Nilesh Maltare
Mahesh Goyani
Safvan Vahora

Principles of Operating System Design and Virtualization Technologies

DeepScience

Principles of Operating System Design and Virtualization Technologies

Nilesh Maltare

Department of Information Technology, Government Engineering College, Modasa, Gujarat, India

Mahesh Goyani

Department of Computer Engineering, Government Engineering College, Modasa, Gujarat, India

Safvan Vahora

Department of Information Technology, Government Engineering College, Modasa, Gujarat, India

DeepScience

Published, marketed, and distributed by:

Deep Science Publishing
USA | UK | India | Turkey
Reg. No. MH-33-0523625
www.deepscienceresearch.com
editor@deepscienceresearch.com
WhatsApp: +91 7977171947

ISBN: 978-93-49307-37-7

E-ISBN: 978-93-49307-65-0

https://doi.org/10.70593/978-93-49307-65-0

Citation: Maltare, N., Goyani, M., & Vahora, S. (2025). *Principles of Operating System Design and Virtualization Technologies*. Deep Science Publishing. https://doi.org/10.70593/978-93-49307-65-0

Contents

PREFACE

Welcome to "Basics of Operating Systems and Virtualization." This book aims to provide a comprehensive introduction to the fundamental concepts of operating systems and virtualization.

To facilitate effective learning, this book employs a variety of pedagogical approaches:

- Analogy: Drawing parallels between complex concepts and everyday experiences to enhance understanding.
- Incremental Learning: Building knowledge step-by-step, ensuring a solid foundation before progressing to more advanced topics.
- Visualization: Utilizing diagrams and visual aids to clarify complex processes and systems.
- Practical Examples and Case Studies: Integrating real-world scenarios to illustrate theoretical concepts.
- Exercises: Providing hands-on exercises to reinforce learning and enable practical application of concepts.

Book Structure

This book is meticulously structured to ensure a logical progression of topics. It begins with the fundamental principles of operating systems and gradually advances to the intricacies of virtualization. Each chapter combines theoretical explanations with practical examples and exercises to reinforce learning.

- Chapter 1: Introduction to Operating Systems: Discusses the services provided by operating systems and the various types available.
- Chapter 2: Process Management: Introduces concepts related to process management, including process life cycle and scheduling.
- Chapter 3: CPU Scheduling: Explains different CPU scheduling algorithms and their applications.
- Chapter 4: Inter-Process Communication: Covers mechanisms for communication between processes, such as message passing and shared memory.
- Chapter 5: Deadlock: Addresses deadlock scenarios and strategies for prevention, avoidance, and detection.
- Chapter 6: Memory Management: Discusses various techniques for managing memory, including partitioning, paging, and segmentation.

- Chapter 7: Virtual Memory: Explores virtual memory concepts, including paging and page replacement algorithms.
- Chapter 8: Disk Scheduling: Examines algorithms for efficient disk scheduling.
- Chapter 9: File Management: Covers file system structures, file allocation methods, and directory systems.
- Chapter 10: I/O Management: Discusses I/O system architecture and strategies for managing input/output operations.
- Chapter 11: Security: Presents fundamental security mechanisms to protect operating systems from threats.
- Chapter 12: Virtualization: Explores virtualization principles, hypervisors, virtual machines, and containerization.
- Chapter 13: Linux Operating System: Delves into the Linux operating system, its architecture, and unique features.

We invite educators, students, and professionals to contribute to this book. Your feedback, suggestions, and contributions are invaluable in making this a continually improving resource for learners worldwide.

We hope that "Basics of Operating Systems and Virtualization" will serve as a vital resource in your educational journey and help you develop a strong foundation in these essential areas of computer science.

Enjoy your exploration of operating systems and virtualization!

Nilesh N. Maltare
Assistant Professor
Department of Information Technology
Government Engineering College, Modasa, Gujarat, India
Mahesh M. Goyani
Assistant Professor
Department of Computer Engineering
Government Engineering College, Modasa, Gujarat, India
Safvan Vahora
Assistant Professor
Department of Information Technology
Government Engineering College, Modasa, Gujarat, India

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

DeepScience
Open Access Books

Chapter 1

Introduction

Abstract: This book chapter delves into the fundamental concepts of operating systems, exploring their essential role in managing computer hardware and software resources. It covers the architecture and functions of operating systems, including process management, memory management, file systems, and input/output control. The chapter also explores different types of operating systems, including batch operating systems, time-sharing systems, and real-time systems, highlighting their distinctive characteristics and uses. Through detailed explanations and practical examples, the chapter aims to provide a comprehensive understanding of how operating systems function and their importance in the computing world.

Keywords: Operating Systems, Process Management Memory Management File Systems Input/Output Control Batch Systems Time-Sharing Systems Real-Time Systems Computer Architecture

Computers, laptops and smartphones are now a days common in everybody's life. One important part of this devices which is not visible is Operating system. An Operating System, or "OS," is software that communicates with the hardware and allows other programs to run(Silberschatz et al., 2012).

An operating system is a software that offers a user-friendly interface and efficiently manages computer hardware.

Operating system is essential for every general-purpose computer to run other programs. The operating system handles fundamental tasks such as taking input from the keyboard, displaying output on the screen, managing files and directories on the disk, and controlling peripheral devices like disk drives and printers.

Operating system provides a base on which other application programs can run. For large systems, the operating system has even greater responsibilities and powers. The Operating System is also responsible for security, ensuring that unauthorized users do not access the system.

Every desktop computer, tablet, and smart phone includes an Operating System that provides basic functionality for the device. Popular desktop operating systems include Windows, Mac OS X, and Linux. Each operating system has its own unique features but all provides a graphical user interface (GUI) that includes a desktop and tools for managing files and folders. Additionally, they enable the installation and execution of programs designed specifically for the operating system. While Windows and Linux can be installed on

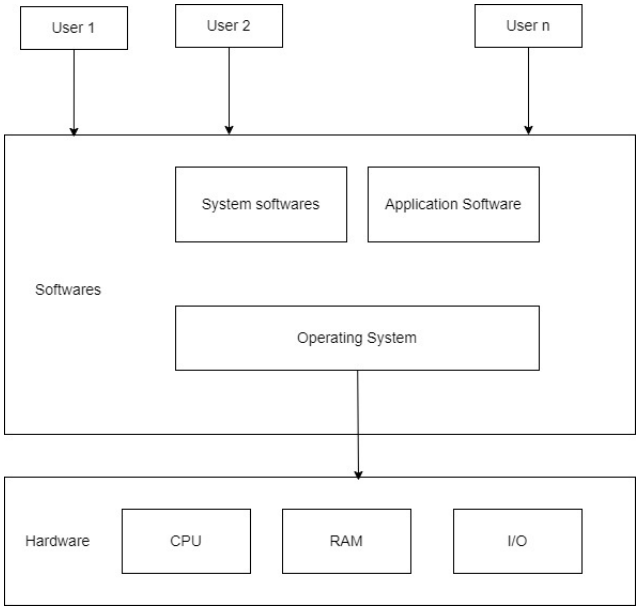


Figure 1.1: Operating System

standard PC hardware, Mac OS X can only run on Macintosh computers. Therefore, the hardware you choose affects what operating system(s) you can run. Mobile devices, such as tablets and smartphones also include operating systems that provide a GUI and can run applications. Common mobile OS include Android, iOS, and Windows Phone. These OS are developed specifically for portable devices and therefore are designed around touchscreen input.

OS is like Brain, Brain controls all parts of the body:

1. What to think? (Use of Mind)
2. What to remember?
3. What to see,hear, smell and feel?
4. What to speak,write? ...

In similar way OS controls computer H/W.

1. Use of Processor to Processing information
2. Use of Storage for storing Information (Storage RAM/ROM/HDD)
3. Use of Input devices for Taking Input from user (Input Devices)
4. Displaying or giving output to user (Output Devices) ...

1.1 History of Operating System

OS's are changing since the birth of computers. We can observe OS changes with change in hardware as presented in table 1.1.

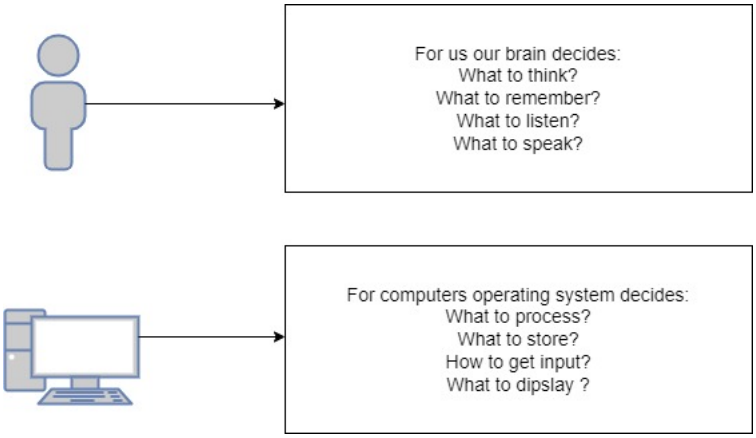


Figure 1.2: Operating System Analogy

Table 1.1: Operating System Development

OS Class	Period	Objectives	Hardware
Bare Machine	First Generation	Program Execution	Vacuum Tubes and Plug-boards
Batch Processing	Second Generation	Improve CPU Utilization	Transistors
Multiprograming/ Time-Sharing	Third Generation	Resource Utilization, User Interaction	ICs
Network / Distributed / Real-time	Fourth Generation	Resource Sharing, Communication	LSI/VLSI

1.1.1 First Generation Operating System

In the 1940s, the earliest electronic digital systems had no operating systems. Those days Machines run from a console with display lights, toggle switches, input device, and printer.These machines consists of thousands of vacuum tubes. They were very big (placed in rooms). They were very slower than even the cheapest personal computers available today. There were no programming languages available. Typical mathematical and scientific calculation problems are solved with computers. Programs are written on sheets and implemented by wiring out (Hardwired). Programs are directly executed on computers without any system software support. This approach is called as *Bare Machine Approach*.

1.1.2 Second Generation Operating System

In 1950's use of transistors made computers better in speed,reliability and cost in comparison with vacuum tube computers. Those computers were as mainframe computer systems. A program or set of programs which was called as a job is executed with help of Batch Monitor(Batch OS).

In Batch OS, major concern is to improve CPU Utilization by making ensure that processor is given some task at all time. Each set of jobs was considered as a batch.

For example, there may be a batch of short Fortran jobs shown in figure (1.3). Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run

as a group. The output for each job were available to users in a form of output tapes or papers. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. The operator then sorts programs into batches with similar requirements. The problems with Batch Systems are following:

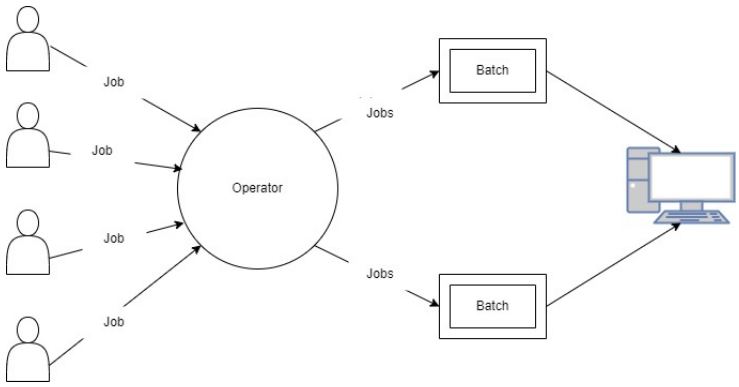


Figure 1.3: Batch Operating System

1. Lack of interaction between the user and job.
2. CPU is often idle, because the speeds of the mechanical I/O devices is slower than CPU.
3. Difficult to provide the desired priority.

1.1.3 Third Generation Operating System

While Second generation computers were consisting individual transistors, Third Generation Computers uses Integrated Circuits (ICs). Computers equipped with ICs were better in terms of price/performance ratio. OS/360 was developed by IBM introduced concept of Multiprogramming.

Multiprogramming

Multiprogramming is very important property of operating systems. A single user or program executed by user cannot utilize CPU or the I/O devices all times. In fact program consist of portions where CPU is used (known as CPU Burst) or I/O devices are used (I/O Burst). Multiprogramming increases CPU utilization by Allocating CPU to another program, when program uses I/O and so on. CPU Scheduler is a part of OS responsible for allocating CPU to programs. When, the first program finishes waiting(I/O) and it will gets CPU back. Multiprogramming increases CPU Utilization by ensuring CPU is busy almost all times.

In multiprogramming system, when one program is waiting for I/O transfer, there is another program ready to utilize the CPU. It is possible to load several jobs in memory for concurrent execution.

Spooling

The technique of *Spooling(Simultaneous Peripheral Operation On Line)* is also evolved with Third-generation operating systems. Spooling allows Multiple programs to do peripheral operations concurrently. With the spooling multiple application can use I/O Devices

without interfering each other. The operating system solves this problem by keeping Each application's output in separate disk file. When an application request for printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time.

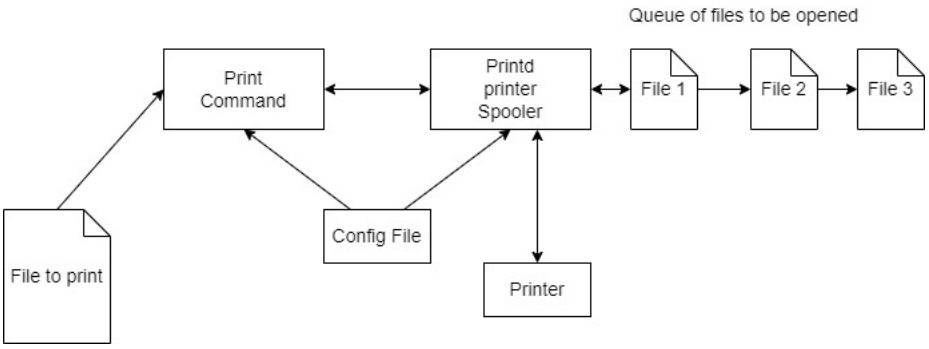


Figure 1.4: Spooling

Benefits of Spooling

1. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or other jobs) may be executed, reading its "cards" from disk and "printing" its output lines onto the disk.

