector. What is the capacity of disk pack?

<u>**Solution-**</u> **Given-**

- Number of surfaces = 16
- Number of tracks per surface = 128
- Number of sectors per track = 256
- Number of bytes per sector = 512 bytes
- Capacity of disk pack
- = Total number of surfaces x Number of tracks per surface x Number of sectors per track x Number of bytes per sector
- = 16 x 128 x 256 x 512 bytes
- = 2^{28} bytes
- = 256 MB

2. A hard disk has 63 sectors per track, 10 platters each with 2 recording surfaces and 1000 cylinders. The address of a sector is given as a triple (c, h, s), where c is the cylinder number, h is the surface number and s is the sector number. Thus, the 0th sector is addressed as (0, 0, 0), the 1st sector as (0, 0, 1), and so on The address <400,16,29> corresponds to sector number:

Solution:

- The data in hard disk is arranged in the shown manner. The smallest division is sector. Sectors are then combined to make a track. Cylinder is formed by combining the tracks which lie on same dimension of the platters.
- Read write head access the disk. Head has to reach at a particular track and then wait for the rotation of the platter so that the required sector comes under it.

Here, each platter has two surfaces, which is the r/w head can access the platter from the two sides, upper and lower. So,<400,16,29> will represent 400 cylinders are passed(0-399) and thus, for each cylinder 20 surfaces (10 platters * 2 surface each) and each cylinder has 63 sectors per surface.

Hence we have passed 0-399 = 400 * 20 * 63 sectors + In 400th cylinder we have passed 16 surfaces(0-15) each of which again contains 63 sectors per cylinder so 16 * 63 sectors. + Now on the 16th surface we are on 29th sector.

So, sector no = 400x20x63 + 16×63 + 29 = 505037.

Disk cache:

A **disk cache** (*cache memory*) is a temporary holding area in the hard disk or random access memory (RAM) where the computer stores information that used repeatedly. The computer can use it to speed up the process of storing and accessing the information much more quickly from the disk cache than if the information stored in the usual place (which might be on a disk or in a part of the computer's memory that takes longer to access). **The term disk cache can also refer to a disk buffer and cache buffer.**

The basic idea behind a *disk cache* is that working with information stored in memory *(RAM)* is much faster than working with information stored on disk. A disk cache is a software *utility* that works by reserving a section of memory where it keeps a copy of information that previously read from your disks. The next time your computer needs that same information, the data can be accessed directly from the cache, bypassing the slower disk. However, if the *CPU (*central processing unit, the chip that runs the computer) needs data that is not in the disk cache, then it has to go to the disk and get it, in which case there is no advantage to having part of your memory set aside for the disk cache.

Disk Failure Modes:

Two types of Common and worthy of failures are **frequency of latent-sector errors (LSEs)** and **block corruption**.

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Frequency of LSEs and Block Corruption

- LSEs arise when a disk sector (or group of sectors) has been damaged in some way. Fortunately, in-disk **error correcting codes** (**ECC**) are used by the drive to determine
- There are also cases where a disk block becomes **corrupt** in a way not detectable by the disk itself.
- These types of faults are particularly insidious because these are **silent faults**; the disk gives no indication of the problem when returning the faulty data.

Some additional findings about LSEs are:

- Costly drives with more than one LSE are as likely to develop additional.
- For most drives, annual error rate increases in year two
- LSEs increase with disk size
- Most disks with LSEs have less than 50
- Disks with LSEs are more likely to develop additional LSEs
- There exists a significant amount of spatial and temporal locality
- Disk scrubbing is useful (most LSEs were found this way)

- **Some additional findings about Block corruption:**
  - Chance of corruption varies greatly across different drive models
  - Within the same drive class
  - Age affects are different across models
  - Workload and disk size have little impact on corruption
  - Most disks with corruption only have a few corruptions

- Corruption is not independent with a disk or across disks in RAID
- There exists spatial locality, and some temporal locality
- There is a weak correlation with LSEs

Handling Latent Sector Errors:

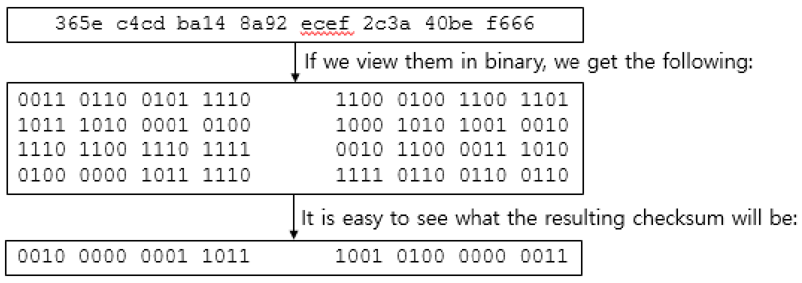
- Latent sector errors are easily detected and handled.
- Using **redundancy mechanisms**:
- In a mirrored RAID or RAID-4 and RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group.

Detecting Corruption: The Checksum

- How can a client tell that a block has gone bad?
- Using **Checksum mechanisms:**
- This is simple the result of a function that takes a chunk of data as input and computes a function over said data, producing a small summary of the contents of the data.

Common Checksum Functions:

- Different functions are used to compute checksums and vary in strength.
- One simple checksum function that some use is based on exclusive or(XOR).



```
365e c4cd ba14 8a92 ecef 2c3a 40be f666
```

If we view them in binary, we get the following:

```
0011 0110 0101 1110     1100 0100 1100 1101
1011 1010 0001 0100     1000 1010 1001 0010
1110 1100 1110 1111     0010 1100 0011 1010
0100 0000 1011 1110     1111 0110 0110 0110
```

It is easy to see what the resulting checksum will be:

```
0010 0000 0001 1011     1001 0100 0000 0011
```

The result, in hex, is 0x201b9403.

- XOR is a reasonable checksum but has its limitations.
- Two bits in the same position within each check sumed unit changed the checksum will not detect the corruption.

Addition Checksum

- This approach has the advantage of being fast.
- Compute 2's complement addition over each chunk of the data
- ignoring overflow

Fletcher Checksum

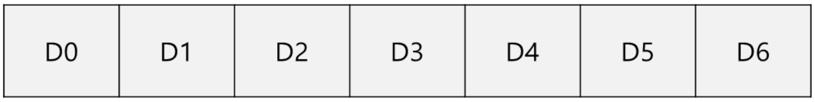
- Compute two check bytes, s1 and s2.
- Assuming a block D consists of bytes d1…dn; s1 is simply in turn is

s1 = s1 + di mod 255(compute over all di);

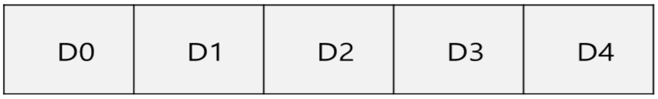
- s2 = s2 + s1 mod 255(again over all di);
- Cyclic redundancy check(CRC)
- Treating D as if it is a large binary number and divide it by an agreed upon value.
- The remainder of this division is the value of the CRC.

Checksum Layout:

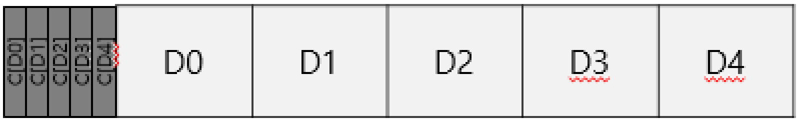
- The disk layout without checksum:



- The disk layout **with checksum:**



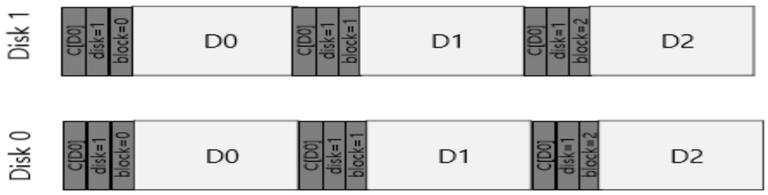
- Store the checksums packed into 512-byte blocks.