2. Spooling is also used for processing data at remote sites. The CPU sends the data via communication paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention.

3. Spooling can keep both the CPU and the I/O devices working at much higher rates.

1.1.4 Timesharing or Multitasking Systems

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. *Time sharing* (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple tasks concurrently by switching among them. Important difference is multitasking allows users to interact with each program while program is running. All modern operating System(for example Windows, LINUX, UNIX) uses multitasking. We all can experience multitasking on our PCs/ Laptops, where multiple task or applications concurrently running. The objective of multiprogramming is to improve CPU Utilization while timesharing focus on user response(interaction). Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

Difference between Multiprogramming and Timesharing Operating System The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, objective is to maximize processor use,

whereas in Time-Sharing Systems objective is to minimize response time. Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receives an immediate response. For example, in a transaction processing, processor execute each user program in a short burst or quantum of computation. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is in few seconds at most. Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems(figure 1.5).

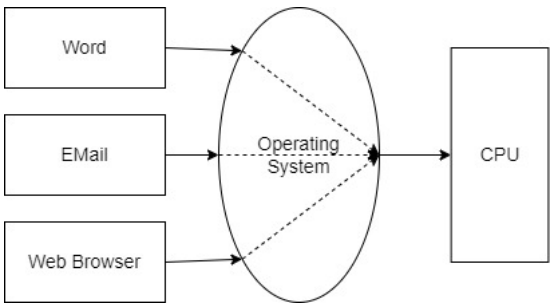


Figure 1.5: Time Sharing Operating System

Advantages of Timesharing operating system are:

1. Provide advantage of quick response.
2. Avoids duplication of software.
3. Reduces CPU idle time.

Disadvantages of Timesharing operating system are:

1. Problem of reliability.
2. Programs may interfere each other. Timesharing operating system needs mechanism for ensuring security and integrity of user programs and data.
3. Problem of data communication.

1.1.5 Fourth Generation Operating System

With the development of LSI (Large Scale Integration) circuits and VLSI, computers are available for more users. In fourth generation OS focuses on user interaction. Earlier OS like DOS,UNIX were simple, they provide CLI(Command Line Interface). CLI was suitable for sophisticated user but not enjoyed by naive users. GUI(Graphical User Interface) was developed to improve usability. Interaction with mouse was promoted by windows. An interesting development is the growth of networks of personal computers. Networking provides sharing of resources and communication. Operating systems which support networking were evolved as Network Operating Systems and Distributed Operating Systems .

1.2 Operating System Services

An operating system provides an environment for the execution of programs. It provides following services to programs and to the users of those programs.

1. User interface: User Interface (UI) take several forms. One is a command-line interface(CLI), which uses text commands. Another is a batch interface, in which commands stored into files, and those files are executed. Modern OS uses graphical user interface (GUI). Some systems provide two or all three of these variations.

2. Program execution: The system must be able to load a program into RAM and run that program.

3. I/O operations: A running program may read or write file or interact with I/O device. Examples are recording to a CD or DVD drive or editing a file. OS manages operation required to do I/O.

4. File Manipulation: The user generally store information in form of files. Each program need to create,delete,read and write files and directories. Program may search for a given file, and list file information. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership. OS needs to provide easy to use, reliable and consistent file system.

5. Communications: One program may interacts with another program. OS provides communication facility between programs whether they are executing on same computer or different computer systems connected by a computer network.

6. Error detection: The operating system needs to detect possible errors. Errors may occur in the CPU and memory hardware or in I/O devices and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location). Operating system should be able to detect errors and recover from those(exception handling).

7. Resource allocation: Multiple programs may request for same resources at simultaneously. OS allocates resources in a manner which resolves conflicts and provide better utilization and efficiency. Example of Computational Resources are processor, memory, files ,I/O devices. Accounting. OS keeps record of allocation of computer resources. This record keeping may be used for improving computing services.

8. Protection and security: Information stored may be used by attackers for bad purpose. OS needs to provides mechanism for protection and security. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security ensures protection from outsiders. One example of security mechanism is authentication with a password, to gain access to system resources.

1.3 Types of Operating System

Operating systems keep evolving over the period of time. Following are few of the important types of operating system which are most commonly used.

1.3.1 Mainframe Operating System

Mainframe Computers used by corporate and governmental organizations for critical applications,web servers, bulk data processing such as census, industry and consumer statistics,

enterprise resource planning and transaction processing. Mainframe Computers may have more than 1000 disks and Terabytes of storage.

A Mainframe Operating System is an Operating System (OS) on a mainframe computer to process large amounts of information and support a great number of users.

Mainframes are designed to handle very high volume input and output (I/O) and emphasize throughput (Number of jobs completed) computing. Mainframe OS offers three kinds of services:

1. Batch : Batch processing focuses on CPU Utilization, no interaction with jobs are required here.
2. Transaction Processing: It handles large number of small request. Example of transaction processing is check processing at a bank or airline reservation.
3. Timesharing : In Timesharing CPU switches between programs to provide interaction.

1.3.2 Server Operating System

Server Operating Systems run on servers, Which are very large personal computers or mainframes.Server Operating Systems works in client server networking environment(figure1.6).The serves large number of users connected with networks.Server operating systems can act as Web server, Mail server, File server, Database server, application server and Print server.

Examples of Server Operating System are Windows Server,UNIX Server,Sun Solaris, Mac OS X Server, Red Hat Enterprise Linux (RHEL) and SUSE Linux Enterprise Server.

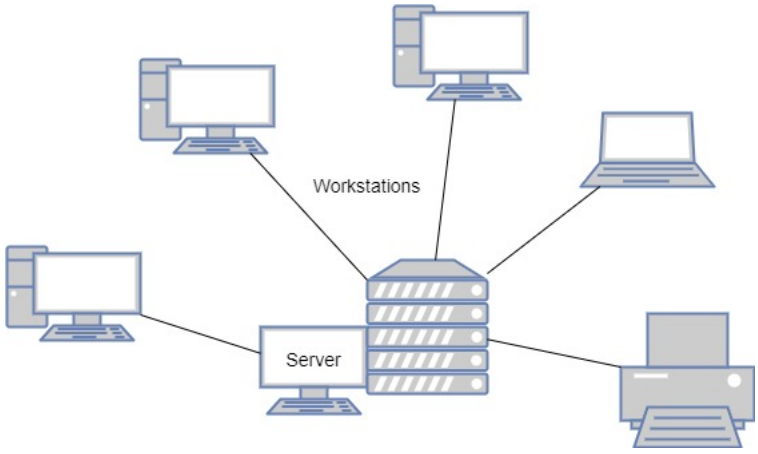


Figure 1.6: Server Operating System

Server OS provides following features:

1. GUI is not important or optional.
2. It is possible to reconfigure and update both hardware and software to some extent without restart.
3. It provides Advanced backup facilities to permit regular and frequent online backups of critical data.

4. It provides flexible and advanced networking capabilities,

5. It provides more Protection and Security features than PC Operating System.

1.3.3 Multiprocessor Operating System

Multiprocessor Operating System have two or more central processing units (CPU),sharing bus, memory and other peripheral devices. These systems are referred as tightly coupled systems(Rajagopal, 1999). Multiprocessor OS A computer system in which two or more CPUs share full access to Main Memory.

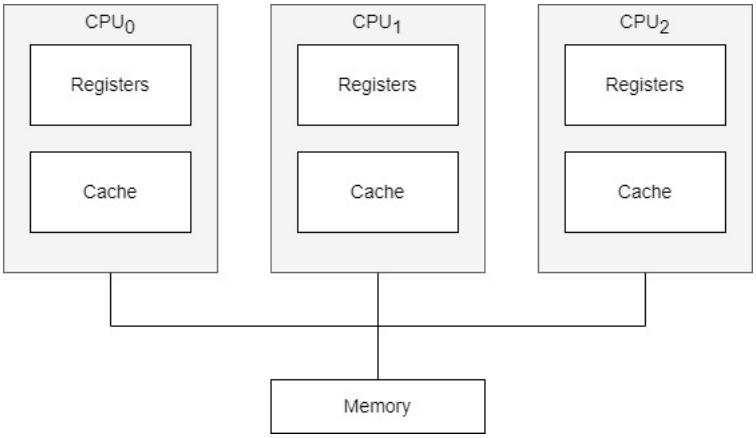


Figure 1.7: Multiprocessor Operating System

Benefits of Multiprocessor OS

1. Increase throughput: Multiprocessor OS uses multiple processor to complete more task per unit time. Although speed up ratio with N processors is not N but less than N.

2. Economy of scale: Multiprocessor systems can save more money than multiple single processor system, because they can share peripherals,mass storage and power supplies.

3. Increased reliability: They have *Graceful Degradation* property. It means failure of single processor will reduce speed of operation, but not stop working of entire system.

1.3.4 Personnel Computer(PC) Operating System

Personnel Computers are becoming powerful and inexpensive. The Operating System for PC's emphasizes user interaction and ease of use. Modern PC Operating Systems provide multitasking and easy to use GUI. Examples of PC Operating Systems are Windows(XP, Vista, 7, 8 ..), Macintosh, Linux(Fedora, Ubuntu ...).

1.3.5 Mobile Operating System

We are all aware about mobiles, PDA's(Personnel Digital Assistant),Tablets. These small devices also runs OS, which is referred as *Mobile Operating Systems*. Developers of mobile systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width. Because of their size, most mobile devices have a small amount of memory, slow

processors, and small display screens. The amount of physical memory in a handheld depends upon the device. As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory. Second important limitation is limited battery power. Mobile OS needs to use the battery power effectively. Modern Mobile Operating Systems combine the features of a PC Operating System with other features, including a touchscreen, cellular, Bluetooth, Wi-Fi, GPS mobile navigation, camera, video camera, speech recognition, voice recorder and music player. Examples of Operating System for mobile devices (smart phones and tablets) are Apple's iOS and Google's Android.

Figure 1.8: Mobile Operating System

1.3.6 Real Time Operating System

Real-time systems are those that are used to offer services such as ATM (Automatic teller machine) or airlines reservation systems. This systems are often called transaction oriented systems. This is because basic information processing involves a transaction. As an example, consider processing a request like withdrawing money. It is a financial transaction, which requires updating accounts within an acceptable response time. Both the process control systems and transaction oriented systems are real time systems. In this systems response time to a request is critical. The main objective is to meet deadlines. Embedded systems run real-time operating systems. A real-time system is used in control device in a dedicated application. Real time systems can be used in devices like washing machines, microwaves and smart TV's . Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are realtime systems. Some automobile engine fuel injection systems, home application controllers, and weapon systems are also real-time systems. Real-time systems can be classified as Hard Real-time Systems and Soft Real time Systems(figure 1.9)

- Hard Real time Systems: Hard real-time means you must absolutely hit every deadline. Very few systems have this requirement. Some examples are nuclear systems, some medical applications such as pacemakers, a large number of defense applications, avionics, etc.

- Soft Real time Systems: Some times hard real time systems may not provide interaction while executing jobs. Soft real-time systems can miss some deadlines but provide better response. We may observe performance degradation if too many deadlines are

missed. A good example is the sound system on your computer. If you miss a few bits, it will not affect.

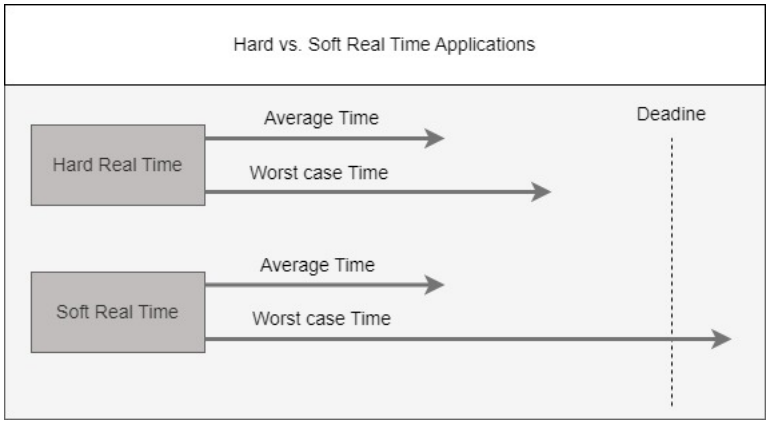


Figure 1.9: Real Time Operating System

1.4 Concepts of OS

1.4.1 Processes

The program consists of a sequence of instructions. The program is stored in the form of files in Secondary storage. A process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed.

"A process is a program in execution."(Silberschatz et al., 2012)

The process consists of the following resources:

1. Executable machine code of program.
2. Data Area to keep variables.
3. Stack to keep track of active subroutines and/or other events).
4. Heap to hold intermediate computation data generated during run time.
5. Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows)and data sources.
6. Security attributes, such as the process owner and the process' set of permissions (allowable operations).
7. Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

The operating system holds most of this information about active processes in data structures called process control blocks.

1.4.2 Files

A collection of data or information is stored in form of files. There are many different types of files: data files, text files , program files, directory files, and so on. Different types of files store different types of information. For example, program files store programs, whereas text files store text. On most modern operating systems, files are organized into one-dimensional arrays of bytes.