Using Checksums:

- When reading a block D, the client reads its checksum from disk Cs(D), **stored checksum**
- Computes the checksum over the retrieved block D, **computed checksum** Cc(D).

- Compares the stored and computed checksums;
  - If they are equal (Cs(D) == Cc(D)), the data is in safe.
  - If they do not match (Cs(D) != Cc(D)), the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time).
- A New Problem: Misdirected Writes
- Modern disks have a couple of unusual failure-modes that require different solutions.
- Misdirected write arises in disk and RAID controllers which the data to disk correctly, except in the *wrong* location



One Last Problem: Lost Writes

- Lost Writes, occurs when the device informs the upper layer that a write has completed but in fact it never is persisted. Scrubbing:
- When do these checksums actually get checked?
- Most data is rarely accessed, and thus remain unchecked.
- To remedy this problem, many systems utilize **disk scrubbing**.
- By **periodically reading** through every block of the system
- Checking whether checksum are still valid
- Reduce the chances that all copies of certain data become corrupted

Overhead of Check summing:

- Two distinct kinds of overheads : **space** and **time**
- Space overheads

Disk itself: A typical ratio might be an 8byte checksum per 4KB data block, for a 0.19% on-disk space overhead.

Memory of the system: This overhead is short-lived and not much of a concern. Time overheads CPU must compute the checksum over each block

To reducing CPU overheads is to combine data copying and check summing into one streamlined activity.

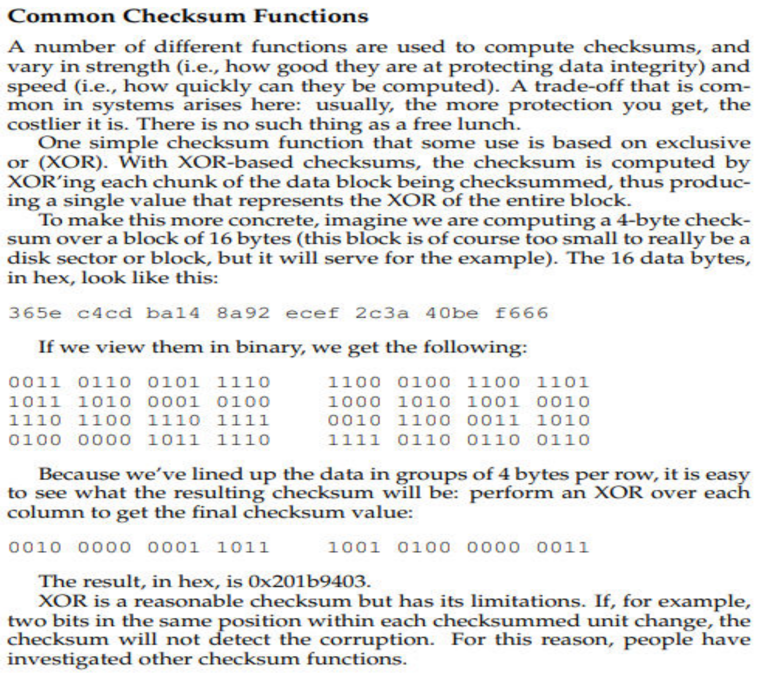
CHAPTER-13: DATA INTEGRITY AND PROTECTION

Disk Failure Modes As you learned in the chapter about RAID, disks are not perfect, and can fail (on occasion). In early RAID systems, the model of failure was quite simple: either the entire disk is working, or it fails completely, and the detection of such a failure is straightforward. This fail-stop model of disk failure makes building RAID relatively simple . What you didn't learn is about all of the other types of failure modes modern disks exhibit. Specifically, as Bairavasundaram et al. studied in great detail modern disks will occasionally seem to be mostly working but have trouble successfully accessing one or more blocks. Specifically, two types of single-block failures are common and worthy of consideration: latent-sector errors (LSEs) and block corruption. We'll now discuss each in more detail.

LSEs arise when a disk sector (or group of sectors) has been damaged in some way. For example, if the disk head touches the surface for some reason (a head crash, something which shouldn't happen during normal operation), it may damage the surface, making the bits unreadable. Cosmic rays can also flip bits, leading to incorrect contents. Fortunately, in-disk error correcting codes (ECC) are used by the drive to determine whether the on-disk bits in a block are good, and in some cases, to fix them; if they are not good, and the drive does not have enough information to fix the error, the disk will return an error when a request is issued to read them. There are also cases where a disk block becomes corrupt in a way not detectable by the disk itself. For example, buggy disk firmware may write a block to the wrong location; in such a case, the disk ECC indicates the block contents are fine, but from the client's perspective the wrong block is returned when subsequently accessed. Similarly, a block may get corrupted when it is transferred from the host to the disk across a faulty bus; the resulting corrupt data is stored by the disk, but it is not what the client desires. These types of faults are particularly insidious because they are silent faults; the disk gives no indication of the problem when returning the faulty data. Prabhakaran et al. describes this more modern view of disk failure as the fail-partial disk failure model . In this view, disks can still fail in their entirety (as was the case in the traditional fail-stop model); however, disks can also seemingly be working and have one or more blocks become inaccessible (i.e., LSEs) or hold the wrong contents (i.e., corruption). Thus, when accessing a seemingly-working disk, once in a while it may either return an error when trying to read or write a given block (a non-silent partial fault), and once in a while it may simply return the wrong data (a silent partial fault). Both of these types of faults are somewhat rare, but just how rare? Figure 45.1 summarizes some of the findings from the two Bairavasundaram studies. The figure shows the percent of drives that exhibited at least one LSE or block corruption over the course of the study (about 3 years, over 1.5 million disk drives). The figure further sub-divides the results into "cheap" drives (usually SATA drives) and "costly" drives (usually SCSI or Fibre Channel). As you can see, while buying better drives reduces the frequency of both types of problem (by about an order of magnitude), they still happen often enough that you need to think carefully about how to handle them in your storage system.

Handling Latent Sector Errors Given these two new modes of partial disk failure, we should now try to see what we can do about them. Let's first tackle the easier of the two, namely latent sector errors. CRUX: HOW TO HANDLE LATENT SECTOR ERRORS How should a storage system handle latent sector errors? How much extra machinery is needed to handle this form of partial failure? As it turns out, latent sector errors are rather straightforward to handle, as they are (by definition) easily detected. When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data. In a mirrored RAID, for example, the system should access the alternate copy; in a RAID-4 or RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group. Thus, easily detected problems such as LSEs are readily recovered through standard redundancy mechanisms.

Detecting Corruption: The Checksum Let's now tackle the more challenging problem, that of silent failures via data corruption. How can we prevent users from getting bad data when corruption arises, and thus leads to disks returning bad data? CRUX: HOW TO PRESERVE DATA INTEGRITY DESPITE CORRUPTION Given the silent nature of such failures, what can a storage system do to detect when corruption arises? What techniques are needed? How can one implement them efficiently? Unlike latent sector errors, detection of corruption is a key problem. How can a client tell that a block has gone bad? Once it is known that a particular block is bad, recovery is the same as before: you need to have some other copy of the block around (and hopefully, one that is not corrupt!). Thus, we focus here on detection techniques. The primary mechanism used by modern storage systems to preserve data integrity is called the checksum. A checksum is simply the result of a function that takes a chunk of data (say a 4KB block) as input and computes a function over said data, producing a small summary of the contents of the data (say 4 or 8 bytes). This summary is referred to as the checksum. The goal of such a computation is to enable a system to detect if data has somehow been corrupted or altered by storing the checksum with the data and then confirming upon later access that the data's current checksum matches the original storage value.