Most operating systems have the concept of a directory. We can think of the directory as a container that holds a group of files together. A student, for example, might have one directory for each course he or she is taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. Directory entries may be either files or other directories. Normally directories are arranged as tree form, such arrangement is called Hierarchical File System .

1.4.3 Address Spaces

All executing programs are stored in main memory. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one placed in memory. Multiprogramming operating systems allow multiple programs to be stored in memory at the same time. Operating system uses protection mechanism to keep them from interfering with one another A different, but equally important memory-related issue, is managing the address space of the processes. Normally, each process has some set of addresses it can use, typically running from 0 up to some maximum.

1.4.4 Input/Output

Input/output or I/O is the communication between a computer and user. Inputs are the signals or data received by the system and outputs are the signals or data sent from it. I/O devices are used by a human (or other system) to communicate with a computer. For example, a keyboard or mouse is an input device for a computer, while monitors and printers are output devices. Some devices for communication between computers, such as modems and network cards, perform both input and output operations. Any transfer of information to or from the CPU/memory, for example by reading data from a disk drive, is considered I/O.

1.4.5 Protection

Some information stored in a computer is confidential. This information may include e-mail, business plans, tax returns, and much more. It is up to the operating system to manage the system security so that files are only accessible to authorized users. If a computer program is run by unauthorized user then he/she may cause severe damage to computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc. Some operating system use dual mode operation as a protection mechanism.

Dual Mode Operations

In order to ensure proper execution of the OS, Operating system distinguish between the execution of operating-system code and user-defined code.The Dual Mode Operation of OS is shown in figure 1.10. OS uses two separate modes of operation: user mode and kernel

mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). In User mode, a subset of instructions is available. We can execute limited set of instructions.

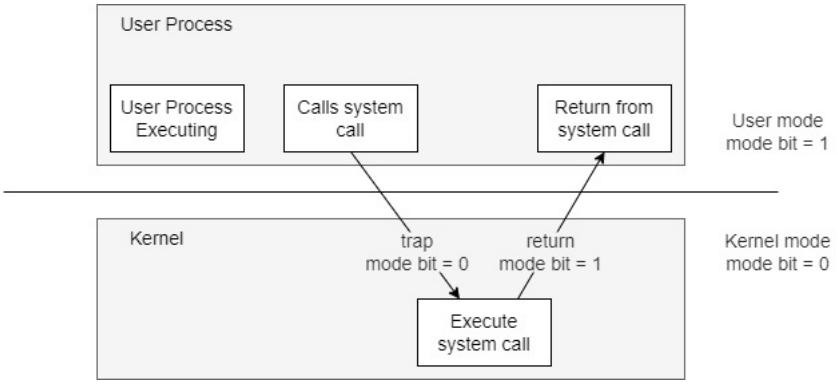


Figure 1.10: Dual Mode of Operation

Benefits of dual model of operation

1. *I/O protection*: all I/O operations are privileged; so user programs can only access I/O by sending a request to the (controlling) OS.

2. *Memory protection*: It uses base/limit registers (in early systems), memory management unit, (MMU, in modern systems) for protection. User programs can only access the memory that the OS has allocated.

For example CPU control, timer (alarm clock), context switch. User programs can only read the time of day,It can not change time. User program can only have as much CPU time as the OS allocates. When a user application requests a service from the OS (via a system call), it must request OS to change mode from user to kernel mode to fulfil the request.

1.4.6 The Shell

In UNIX operating system consists of two major parts: kernel and shell. Kernel performs low level tasks like process management,memory management and I/O management. UNIX shell provides interface through which user gives commands(figure 1.11) (ISRD, 2006). This commands are interpreted by shell. It makes heavy use of many operating system features. Many shells exist, including sh, csh, ksh, and bash. All of them support the functionality described below, which derives from the original shell (sh) (Bach, 1986).

1.5 System Calls

Most user processes require a system call to seek OS services. Systems call provide a library or API that sits between normal programs and the operating system. The system call is how a program requests a service from an operating system's kernel.

Program uses hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system (figure 1.12). Example of System calls:

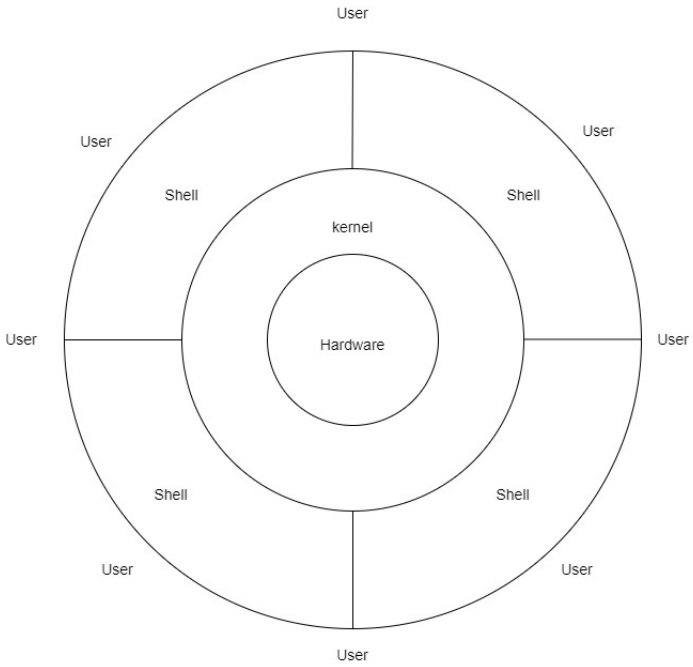


Figure 1.11: Shell

1. To create or terminate processes.
2. To access or allocate memory.
3. To get or set process attributes.
4. To create, open, read, write files.
5. To change access rights on files.
6. To mount or un-mount devices in a file system.
7. To make network connections.
8. Set parameters for the network connection.
9. Open or close ports of communication.
10. To create and manage buffers for device or network communication.

1.6 Operating System Structures

Operating system is a very large and complex software. Operating system should be stable, flexible, robust,efficient and easy to use. To meet this requirements OS needs to be designed with software engineering principles. One of the best practice to reduce complexity is to divide large OS into small components. Each of small components are called modules. Each module should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

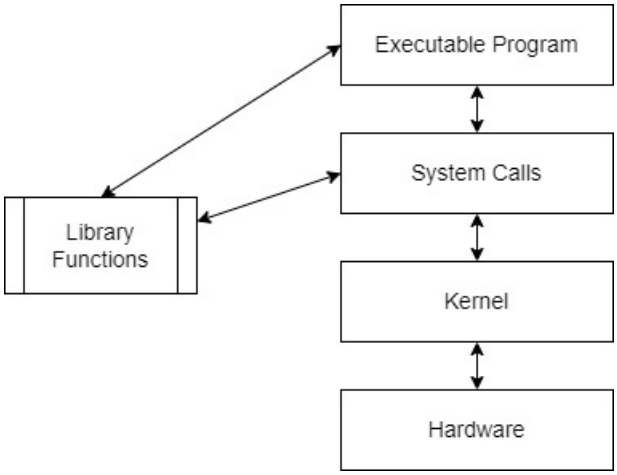


Figure 1.12: System Calls

1.6.1 Monolithic Systems

Many operating systems do not have well-defined structures.In this approach the entire operating system runs as a single program. If no systematic approach of system development is used than system is said to have monolithic structure. Small systems generally developed in same way.MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular.

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Benefits:

1. Monolithic kernels are faster.
2. Monolithic kernels are simple to implement.

Disadvantages:

1. Monolithic systems are not flexible.Addition/removal is not possible
2. Monolithic systems are difficult to debug.
3. Since the device driver reside in the kernel space it make monolithic kernel less secure.

1.6.2 Layered Systems

Layering is software engineering guideline for handling large and complex systems. In Layered model, We can break operating systems into smaller pieces or components(figure 1.13). The related components are kept in particular layer. The bottom layer is the hardware, while the highest layer is the user interface.Each layer can be changed without disturbing others.

The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can

be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Example of layered model is the THE system, which was built by E. W. Dijkstra (1968) and his students.

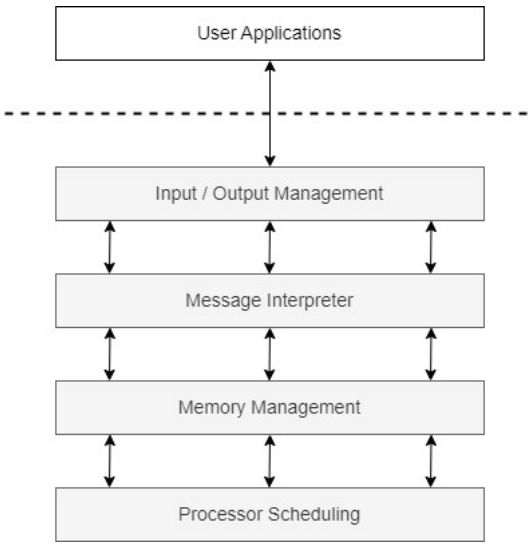


Figure 1.13: Layered Operating System

Benefits:

1. The layered operating system are flexible, modular and easy to build.
2. Each layer is independent of others that is the reason each can be developed independently. Overall development time can be reduced by dividing layers to various developer teams.

Disadvantages:

1. It requires an appropriate definition of the various layers.
2. Careful planning of the proper placement of the layer is necessary.
3. Layered OS are generally slower than monolithic and microkernel OS.

1.6.3 Microkernels

Microkernel OS structures the operating system by removing all nonessential components from the kernel. Only the very important parts like IPC(Inter process Communication), basic scheduler, basic memory handling and basic I/O primitives are put into the kernel. Others nonessential portions of the kernel are maintained as server processes in User Space. Communication between components of the OS is provided by message passing. Microkernel is a smaller kernel.The main function of the microkernel is to provide process management and communication facility between the client program(Liu et al., 2021). The other services runs in user space. Microkernel approach provides flexibility,modularity, security and reliability(figure1.14)

Benefits:

1. Extending the operating system becomes much easier.

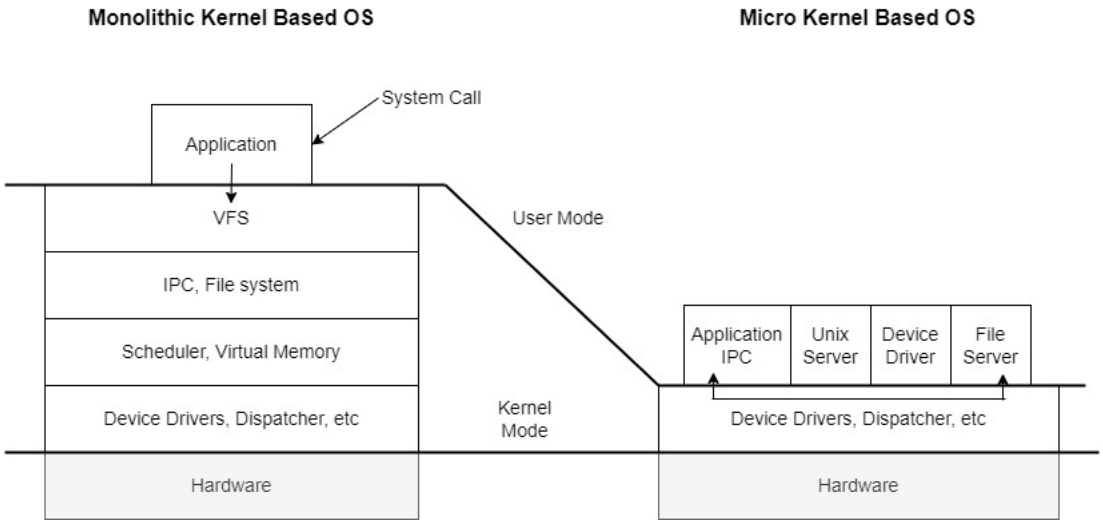


Figure 1.14: Microkernel Operating System

2. Since the kernel is smaller,we do not need to change kernel frequently.

3. The microkernel also provides more security and reliability.

Disadvantages:

1. Slower Processing due to additional Message Passing

Examples: Windows NT and MINIX are examples of microkernel OS.

1.6.4 Client-Server Model

With the availability of inexpensive PCs, Terminals connected to centralized systems have been replaced. PCs at client end have more processing power than terminals. In client server architecture clients PC request server. Server systems handles requests generated by client systems. Server provides some services to clients. Client can get services by requesting for service. Server respond by doing some processing and returning some result. Examples of computer applications that use the client–server model are Email, network printing, and the World Wide Web. Servers are classified by the services they provide. For instance, a web server serves web pages and a file server serves computer files. A shared resource may be any of the server computer's software and electronic components, from programs and data to processors and storage devices. The sharing of resources of a server constitute a service. Generally clients and servers runs on different computers, connected by a local or wide-area network. But they may run on same machines also. For example, a single computer can run web server and file server software at the same time to serve different data to clients making different kinds of requests. Client software can also communicate with server software within the same computer.

Benefits:

1. Client Server OS provides Enhanced Data Sharing.

2. We can share Resources among different Platforms(different OS).

3. Administrator can manage overall system from any place.

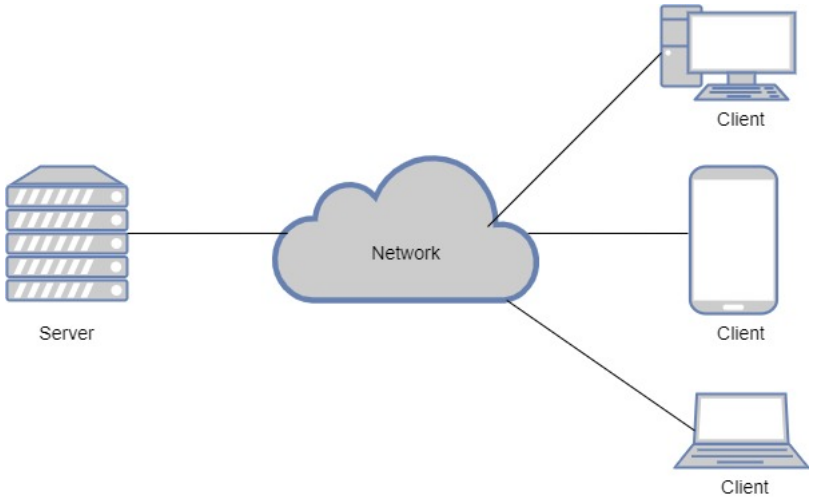


Figure 1.15: Client Server Operating System

Disadvantages:

1. High resource requirement at server end.
2. It requires more sophisticated software than PC operating system

1.6.5 Virtual Machines

Virtual machine creates the illusion that each separate execution environment(Operating systems) is running its own private computer. A virtual machine is a software computer that, like a physical computer, runs an operating system and applications. A virtual machine (VM) is an operating system OS or application environment that is installed on software which imitates dedicated hardware. The end user has the same experience on a virtual machine as they would have on dedicated hardware. Specialized software called a hypervisor emulates the PC client or server's CPU, memory, hard disk, network and other hardware resources completely, enabling virtual machines to share the resources. The hypervisor can emulate multiple virtual hardware platforms that are isolated from each other, allowing virtual machines to run Linux and Windows server operating systems on the same underlying physical host.

Benefits:

1. Virtual-machine system can be used as testing platform for new operating-systems. Operating systems are large and complex programs. It is difficult to modify OS. Because the operating system executes in kernel mode,a wrong change could cause an error that would destroy the entire system (Buyya et al., 2013). Thus, it is necessary to test all changes to the operating system carefully. Bug in OS will not affect actual hardware.
2. Multiple OS can be used on same computer.
3. Easy maintenance, availability and convenient recovery.

Disadvantages:

1. In Virtual Machine environment, Direct sharing of resources is not possible in some cases.

2. Virtual machine is not that efficient as a real one when accessing the hardware.

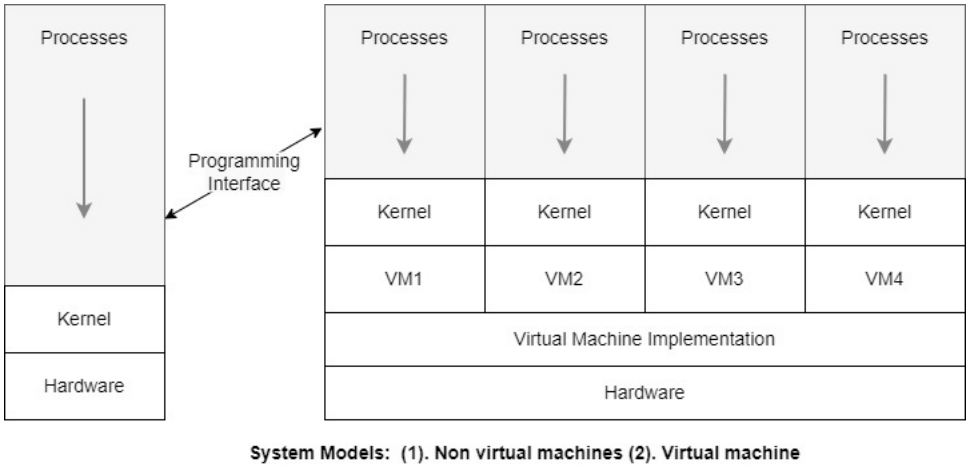


Figure 1.16: Virtual Machine

Example: VMware VMware is a popular virtual machines software. VMware runs as an application on a host operating system such as Windows or Linux. It allows this host system will run concurrently several different guest operating systems. Each guest OS will run independent on their virtual machines. Consider the following scenario: A developer has designed an application and would like to test it on Linux, FreeBSD, Windows NT, and Windows XP. One option is for her to obtain four different computers, each running a copy of one of these operating systems. Another alternative is for her first to install Linux on a computer system and test the application, then to install FreeBSD and test the application, and so on. This option allows her to use the same physical computer but is time-consuming, since she must install a new operating system for each test.

Virtual Machine software like VMware allows multiple operating system running concurrently Such testing could be accomplished concurrently on the same physical computer.

1.6.6 Exokernels

Exokernel is an operating system kernel developed by the MIT Parallel and Distributed Operating Systems group(Tanenbaum and Tanenbaum, 1995). Generally Kernel handles low-level responsibilities of controlling hardware (particularly memory allocation). In exokernel application programmer can directly interact with hardware. Most of the developers prefer that OS takes responsibility of this low-level tasks, because they want to focus on writing applications.In some cases developer wants to to control hardware directly without interacting with OS. An exokernel just allocates physical hardware resources to programs. This allows the program to use library operating systems.Exokernel architectures may have great performance benefits, If program is correctly written. But in some case of bugs, it may lead to crash. Exokernels are very small, since functionality is limited to ensuring protection and multiplexing of resources.

Benefits:

1. We can build higher performance applications by giving them flexible, extensible access to OS primitives.

Disadvantages:

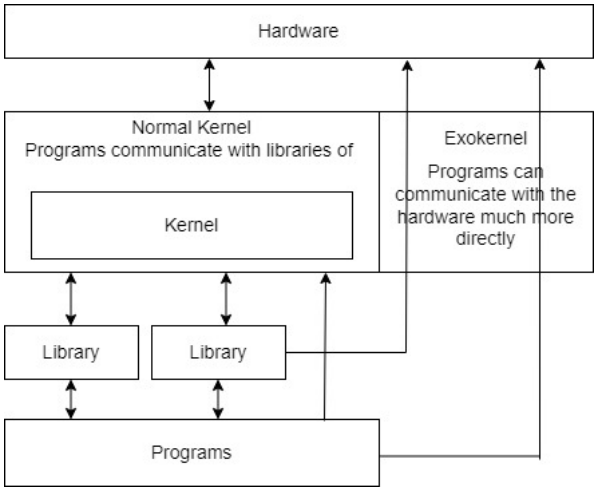


Figure 1.17: Exokernel

1. Exokernel technology is still not thoroughly researched and tested.
2. Different applications to run simultaneously on the same system, that would also mean different look and feels for each of them.(SINHA, 1998)

Conclusion: In conclusion, this book chapter provides an in-depth analysis of operating systems, highlighting their critical role in the effective management of computer resources. By understanding the core functions such as process management, memory management, and file systems, readers gain insight into the complexities and efficiencies of operating systems. The discussion of different types of operating systems, including batch, time-sharing, and real-time systems, underscores the versatility and adaptability of operating systems in various computing environments. The chapter emphasizes the continuous evolution of operating systems to meet the growing demands of technology and user requirements, underscoring their pivotal role in the advancement of computer science.

1.7 Exercise

1. What are the two main functions of an operating system?
2. What is the difference between time sharing and multi-programming systems?
3. Which of the following instructions should be allowed only in kernel mode? (a) Disable all interrupts. (b) Read the time-of-day clock. (c) Set the time-of-day clock. (d) Change the memory map.
4. Describe system calls?
5. Explain development of OS with different generation of computers?
6. What do you understand by spooling? Give various advantages of it.
7. What is the difference between time-sharing and multi programming systems?
8. What are the difference between Hard and Soft Real-time operating systems?

9. Discuss various constraints in developing Mobile operating systems.
10. Compare Micro-kernel with Monolithic operating systems.
11. Describe OS Services.
12. What do you understand by files?
13. Explain basic services provided by Operating System on bare Hardware machine.
14. What is Virtual Machine? Explain virtual machine & Client/Server architecture of Operating System?

1.8 Multiple Choice Questions

1. To access the services of the operating system, the interface is provided by the _____
   (a) Library
   (b) Assembly instructions
   (c) System calls
   (d) API
2. Which one of the following is not true?
   (a) kernel remains in the memory during the entire computer session
   (b) kernel is made of various modules which can not be loaded in running operating system
   (c) kernel is the first part of the operating system to load into memory during booting
   (d) kernel is the program that constitutes the central core of the operating system
3. Which is true for real time operating system:
   (a) process scheduling can be done only once
   (b) all processes have the same priority
   (c) kernel is not required
   (d) a task must be serviced by its deadline period
4. Windows is a(n)
   (a) Operating System
   (b) Database Management System
   (c) Compiler
   (d) Word Processing System
5. Embedded operating system is used in
   (a) On a networked PC
   (b) On a mainframe
   (c) On a desktop operating system

(d) On a PDA

6. Which among these requires a device driver?

   (a) Register
   (b) Cache
   (c) Main memory
   (d) Disk

7. The purpose of multiprogramming is to use the processor and peripherals in

   (a) efficient way
   (b) bad way
   (c) systematic way
   (d) functional way

8. The main objective of time sharing operating system:

   (a) Avoid thrashing
   (b) Faster execution
   (c) Increase multiprogramming
   (d) Provide faster response to user

9. What operations mentioned are done by an operating system?

   (a) Maintaining recycle bin
   (b) Transfer files
   (c) Opens a program
   (d) All of the above

10. Which one of the following isn't considered a real-time operating system?

    (a) PSOS
    (b) Windows
    (c) linuxRT
    (d) VRTX

11. Which one of the following isn't a mobile operating system?

    (a) DOS
    (b) Android
    (c) iPhone OS
    (d) Blackberry

12. Which multiprocessor system contains a master slave relationship?

    (a) Symmetric Multiprocessors
    (b) Singleton Multiprocessors

(c) Asymmetric Multiprocessors

(d) Both A and B

13. The main objective in building the multiprocessor OS is

(a) greater throughput

(b) enhanced fault tolerance

(c) Both A and B

(d) None of the above

14. ____ is a large kernel, including scheduling file system, networking, device drivers, memory management and more.

(a) Monolithic kernel

(b) Micro kernel

(c) Macro kernel

(d) Mini kernel

15. In ___ architecture assigns only a few essential functions to the kernel, including address spaces, Inter process communication (IPC) and basic scheduling.

(a) Monolithic kernel

(b) Micro kernel

(c) Macro kernel

(d) Mini kernel

16. ____ is the objective of multiprogramming?

(a) To minimize CPU utilization

(b) Have multiple programs waiting in a queue ready to run

(c) Have some process running at all times

(d) Both B & C

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 2

Processes

Abstract: This chapter provides a comprehensive overview of process management within operating systems. It delves into the lifecycle of a process from creation to termination, emphasizing the importance of process control blocks (PCBs) and the various states a process can be in. The chapter also explores the differences between processes and threads, detailing the advantages and disadvantages of user-level and kernel-level threads. Through detailed explanations and examples, the chapter elucidates how operating systems handle multitasking, scheduling, and synchronization to manage multiple processes efficiently.

Keywords: Process management, Process Control Block (PCB), Multitasking, Threads, User-level threads, Kernel-level threads, Process states, Scheduling, Synchronization.

The objective of this chapter is to explain how a process is created and how processes are managed. In computer, when we click any icon (file icon or application), the application starts up. Anytime when we run a program (application), it needs to be loaded in main memory. Active Program may interact with Input Output devices (Keyboard,Mouse, Display,Printer). Active or Running program is called as a process. Process can be defined as:

A process is a program in execution.

A process require resources like CPU time, memory, files, and I/O devices at the time of execution. These resources are allocated to the process either when it is created or while it is executing. Programs are executed sequentially. However, in real-time, one processor is shared among multiple concurrently running programs, a concept known as multitasking. So one running program may be blocked and resume its work after sometime. The operating system is responsible *Process Management*. It handles creation and deletion of processes, scheduling, synchronization, communication, and deadlock handling for processes. It may happen that two or more processes are instantiated with same program. For example when we open multiple files in editor, one process is responsible for handling each file.

2.1 Process

To understand how process is created, lets imagine that we have written a program called a.c in C. On execution, this program may read in some data and output some data.When

we save this program as a.c, it is simple file stored in hard disk . It has no dynamics of its own means, it cannot cause any input processing or output to happen. When we compile a C language program we get an executable file. After that when we run this program,it is loaded in main memory. The processor process instructions in program and it may interact with I/O devices.

[Difference between Program and Process] Program is a text script, a program in execution is a process. In other words, A process is an executable entity – it's a program in execution.

Operating system maintains information about process in *Process Table* . Process table contains the all information required to run , block and resume process(In multiprogramming/ multitasking OS process may blocked or suspended). A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time. Figure 2.1 shows program is stored in secondary storage, It becomes process when it is loaded in main memory.

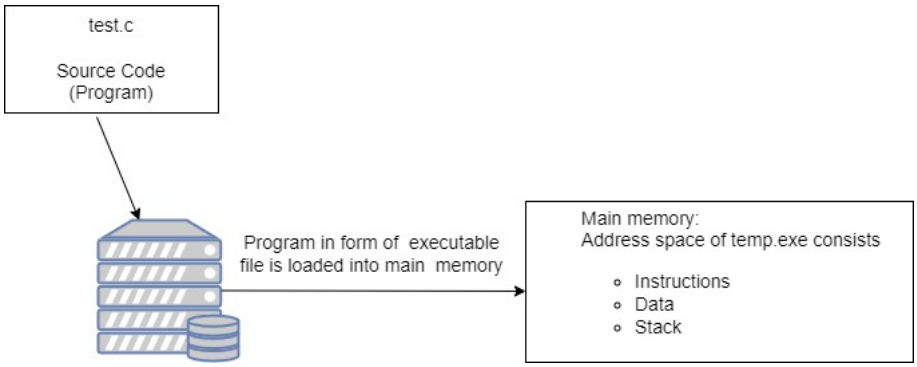


Figure 2.1: Program vs Process

2.2 Process Model

In this model, all programs running on the computer, including the operating system, are considered as processes. A process is an instance of an executing program. We can think each process execute sequentially but in reality, the real CPU switches back and forth from process to process. To understand the system, it is much easier to think about a collection of processes running in parallel than to try to keep track of how the CPU switches from program to program.This rapid switching is called multiprogramming.