Common Checksum Functions

A number of different functions are used to compute checksums, and vary in strength (i.e., how good they are at protecting data integrity) and speed (i.e., how quickly can they be computed). A trade-off that is common in systems arises here: usually, the more protection you get, the costlier it is. There is no such thing as a free lunch.

One simple checksum function that some use is based on exclusive or (XOR). With XOR-based checksums, the checksum is computed by XOR'ing each chunk of the data block being checksummed, thus producing a single value that represents the XOR of the entire block.

To make this more concrete, imagine we are computing a 4-byte checksum over a block of 16 bytes (this block is of course too small to really be a disk sector or block, but it will serve for the example). The 16 data bytes, in hex, look like this:

```
365e c4cd ba14 8a92 ecef 2c3a 40be f666
```

If we view them in binary, we get the following:

```
0011 0110 0101 1110     1100 0100 1100 1101
1011 1010 0001 0100     1000 1010 1001 0010
1110 1100 1110 1111     0010 1100 0011 1010
0100 0000 1011 1110     1111 0110 0110 0110
```

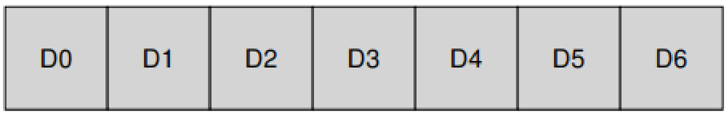
Because we've lined up the data in groups of 4 bytes per row, it is easy to see what the resulting checksum will be: perform an XOR over each column to get the final checksum value:

```
0010 0000 0001 1011     1001 0100 0000 0011
```

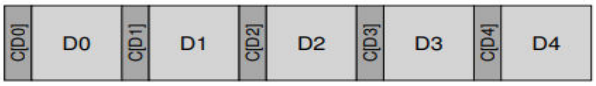
The result, in hex, is 0x201b9403.

XOR is a reasonable checksum but has its limitations. If, for example, two bits in the same position within each checksummed unit change, the checksum will not detect the corruption. For this reason, people have investigated other checksum functions.

Checksum Layout Now that you understand a bit about how to compute a checksum, let's next analyze how to use checksums in a storage system. The first question we must address is the layout of the checksum, i.e., how should checksums be stored on disk? The most basic approach simply stores a checksum with each disk sector (or block). Given a data block D, let us call the checksum over that data C(D). Thus, without checksums, the disk layout looks like this:

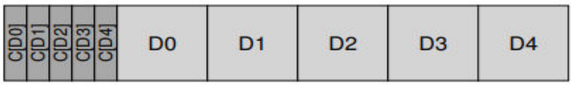


With checksums, the layout adds a single checksum for every block:



Because checksums are usually small (e.g., 8 bytes), and disks only can write in sector-sized chunks (512 bytes) or multiples thereof, one problem that arises is how to achieve the above layout. One solution employed by drive manufacturers is to format the drive with 520-byte sectors; an extra 8 bytes per sector can be used to store the checksum.

In disks that don't have such functionality, the file system must figure out a way to store the checksums packed into 512-byte blocks. One such possibility is as follows:



In this scheme, the n checksums are stored together in a sector, followed by n data blocks, followed by another checksum sector for the next n blocks, and so forth. This approach has the benefit of working on all disks, but can be less efficient; if the file system, for example, wants to overwrite block $D1$, it has to read in the checksum sector containing $C(D1)$, update $C(D1)$ in it, and then write out the checksum sector and new data block $D1$ (thus, one read and two writes). The earlier approach (of one checksum per sector) just performs a single write.

Using Checksums with a checksum layout decided upon, we can now proceed to actually understand how to use the checksums. When reading a block D, the client (i.e., file system or storage controller) also reads its checksum from disk Cs(D), which we call the stored checksum (hence the subscript Cs). The client then computes the checksum over the retrieved block D, which we call the computed checksum Cc(D). At this point, the client compares the stored and computed checksums; if they are equal (i.e., Cs(D) == Cc(D), the data has likely not been corrupted, and thus can be safely returned to the user. If they do not match (i.e., Cs(D) != Cc(D)), this implies the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time). In this case, we have a corruption, which our checksum has helped us to detect.

One Last Problem: Lost Writes Unfortunately, misdirected writes are not the last problem we will address. Specifically, some modern storage devices also have an issue known as a lost write, which occurs when the device informs the upper layer that a write has completed but in fact it never is persisted; thus, what remains is the old contents of the block rather than the updated new contents. The obvious question here is: do any of our check summing strategies from above (e.g., basic checksums, or physical identity) help to detect lost writes? Unfortunately, the answer is no: the old block likely has a matching checksum, and the physical ID used above (disk number and block offset) will also be correct

Overheads Of Check summing Before closing, we now discuss some of the overheads of using checksums for data protection. There are two distinct kinds of overheads, as is common in computer systems: space and time. Space overheads come in two forms. The first is on the disk (or other storage medium) itself; each stored checksum takes up room on the disk, which can no longer be used for user data. A typical ratio might be an 8- byte checksum per 4 KB data block, for a 0.19% on-disk space overhead. The second type of space overhead comes in the memory of the system. When accessing data, there must now be room in memory for the checksums as well as the data itself. However, if the system simply checks the checksum and then discards it once done, this overhead is short-lived and not much of a concern. Only if checksums are kept in memory (for an added level of protection against memory corruption [Z+13]) will this small overhead be observable.

While space overheads are small, the time overheads induced by check summing can be quite noticeable. Minimally, the CPU must compute the checksum over each block, both when the data is stored (to determine the value of the stored checksum) and when it is accessed (to compute the checksum again and compare it against the stored checksum). One approach to reducing CPU overheads, employed by many systems that use checksums (including network stacks), is to combine data copying and check summing into one streamlined activity; because the copy is needed anyhow (e.g., to copy the data from the kernel page cache into a user buffer), combined copying/check summing can be quite effective.

CHAPTER-14: REAL TIME OPERATING SYSTEMS

Advanced Operating Systems (OS) refer to a category of operating systems that offer more sophisticated features and capabilities compared to traditional or basic operating systems. These advanced features are designed to meet the specific demands of modern computing environments, including distributed systems, real-time systems, and high-performance computing. Some of the common types of advanced operating systems include:

Real-Time Operating Systems (RTOS): RTOS is designed to handle real-time applications where timely response is critical. They are used in embedded systems, control systems, robotics, and other time-sensitive applications. RTOS ensures that tasks are executed within specific time constraints, minimizing delays and providing predictable and deterministic behavior.

A distinct feature of real-time systems is that jobs have completion deadlines. A job should be completed before its deadline to be of use (in soft real-time systems) or to avoid a disaster (in hard real-time systems).

- **Hard Real Time:** In Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration.

  Example: Medical critical care system, Aircraft systems, etc
- **Firm Real time:** These type of RTOS also need to follow the deadlines. However, missing a deadline may not have big impact but could cause undesired affects, like a huge reduction in quality of a product.

  Example: Various types of Multimedia applications
- **Soft Real Time:** Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of

time is acceptable. So, deadlines are handled softly by this type of RTOS.

Example: Online Transaction system and Livestock price quotation System.