2.2.1 Degree of Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. The number of process loaded in memory are called as *Degree of Multiprogramming*. If process computes only 20% of the time and other time it is spending in I/O or in other activity. We can

load five processes in memory to utilize CPU. Then we will be able to utilize CPU 100%. Because when process goes for I/O other process will be ready to use CPU. In reality all five processes may wait for I/O at the same time. A better model can be considering probability. If a process is having *p* portion of I/O. If n processes is loaded in memory. We can observe that CPU utilization can be increased with increasing Degree of Multiprogramming. The CPU utilization is then given by the formula:

$$CPUWaste = p^n \tag{2.1}$$

$$CPUUtilization = 1 - CPUWaste \tag{2.2}$$

$$CPUUtilization = 1 - p^n \tag{2.3}$$

For example if processes spend 80% of their time in I/O. If we load 10 processes in memory then CPU waste will be around 10%. We will be able to get CPU Utilization up to 90%

$CPUWaste = 0.8^{10}$
$CPUWaste = 0.1$
$CPUUtilization = 1 - 0.1$
$CPUUtilization = 0.9$

This model is only approximation, In reality we will not get increase in CPU Utilization with increase in Degree of Multiprogramming after certain limit. These are few reason:

1. When one process goes for I/O and other process is allocated to CPU. The time required in this activity is not used in processing. This time to schedule other process is *Context Switch*.

2. We are assuming all n processes are independent.It is possible that one process is waiting for others to complete or may waiting for resource which is in use by other process. It may not be possible sometime to completely run process independently.

3. The size of main memory also limits Degree of Multiprogramming.

2.3 Process States

the process state refers to the current condition or stage of execution that a process undergoes during its lifetime. It helps the operating system manage and track the progress and resources associated with each process. When we run a process, It may be running or blocked or terminated. Process state is current operating condition. Figure 2.2 shows transition among process states. Each process may be in one of the following states:

- New: The process is being created. OS still not scheduled process to execute.

- Ready: It enters the ready state when it is considered for scheduling.The process is waiting to be assigned to a processor.

- Running: When a processor is available then one of the processes in "Ready" state may be chosen to run. It moves to the state "Running".Instructions of process are being executed.

- Waiting: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Terminated: The process has finished execution.

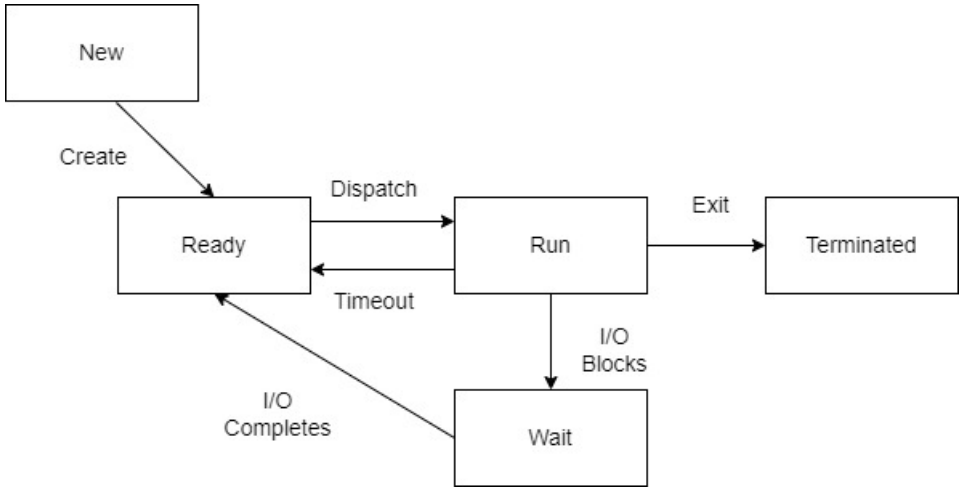


Figure 2.2: Process States

2.4 Process Control Block

Each process is represented in the operating system by a process control block (PCB). OS maintain all information about process like it's state, list of open files, current instruction being executed in a data structures for management of processes.This data structure is PCB (Figure 2.2). It is also known as task control block. It contains many pieces of information associated with a specific process, including these:

- Process state: The state may be new, ready, running, waiting and terminated.
- Program counter: The counter indicates the address of the next instruction to be executed for this process.
- CPU register: The computer uses registers as high speed temporary storage. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- Scheduling Information: This information includes a process priority and other scheduling parameters.
- Memory Information: This includes information about memory allocate to process. Example is value of the base and limit registers which indicate starting and ending memory address of process.
- Accounting Information: This information includes the amount of CPU and real time used, time limits.
- I/O status Information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

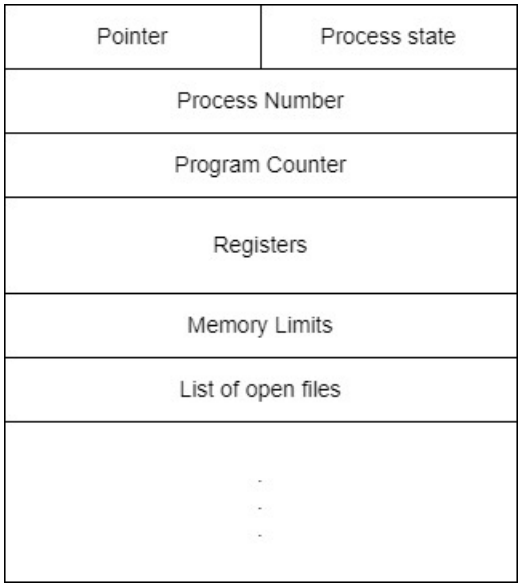


Figure 2.3: Process Control Block

2.5 Context Switch

A context switch occurs when a computer's CPU switches from one process or thread to a different process or thread.This allow for one CPU to handle multiple processes without the need for additional processors. Multitasking operating system uses context switch to allow different processes to run at the same time. The whole process of context switch is shown in Figure 2.4.

A context switch is the process of storing and restoring the state (context) of a process or thread so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system. What constitutes the context is determined by the processor and the operating system.

Typically, there are three situations that a context switch is necessary:

- Multitasking : When the CPU needs to switch processes in and out of memory, so that more than one process can be running.
- Kernel/User Switch : When switching between user mode to kernel mode.
- Interrupts : When the CPU is interrupted to return data from a disk read.

In context switching OS stores and restores process state(containing contents of registers, program counter and stack). Process state is generally contained in *PCB(Process Control Block)* (Anderson and Dahlin, 2014). Switching from one process to another requires a certain amount of time for doing the administration – saving and loading registers and memory maps, updating various tables and lists etc. Context switches are usually computationally intensive. The time required for context switch is overhead because it is not used in any processing. Most operating systems optimize context switches time to improve performance.

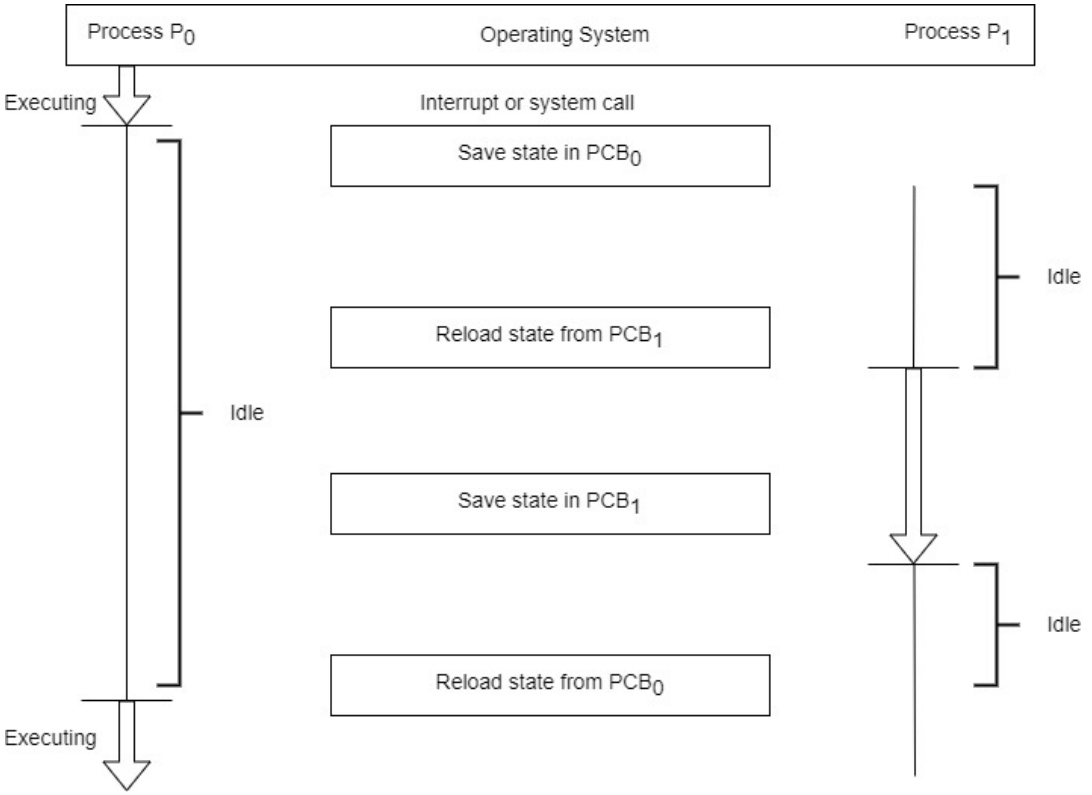


Figure 2.4: Process Context Switch

Context switching can be performed primarily by software or hardware. Some processors, like the Intel 80386 and its successor,have hardware support for context switches. They provide use of a special data segment designated the task state segment or TSS. When a task switch occurs the CPU can automatically load the new state from the TSS.Mainstream operating systems, including Windows and Linux, do not use this feature.Hardware context switching does not save all the registers (only general purpose registers). Software context switch is implemented in software(OS). This switching can be selective and store only those registers that need storing, whereas hardware context switching stores nearly all registers whether they are required or not.

2.6 Operation on Process

The processes can execute concurrently, and they may be created and destroyed dynamically. The operating system provides a mechanism for process creation and termination.

2.6.1 Process Creation

A process may create several new processes. Process can create new process with help of a create-process system call. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes can create other processes, forming a tree of processes.There are four principal events that cause a process to be created:

1. System initialization.

2. Execution of process creation system call by a running process.

3. A user request to create a new process.

4. Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that interacts with a (human) user and perform work for them. Other are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming e-mails, sleeping most of the day but suddenly springing to life when an incoming e-mail arrives. Another background process may be designed to accept an incoming request for web pages hosted on the machine, waking up when a request arrives to service that request.

Process creation in UNIX and Linux are done through fork() or clone() system calls (Tanenbaum and Bos, 2014). There are several steps involved in process creation. The first step is the validation of whether the parent process has sufficient authorization to create a process. Upon successful validation, the parent process is copied almost entirely, with changes only to the unique process id, parent process, and user-space. Each new process gets its own user space.

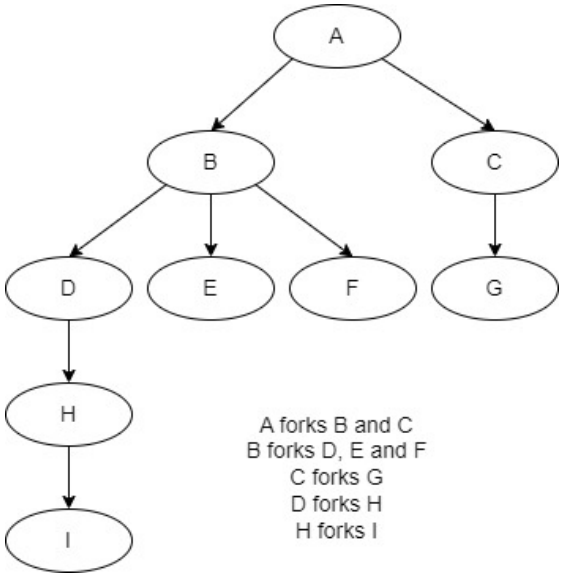


Figure 2.5: Process Hierarchy

2.6.2 Process Termination

A process terminates when it finishes executing its last statement. Its resources are returned to the system. It is deleted from any system lists or tables, and its process control block (PCB) is erased. Its memory space is returned to a free memory pool. The new process terminates the existing process, usually due to following reasons:

1. Normal Exist: Most processes terminates because they have done their job. This call is exist in UNIX.

2. Error Exist: When process discovers a fatal error. For example, a user tries to compile a program that does not exist.

3. Fatal Error: An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.

4. Killed by another Process: A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is kill. In some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

2.7 Threads

Threads are same as *Processes*, to do more than one thing at a time. Like processes, threads also to run concurrently.

A thread is a basic unit of CPU utilization.We may have one or more thread within a same process.Threads are a finer-grained unit of execution than processes. That is the reason sometimes they are called *Lightweight Processes*.

When you invoke a program, OS creates a new process. Process may create a single thread, which runs the program sequentially. That thread can create additional threads to perform different tasks concurrently.

For example, when we run a word-processor program, a single thread is executing to perform all task. This single thread allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run spell checker within the same process. But Modern word processor like Microsoft Word allows user to type at the same time spellchecker and auto recovery thread(Which saves text at periodical interval) is also working. Multiple threads in same Microsoft Word process made it possible.

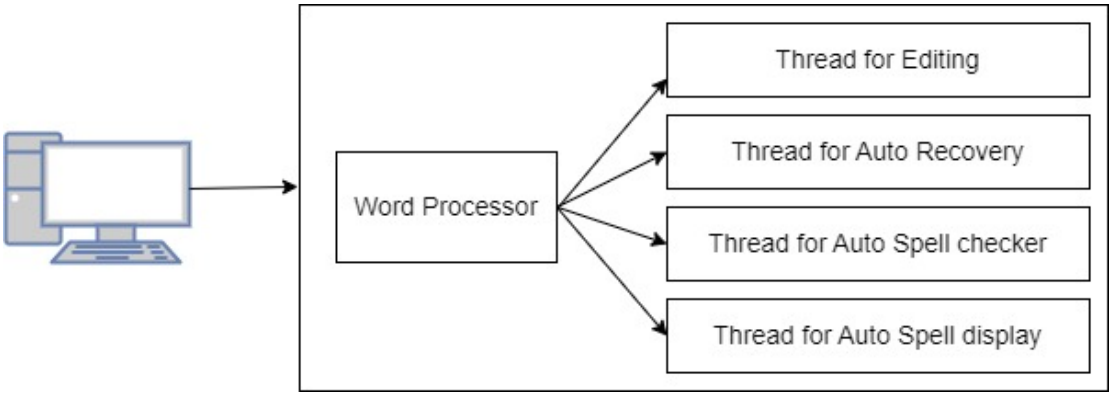


Figure 2.6: Thread Example: Word Processor Application process may contain different threads

2.7.1 Benefits

These are few *reasons* or *benefits* to use threads over process:

- Concurrency : In many applications multiple activities are going on concurrently.By decomposing such an application into multiple sequential threads that run in virtually parallel manner, the programming model becomes simpler.

- Responsiveness : Multi threading may allow a program to interact even if part of it is blocked or doing processing. For example, a multi-threaded web browser allow user to open new tab while other tab is loading.
- Faster: They are lighter weight than processes. they can be created and destroyed faster and easily than processes. In many systems, creating a thread 10-100 times faster than creating a process.
- Performance: Threads may not give performance gain when all of them are CPU bound. Most of the time a task is mix of CPU burst and I/O burst. Threads can overlap these activities, thus speeding up the application.
- Use of Multiple CPUs: Threads are useful for Multicore / Multiprocessor systems. In those systems, Each execution unit can be assigned a single thread. In this way we can utilize multiple processor.

2.7.2 Difference between thread and process

Thread generally contains less information than process. It consists of a thread ID, a program counter, a register set, and a stack. It shares code section, data section, and other operating-system resources, such as open files and signals with other threads in same process. We summarize here difference between Threads and Processes:

1. Processes are typically independent, while threads exist as subsets of a process.
2. Processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources.
3. Processes have separate address spaces, whereas threads share their address space.
4. Processes interact only through system-provided inter-process communication mechanisms.
5. Context switching between threads in the same process is typically faster than context switching between processes.

2.8 Types of Threads

Threads are useful in writing interactive application and improving CPU utilization. Threads are supported at two levels :kernel threads and user threads. User threads are supported above the kernel. They are managed without kernel support. User threads runs in user space. Kernel threads are supported and managed directly by the operating system. Kernel threads runs in kernel space. Virtually all operating systems for example Windows XP, Linux, Mac OS X, Solaris, and UNIX support kernel threads. A thread library provides the programmer an API for creating and managing threads.

Three main thread libraries are in use today:

1. POSIX Pthreads
2. Win32
3. Java

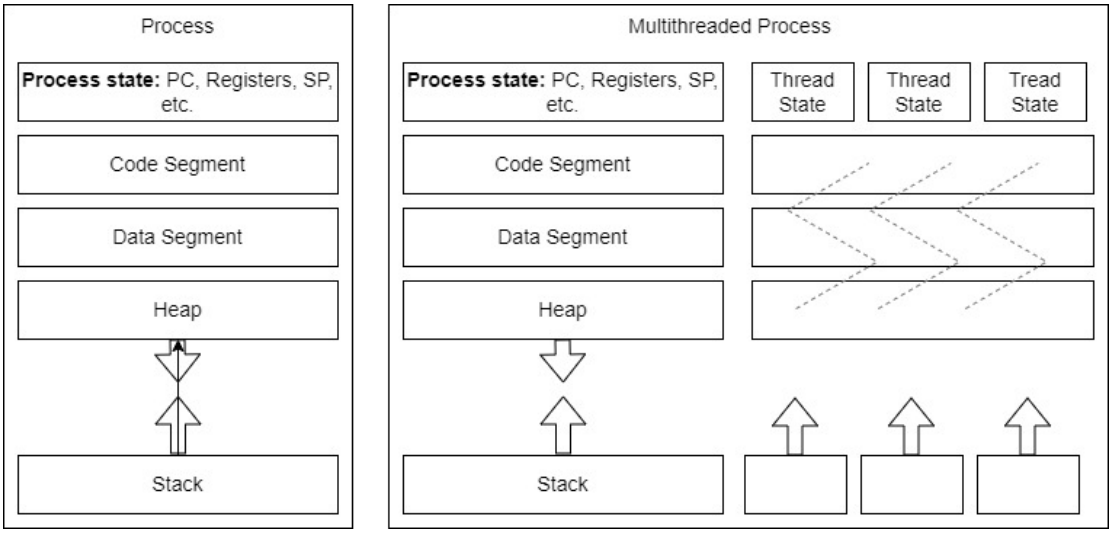


Figure 2.7: Thread vs Process

Pthreads, the threads extension of the POSIX standard, supports both user and kernel level threads. Pthreads library is available in UNIX and LINUX OS.The Win32 thread library is a kernel-level library available on Windows systems. The Java thread API allows thread creation and management directly in Java programs.

2.8.1 User Level Threads

The first approach is to provide a library entirely in user space with no kernel support. User-Level threads can be created easily than process because, much less state to allocate and initialize. They are implemented at user level. User-Level threads are managed entirely by the run-time system (user-level library).The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.User-Level threads are small and fast, each thread is represented by a PC,register,stack, and small thread control block. Creating a new thread, switching between threads, and synchronizing threads are done via procedure call i.e. no kernel involvement. User-Level threads are hundred times faster than Kernel-Level threads.

Benefits:

1. A user-level threads package can be implemented on an Operating System that does not support threads.
2. User threads are easy and fast to create.
3. Communication
4. Thread switching is not much more expensive than a procedure call.

Disadvantages:

1. User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, Os can make poor decisions like scheduling a process with idle threads. OS may blocking a process whose thread initiated an I/O even though the process has other threads that can run. OS may stop a process with a thread holding a lock. Solving this requires communication between between kernel and user-level thread manager.

2. There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.

3. User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Examples: Java implements threads as User level threads.

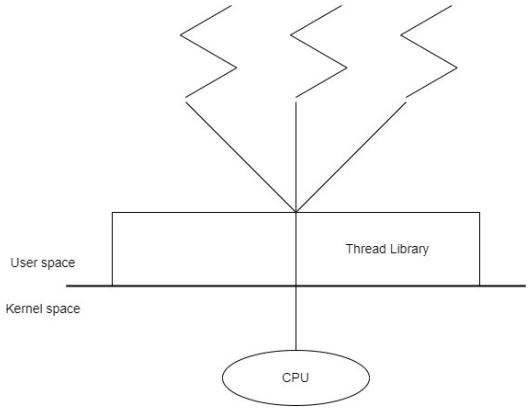


Figure 2.8: User Level Threads

2.8.2 Kernel Level Threads

Operating system manages threads. All thread operations are implemented in the kernel. The OS schedules all threads in the system. OS managed threads are called kernel-level threads. They sometimes are called *Light Weight Processes*. In this method, the kernel knows about threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Benefits:

1. Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.

2. Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

1. The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.

2. Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

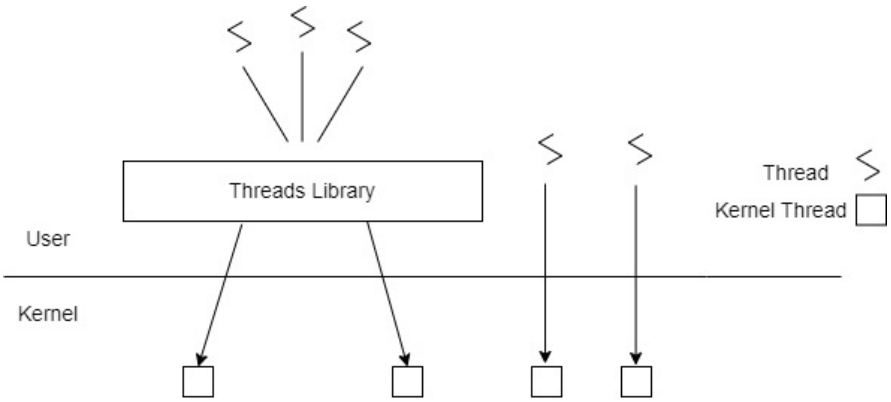


Figure 2.9: Kernel Level Threads

Examples: Windows NT implement Kernel Level Threads and Solaris as Lightweight processes(LWP).

Conclusion: The management of processes is a fundamental aspect of operating system functionality. Understanding how processes are created, managed, and terminated provides insight into the efficiency and effectiveness of an operating system. This chapter highlighted the critical role of process control blocks and the various states processes can transition through during their lifecycle. Additionally, the comparison between processes and threads, and the discussion on user-level versus kernel-level threads, provided a deeper understanding of concurrent execution within operating systems. Effective process management ensures optimal CPU utilization and system performance, which are essential for modern computing environments.

2.9 Exercise

1. What do you understand by process management?
2. How process is different than program?
3. Explain process states with state transition graph.
4. What is the purpose of PCB. Write various contents of PCB?
5. Explain the concept of the Program Counter (PC) and its role in process execution. What happens to the PC during a context switch?
6. Discuss the advantages and disadvantages of a monolithic kernel design compared to a microkernel design for managing processes.
7. Explain process of context switch. Differentiate between H/W and S/W context switch.
8. Why threads are useful. Compare threads with process?
9. Describe user level threads and kernel threads.
10. What is System Call in OS? Explain fork() system call to create new process in UNIX OS.

11. Differentiate between the "Ready" and "Waiting" states in the process model.

12. Discuss the role of the kernel in context switching for kernel-level threads.

2.10 Multiple Choice Questions

1. Which is not the state of a process of the following?

   (a) Old
   (b) New
   (c) Waiting
   (d) Running

2. The Process Control Block is a ____:

   (a) Process type variable
   (b) Secondary storage section
   (c) Data Structure
   (d) None of above

3. Following contains an entry for each process present in the operating system.

   (a) Program Counter
   (b) Process Table
   (c) Process Register
   (d) Process Unit

4. The degree of multiprogramming is number of processes

   (a) Loaded in memory
   (b) In the I/O queue
   (c) Executed per unit time
   (d) Created

5. A process terminates when_____:

   (a) It is removed the job queue
   (b) Its process control block is de-allocated
   (c) It is removed from waiting queue
   (d) None of the above

6. _____ is initiated by the user process itself in the only state transition:

   (a) wakeup
   (b) dispatch
   (c) block
   (d) none of the mentioned

7. A process control block should contain _____?

   (a) Process ID
   (b) A list of all open files
   (c) Location to store register value
   (d) All of these

8. In the PCB the context of a process does not contain:

   (a) Memory-management information
   (b) Value of the CPU registers
   (c) Context switch time
   (d) Process state

9. The interval from the time of submission of a process to the time of completion is termed as _____

   (a) waiting time
   (b) turnaround time
   (c) response time
   (d) throughput

10. A process executes the following code

    for (i = 0; i < n; i++) fork();

    The total number of child processes created is

    (a) n
    (b) 2^n - 1
    (c) 2^n
    (d) $2^{(n+1)} - 1$

11. A thread is usually defined as a "light weight process" because an operating system (OS) maintains smaller data structures for a thread than for a process. In relation to this, which of the following is TRUE?

    (a) On per-thread basis, the OS maintains only CPU register state
    (b) The OS does not maintain a separate stack for each thread
    (c) On per-thread basis, the OS does not maintain virtual memory state
    (d) On per-thread basis, the OS maintains only scheduling and accounting information

12. The time taken to switch between user and kernel modes of execution be t1 while the time taken to switch between two processes be t2.

    Which of the following is TRUE?

    (a) t1 > t2
    (b) t1 = t2

(c) t1 $<$ t2

(d) nothing can be said about the relation between t1 and t2

13. Consider the following statements about user level threads and kernel level threads. Which one of the following statement is FALSE?

(a) Context switch time is longer for kernel level threads than for user level threads.

(b) User level threads do not need any hardware support.

(c) Related kernel level threads can be scheduled on different processors in a multi-processor system.

(d) Blocking one kernel level thread blocks all related threads.

14. Termination of the process terminates _____

(a) first thread of the process

(b) first two threads of the process

(c) all threads within the process

(d) no thread within the process

15. When does a context switch occur _____?

(a) When an error occurs in a running program

(b) When a process makes a system call

(c) When an external interrupt occurs

(d) All of these

16. An operating system module that performs context switching is called _____?

(a) Context switcher

(b) Dispatcher

(c) CPU scheduler

(d) None of these

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 3

Scheduling

Abstract: The chapter on operating systems delves into the intricate mechanisms and algorithms that manage the execution of processes in a computer system. It begins with an overview of scheduling queues and the types of schedulers, highlighting their roles in process management. The chapter then explores various scheduling algorithms such as First-Come, First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin Scheduling, providing detailed examples and comparisons of their performance. It also covers multiprocessor scheduling algorithms and evaluates their effectiveness. By analyzing the strengths and weaknesses of each algorithm, the chapter aims to equip readers with a comprehensive understanding of process scheduling in operating systems.