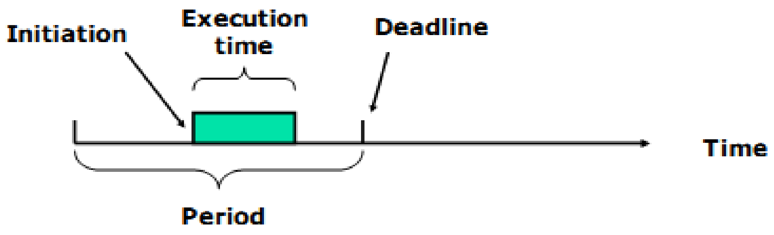


Figure1.3. Real Time Operating Systems

Examples: of RTOS include Free RTOS, VxWorks, and QNX.

REAL-TIME SCHEDULING

RTS stands for **"Real-Time Scheduling."** RTS is a scheduling technique used to manage tasks and processes in real-time systems. Real-time systems have strict timing requirements, where tasks must be completed within specific deadlines to ensure correct system operation. In particular, these tasks are related to control of certain events (or) reacting to them. Real-time tasks can be classified as hard real-time tasks and soft real-time tasks.

There are two main types of real-time scheduling:

1. **Hard Real-Time Scheduling:** In hard real-time systems, missing a deadline can lead to catastrophic consequences. Tasks have absolute deadlines that must be met under all circumstances. Meeting deadlines is the highest priority, even if it means sacrificing other tasks or processes.

There are different approaches to hard real-time scheduling, such as static, dynamic, priority-based, or planning-based. Static scheduling assigns fixed priorities to tasks at compile time and uses a preemptive scheduler to run the highest priority task at any given time. Dynamic scheduling determines the priorities of tasks at run time based on factors such as deadlines, resource availability, or workload. Priority-based scheduling assigns priorities to tasks according to some criteria, such as rate-monotonic (shorter period implies higher priority) or deadline-monotonic (earlier deadline implies higher priority). Planning-based scheduling uses a fixed time interval and executes a task only if it satisfies the time constraint.

2. **Soft Real-Time Scheduling:** Soft real-time systems have deadlines that are not absolute but are more like preferences. While meeting deadlines is essential, occasional deadline misses might be tolerated as long as they don't jeopardize the overall system stability.

There are different approaches to soft real-time scheduling, such as static, dynamic, priority-based, or best-effort. Static scheduling assigns fixed priorities to tasks at compile time and uses a pre-emptive scheduler to run the highest priority task at any given time. Dynamic scheduling determines the priorities of tasks at run time based on factors such as deadlines, resource availability, or workload. Priority-based scheduling assigns priorities to tasks according to some criteria, such as earliest deadline first (EDF) or least laxity first (LLF). Best-effort scheduling tries to meet the deadlines of tasks as much as possible, but does not guarantee their completion.

Soft real-time scheduling is a practical and flexible technique for handling various types of applications that have diverse and dynamic requirements.

Some of the benefits of soft real-time scheduling are:

- **Improved resource utilization:** Soft real-time scheduling can allow the system to use the available resources more efficiently and effectively, by adapting to the changing conditions and demands of the tasks.
- **Reduced over-provisioning:** Soft real-time scheduling can avoid the need for over-provisioning the system with excessive resources, by tolerating some deadline misses and providing trade-offs between quality and timeliness.

- **Enhanced user satisfaction:** Soft real-time scheduling can improve the user satisfaction and experience, by providing acceptable levels of quality and responsiveness for the tasks.

For example, a soft real-time system could be a video streaming service, a voice recognition system, or a multimedia application.

In real-time systems, the scheduler is considered as the most important component which is typically a short-term task scheduler. The main focus of this scheduler is to reduce the response time associated with each of the associated processes instead of handling the deadline.

If a pre-emptive scheduler is used, the real-time task needs to wait until its corresponding tasks time slice completes. In the case of a non-pre-emptive scheduler, even if the highest priority is allocated to the task, it needs to wait until the completion of the current task. This task can be slow (or) of the lower priority and can lead to a longer wait.