Keywords: Operating Systems, Scheduling Queues, Process Management, First-Come, First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, Round Robin Scheduling, Multiprocessor Scheduling, Scheduler Types, Process Scheduling Algorithms,

Computer frequently switches among multiple processes or threads in case of Multiprogramming. There may be two or more processes are staying in the ready state. Scheduler, which is the part of OS is responsible in making decision that which process to run next. Scheduler uses various algorithm for it, which are called as scheduling algorithms.

3.1 Scheduling Queues

Data structures play an important role in management of processes. OS may use more than one data structure in the management of processes. The transition of process from one queue to another is shown in Figure 3.1.
It may maintain following scheduling queues:

- Ready Queue: All ready to run processes are contained in queue.
- Waiting Queue: OS maintains separate queues for blocked processes. It may even have a separate queue for each of the likely events (including completion of IO).
- Job Queue: When the process enters into the system, it is put into a job queue. This queue consists of all processes in the system.
- Device Queue: It is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

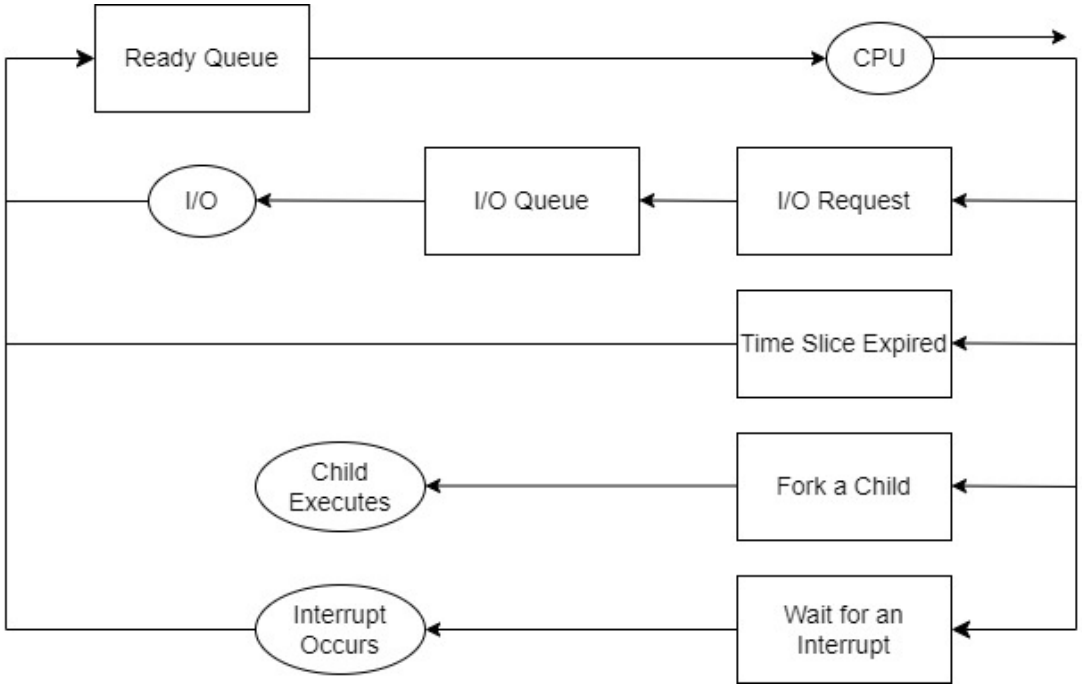


Figure 3.1: Scheduling Queue

3.2 Scheduler's Types

Scheduler's main task is to select the jobs to be submitted into the system and to decide which process to run. Figure 3.2 shows types of scheduler and their interaction with various scheduling queues.
Schedulers are of three types:

1. Long Term Scheduler It is also called *job scheduler*. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

2. Short Term Scheduler It is also called *CPU scheduler*. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them. Short term scheduler also known as *dispatcher*, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

3. Medium Term Scheduler Medium term scheduling is part of the *swapping*. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

[Swapping] Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

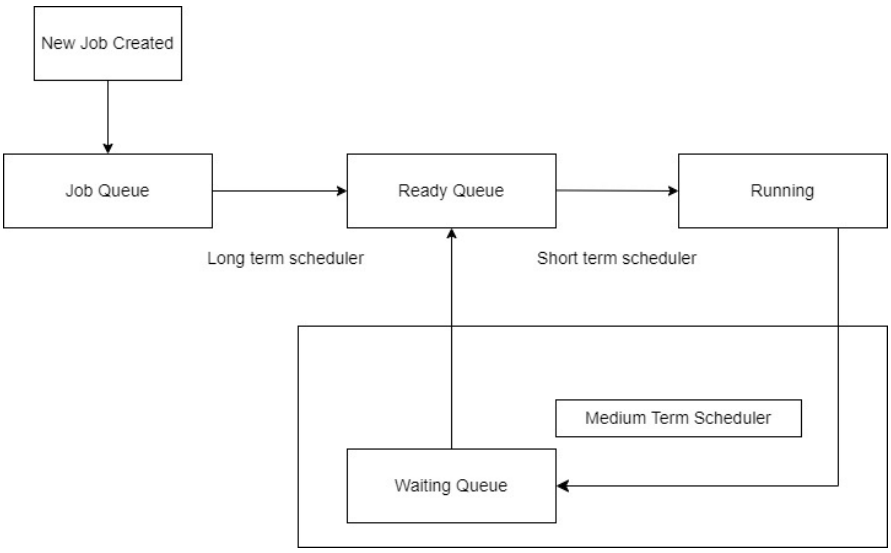


Figure 3.2: Types of Schedulers

Batch job scheduler is an example of long term scheduler whose task is to admitting print jobs. The short term scheduler can be a Round robin or priority scheduler allocating CPU to processes and Medium term scheduler will work to swapping out an inactive process to free memory. Table 3.1 shows brief comparison of Long Term Scheduler, Short Term Scheduler and Medium Term Scheduler.

3.3 Scheduling Criteria

There are various criteria for scheduling. Some criterion's are important in some system. For example, In server OS improving CPU utilization and system performance is objective. While in Time-sharing or PC OS user response is important. Many objectives must be considered in the design of a scheduling discipline. In particular, a scheduler should consider fairness, efficiency, response time, turnaround time, throughput.
There are also some goals that are desirable in all systems.

- Fairness: Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU. It should not happen that any process will wait very long time (indefinite postponement). Note that giving equivalent or equal time is not fair. Think of safety control and payroll at a nuclear plant.
- Policy Enforcement: The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the safety control processes must be given highest priority. It is always scheduled before other processes.

Table 3.1: Comparison of Schedulers

Sr. No.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued

- Efficiency: Scheduler should keep the system (or in particular CPU) busy always or most of the time. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second.

- Response Time: A scheduler should minimize the response time for interactive user. The response time is the time from the submission of a request until the first response is produced.

- Turnaround Time: The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O. It is important to reduce turn around time in batch system. A scheduler should minimize the time batch users must wait for an output.

- Throughput: Throughput is the number of processes that are completed per time unit. A scheduler should maximize the throughput.For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

- Waiting Time: Waiting time is total time process spent waiting in the ready queue.

It is desirable to *maximize Efficiency and throughput* and to *minimize turnaround time, waiting time, and response time*. Some of these goals are contradictory. In some cases, we may focus on one criteria more than other. For example, to guarantee that all users get good service, we may want to minimize response time but it may not give you efficiency.While if scheduler gives importance on efficiency, it may not be able to provide quick response to user.

3.4 Preemptive Vs Nonpreemptive Scheduling

The Scheduling algorithms(Ramamritham and Stankovic, 1994) can be divided into two categories with respect to how they deal with clock interrupts.

1. *Nonpreemptive Scheduling*
   A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.
   Following are some characteristics of nonpreemptive scheduling:

   (a) In nonpreemptive scheduling, a scheduler executes jobs in the following two situations:
      i. When a process switches from running state to the waiting state.
      ii. When a process terminates.

   (b) In nonpreemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.

   (c) In nonpreemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.

2. *Preemptive Scheduling*
   A scheduling discipline is preemptive if, scheduler is able to take CPU from process before it leaves CPU voluntary. In preemptive scheduling, processes can go from running to be temporarily suspended while in non preemptive scheduling processes run to completion method.

In preemptive scheduling, the operating system can interrupt the currently running process and allocate the CPU to another process according to a predefined priority scheme or set of rules.

3.5 Scheduling Algorithm

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms.

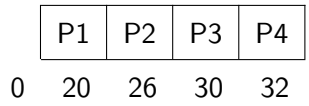
3.5.1 First-Come, First-Served Scheduling (FCFS)

Simplest CPU-scheduling algorithm is the first-come, first-served(FCFS) scheduling algorithm. We can take an anology, in railway reservation queue person who comes first will get his ticket first. In this algorithm, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.
Consider the following set of processes that arrive in order(P1,P2,P3,P4), with the length of the CPU burst given in milliseconds given in Table 3.2:

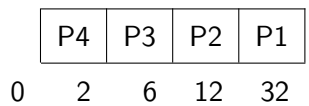
Table 3.2: FCFS Example

Process	CPU Burst Time
P1	20
P2	6
P3	4
P4	2

For this example processes arrived in the order P1,P2,P3,P4. Scheduler which uses FCFS allocate CPU is same order.
Following Gantt chart shows execution of processes:
The waiting time(WT) for processes will be:



$WT(P1) = 0$
$WT(P2) = 20$
$WT(P3) = 26$
$WT(P4) = 30$
$AverageWT = 0 + 20 + 26 + 30/4$
The turnaround time(TA) for processes will be:
$TA(P1) = 20$
$TA(P2) = 26$
$TA(P3) = 30$
$TA(P4) = 32$
$AverageTA = 20 + 26 + 30 + 32/4$
The benefit of this algorithm is that it is easy to understand and easy to implement. It is also fair in the same sense that process requested CPU first will get CPU first.
If same processes arrive in order P4,P3,P2,P1 than the waiting time for processes will be:



$WT(P1) = 12$

$WT(P2) = 6$
$WT(P3) = 2$
$WT(P4) = 0$
$AverageWT = 12 + 6 + 2 + 0/4$
The turnaround time(TA) for processes will be:
$TA(P1) = 32$
$TA(P2) = 12$
$TA(P3) = 6$
$TA(P4) = 2$
$AverageTA = 32 + 12 + 6 + 2/4$

The average waiting time under the FCFS is generally very long. It depends upon the arrival order of process.We can take an example, assume we have one CPU-bound process(large CPU burst) and many I/O-bound processes(small CPU burst). If the CPU-bound process arrives first. Scheduler will allocate CPU First to that process. It will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. The CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes will have to wait until the CPU-bound process completes its CPU burst. This effect is known as *convoy effect* in which all the other processes wait for the one big process to leave CPU. This effect results in lower CPU and device utilization than allowing shorter processes to complete execution. The FCFS scheduling algorithm is *nonpreemptive*. The FCFS algorithm is not suitable for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

Remark. FCFS scheduling is suitable for scenarios where fairness and simplicity are more important than optimizing resource utilization or response time. However, it may not be ideal for systems with diverse workload characteristics or real-time requirements, as it does not prioritize critical tasks or consider the urgency of processes.

3.5.2 Shortest-Job-First Scheduling(SJF)

A different approach to CPU scheduling is the shortest-job-first (SJF) or Shortest Process First(SPF) scheduling algorithm. This algorithm select shortest process for scheduling. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. This scheduling method can also be called the *shortest-next-CPU-burst algorithm*, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
As an example of SJF scheduling, consider the following set of processes given in Table 3.3:

For this example processes arrived in the order P1,P2,P3,P4. Scheduler which uses SJF allocate CPU to the shortest process. In this case P2, P3,P4 are having next cpu burst of 4. The scheduler pick P2 which comes before P3,P4. After P2, Scheduler choose next process in order P3, P4, P1.
Following Gantt chart shows execution of processes:

The waiting time(WT) for processes will be:

Table 3.3: SJF Example

Process	CPU Burst Time
P1	8
P2	4
P3	4
P4	4

P2	P3	P4	P1

0 4 8 12 20

$WT(P1) = 12$
$WT(P2) = 0$
$WT(P3) = 4$
$WT(P4) = 8$
$AverageWT = 12 + 0 + 4 + 8/4$
The turnaround time(TA) for processes will be:
$TA(P1) = 20$
$TA(P2) = 4$
$TA(P3) = 8$
$TA(P4) = 12$
$AverageTA = 20 + 4 + 8 + 12/4$
The SJF scheduling algorithm is optimal algorithm. It alway gives minimum average waiting time and turn around time for a given set of processes. By completing a short process before a long, we can decreases the overall waiting time.
The real difficulty with the SJF algorithm is knowing the length of the next CPU request. SJF scheduling is used frequently in long-term scheduling.For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. The user can give short time to get faster response, which may lead to poor performance. SJF algorithm cannot be implemented at the level of short-term CPU scheduling. Because there is no way to know the length of the next CPU burst.
One approach is to use approximate SJF scheduling. We may not know the length of the next CPU burst. But we can use previous CPU burst length to predict value of next CPU burst.Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the nth CPU burst and T_{n+1} is predicted value of n+1 burst then

$$T_{n+1} = at_n + (1 - a)T_n \tag{3.1}$$

Where a is constant and $0 <= a <= 1$ holds. This formula is exponential average formula. In which a controls weightage of historical and recent value of CPU burst. t_n is the recent value of CPU burst and T_n is our approximate CPU burst calculated on basis of past values of CPU burst.
If a = 0, then $T_{n+1} = T_n$ and recent value of CPU burst has no effect . if a = 1, then $T_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old

and irrelevant). More commonly, a $=1/2$, so recent history and past history are equally weighted.
We can take an example: If process last CPU burst is 2, a=0.5 and assuming last predicted value of Tn=0 then new estimated value of next CPU Burst is: $T_{n+1} = 0.5*2+(1-0.5)*0$ =1 in same way T_{n+2} can be calculated(if $t_{n+1} = 4$): $T_{n+2} = .5 * 4 + (1 - 0.5) * 1$ =2.5

Remark. SJF scheduling relies on accurate estimations of process burst times, which may not always be available. Additionally, it may suffer from starvation for longer processes if a continuous stream of short processes keeps arriving.