A better approach is designed by combining both pre-emptive and non-pre-emptive scheduling. This can be done by introducing time-based interrupts in priority based systems which means the currently running process is interrupted on a time-based interval and if a higher priority process is present in a ready queue, it is executed by pre-empting the current process.

The choice of a specific real-time scheduling algorithm depends on the requirements and characteristics of the real-time system, the importance of meeting deadlines, and the nature of the tasks being executed.

Real-time scheduling is crucial in various domains, including industrial automation, aerospace, medical devices, automotive systems, and multimedia applications, where timing precision and predictability are of utmost importance.

There are various real-time scheduling algorithms to manage tasks in real-time systems, some of which include:

1. **Rate-Monotonic Scheduling (RMS):** A priority-based scheduling algorithm where tasks with shorter periods have higher priority.

2. **Earliest Deadline First (EDF):** A priority-based algorithm where tasks with the closest deadlines have higher priority.
3. **Fixed-Priority Scheduling:** Tasks are assigned fixed priorities based on their characteristics and requirements.
4. **Dynamic-Priority Scheduling:** Priorities of tasks are adjusted dynamically based on their behavior and deadlines.

Advantages of Scheduling in Real-Time Systems:

- **Meeting Timing Constraints:** Scheduling ensures that real-time tasks are executed within their specified timing constraints. It guarantees that critical tasks are completed on time, preventing potential system failures or losses.
- **Resource Optimization:** Scheduling algorithms allocate system resources effectively, ensuring efficient utilization of processor time, memory, and other resources. This helps maximize system throughput and performance.
- **Priority-Based Execution:** Scheduling allows for priority-based execution, where higher-priority tasks are given precedence over lower-priority tasks. This ensures that time-critical tasks are promptly executed, leading to improved system responsiveness and reliability.
- **Predictability and Determinism:** Real-time scheduling provides predictability and determinism in task execution. It enables developers to analyze and guarantee the worst-case execution time and response time of tasks, ensuring that critical deadlines are met.
- **Control Over Task Execution:** Scheduling algorithms allow developers to have fine-grained control over how tasks are executed, such as specifying task priorities, deadlines, and inter-task dependencies. This control facilitates the design and implementation of complex real-time systems.

Disadvantages of Scheduling in Real-Time Systems:

- **Increased Complexity:** Real-time scheduling introduces additional complexity to system design and implementation.

Developers need to carefully analyze task requirements, define priorities, and select suitable scheduling algorithms. This complexity can lead to increased development time and effort.

- **Overhead:** Scheduling introduces some overhead in terms of context switching, task prioritization, and scheduling decisions. This overhead can impact system performance, especially in cases where frequent context switches or complex scheduling algorithms are employed.
- **Limited Resources:** Real-time systems often operate under resource-constrained environments. Scheduling tasks within these limitations can be challenging, as the available resources may not be sufficient to meet all timing constraints or execute all tasks simultaneously.
- **Verification and Validation:** Validating the correctness of real-time schedules and ensuring that all tasks meet their deadlines require rigorous testing and verification techniques. Verifying timing constraints and guaranteeing the absence of timing errors can be a complex and time-consuming process.
- **Scalability:** Scheduling algorithms that work well for smaller systems may not scale effectively to larger, more complex real-time systems. As the number of tasks and system complexity increases, scheduling decisions become more challenging and may require more advanced algorithms or approaches.

Reference:

1. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, Arpaci-Dusseau Books, May, (2014).

CHAPTER CONTRIBUTORS

Dr.V.Naresh
Dr.P.V.R.D.Prasad
M.Srikanth
Dr.Veeramallu
Keerthi Samhitha
B.Pravalika
Dr.B.Vijay Kumar
S.N.V.Jyotshna Kosuru
Y.Bhagya Laxmi

(DEEMED TO BE UNIVERSITY)

KONERU LAKSHMAIAH EDUCATION FOUNDATION,
GREEN FIELDS, VADDESWARAM,
GUNTUR-522502

www.kluniversity.in