3.5.3 Preemptive Shortest-Job-First or Shortest Remaining Time First(SRTF)

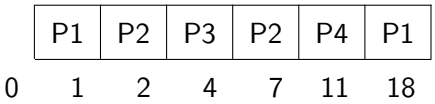
In a non-preemptive SJF algorithm, scheduler will allow the currently running process to finish its CPU burst,even if process with shorter CPU burst comes in ready queue. Whereas Preemptive SJF, will preempt the currently executing process if process with shorter burst request for execution. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling. We can take following example given in Table 3.4 to understand Preemptive SJF scheduling:

Table 3.4: SRTF Example

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

Preemptive SJF scheduler schedule in following way:

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.Process P3 arrive at time 2.The remaining time for process P1 (7 milliseconds) and P2 is (3 milliseconds) are larger than the time required by process P3 (2 milliseconds), so process P2 is preempted, and process P3 is scheduled. At time 3, process P4 arrived. The remaining time(1) for P3 is shortest among all. So it will complete its execution first, after that P2, P4, P1 completes their execution in order.



The waiting time(WT) for processes will be:
$WT(P1) = 11 - 1 - 0 = 10$
$WT(P2) = 4 - 1 - 1 = 2$
$WT(P3) = 2 - 2 = 0$
$WT(P4) = 7 - 3 = 4$
$AverageWT = 10 + 2 + 0 + 4/4$ The turnaround time(TA) for processes will be:
$TA(P1) = 18 - 0 = 18$
$TA(P2) = 7 - 1 = 6$
$TA(P3) = 4 - 2 = 2$
$TA(P4) = 11 - 3 = 8$
$AverageTA = 18 + 6 + 2 + 8/4$ While for same set of processes Nonpreemptive SJF scheduling would result in an average waiting time of $0 + 6 + 9 + 11/4$ milliseconds.

Remark. In interactive systems such as personal computers, servers, or real-time systems handling user requests, SRTF scheduling can lead to better responsiveness by quickly responding to short user interactions or requests. It may suffer from increased context switching overhead due to its preemptive nature, and it may not be suitable for systems with high levels of process arrival rate or where process burst times are unpredictable.

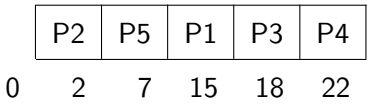
3.5.4 Priority Scheduling

In priority scheduling algorithm, scheduler will choose process with highest priority. Equal-priority processes are scheduled in FCFS order.
An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. We discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. Here, we assume that low numbers represent high priority.
Consider the following set of processes(Table 3.5), assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds: The

Table 3.5: Priority Scheduling Example

Process	CPU Burst Time	Priority
P1	8	3
P2	2	1
P3	3	4
P4	4	4
P5	5	2

scheduler choose Highest priority process P2 first, then choose processes in order P5, P1, P3, P4. The Gantt chart for this algorithm given as:



The waiting time(WT) for processes will be:
$WT(P1) = 7$
$WT(P2) = 0$
$WT(P3) = 15$
$WT(P4) = 18$
$WT(P5) = 2$
$AverageWT = 7 + 0 + 15 + 18 + /5$ The turnaround time(TA) for processes will be:
$TA(P1) = 15$
$TA(P2) = 2$
$TA(P3) = 18$
$TA(P4) = 22$
$TA(P5) = 7$
$AverageTA = 15 + 2 + 18 + 22 + 7/5$

Few characteristics of Priority Scheduling:

1. Priorities can be defined either internally or externally.
   Internally defined priorities are defined by Operating system. For example OS can calculate priority on basis of time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
   External priorities are set by criteria outside the OS. It depends upon the importance of the process, the type of user initiated process.

2. Priority scheduling can be either pre-emptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
   A pre-emptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
   A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

3. A major problem with priority scheduling algorithms is *indefinite blocking, or starvation*. A process that is ready to run but waiting for the CPU can be considered blocked.

Remark. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Generally priority scheduling algorithm are used with combination of round-robin scheduling. We can define priority classes based on priority. If there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If processes belongs to lower priority classes, then they may be starved.because it may happen that high priority process comes one after another and low priority process is waiting long for processor

[Aging] In aging priorities of jobs increase the longer they wait. Under this scheme scheduler increase priority of a low-priority job with time. They moved up in high priority queue, where they have better chance to avail CPU.

3.5.5 Round Robin Scheduling

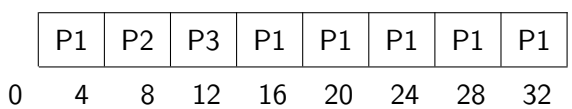
In Round Robin(RR) Scheduling,Each process is provided a fix time to execute called quantum. Once a process is executed for given time period. Process is preempted and other process executes for given time period. Context switching is used to save states of preempted processes.
RR scheduling involves extensive overhead, especially with a small time unit. RR scheduling provides balanced throughput between FCFS and SJF. In RR scheduling,shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJF. RR scheduling provides good average response time. In this,waiting time is dependent on number of processes, and not average process length. System using RR may suffer from high waiting times and may not be able to met deadlines . Starvation can never occur, since no priority is given. It is similar to FCFS, order of time unit allocation is based upon process arrival time. If size of time quantum is too large than it may behave like FCFS. That is the reason it may called as *Preemptive FCFS*.
We can take following example to understand Round Robin Scheduling(Table 3.6): For

Table 3.6: Round Robin Scheduling Example

Process	Burst Time
P1	24
P2	4
P3	4

above set of processes, RR Scheduler choose P1 first and allocate it CPU for time quantum =4. If process voluntary leave CPU before time quantum then it will allocate CPU to other process. In this case P1 does not leave CPU voluntary, that is the reason CPU will preempt P1 and allocate CPU to next process P2 for 4 ms. After P2 completes scheduler picks next process in ready queue which is P3. After completion of P3 scheduler picks P1 and so on.
The waiting time(WT) for processes will be:
$WT(P1) = 8$
$WT(P2) = 4$
$WT(P3) = 8$
$AverageWT = 8 + 4 + 0/3$
The turnaround time(TA) for processes will be:
$TA(P1) = 32$
$TA(P2) = 8$
$TA(P3) = 12$
$AverageTA = 32 + 8 + 12/3$

Gantt Chart for time quantum=4 :

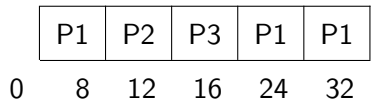


Remark. The selection of time quantum(length of time quantum) is important for RR Scheduling. If we choose small time quantum, we will get better response but may suffers

from large number of context switch. If we take large time quantum, our scheduling behaves like FCFS which is having problem of higher waiting time.

What will happen if we increase time quantum for above example. The waiting time(WT) for processes will be:
$WT(P1) = 8$
$WT(P2) = 8$
$WT(P3) = 12$
$AverageWT = (8 + 8 + 12)/3$ The turnaround time(TA) for processes will be:
$TA(P1) = 32$
$TA(P2) = 12$
$TA(P3) = 16$
$AverageTA = 32 + 12 + 16/3$

Gantt chart for time quantum=8 :



The characteristics of RR Scheduling:

1. The main advantage of round robin algorithm over FCFS is that it is *starvation free*. Every process will be executed by CPU for fixed interval of time (which is set as time slice). So in this way no process left waiting for its turn to be executed by the CPU .
2. RR algorithm is simple and easy to implement .
3. The length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

   Remark. Round Robin scheduling is versatile and widely used in various computing environments due to its simplicity, fairness, and suitability for a wide range of workloads.

3.5.6 Multilevel Queue Scheduling

In this scheduling processes grouped like foreground(interactive process) ,background(batch process),etc.These two types of processes have different response-time requirements. They should be scheduled differently. A multilevel queue scheduling algorithm partitions the ready queue into multiple queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue can have its own scheduling algorithms.For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. Figure 3.3 shows queues in multilevel queue scheduling.

Remark. In this algorithm jobs cannot switch from queue to queue. Once they are assigned a queue, they remain in same queue until they finish.

An example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

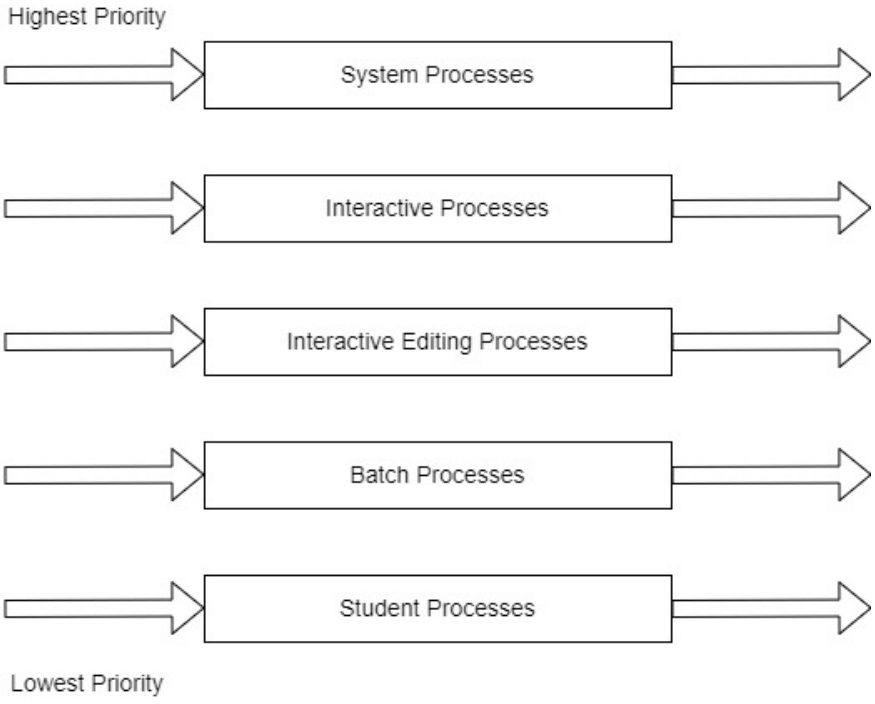


Figure 3.3: Multilevel Queue Scheduling

Upper queue has greater priority over its lower-priority queues. No process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Another option is to give fixed time-slice to the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis. Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. Processes do not able to change their queues(foreground or background nature). This setup has the advantage of low scheduling overhead, but it is inflexible.

[Multilevel Feedback Scheduling] It is one special case of Multilevel queue scheduling where processes can change their queues. The multilevel feedback-queue scheduling algorithm, allows a process to move between queues. It may separate processes based on length of their CPU bursts.

If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. The technique of aging can also be implemented in this scheduling. Process waiting in a lower-priority queue from long time may be moved to a higher-priority queue. We can summarize schedul-

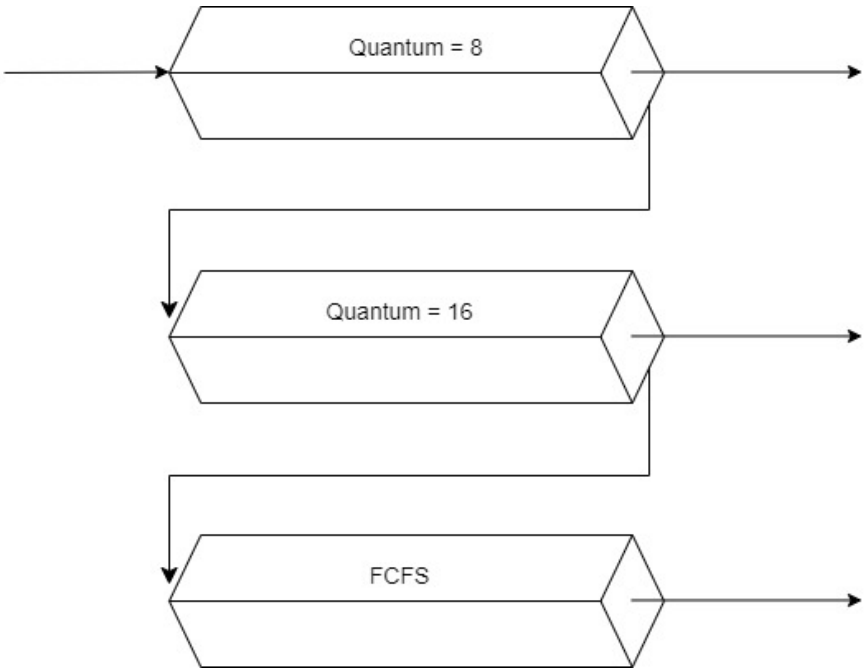


Figure 3.4: Multilevel Feedback Queue Scheduling

ing goals and algorithms suitable for different systems:

Table 3.7: OS Goals and Scheduling Algorithms

OS Types	Goal	Scheduling Algorithm
Batch	Throughput	FCFS
	Turn around time	Shortest Job First
	CPU Utilization	Shortest Remaining Time First
Interactive	Response time	Round Robin
	Proportionality	Priority Scheduling
Realtime	Meeting deadline	Guaranteed Scheduling
	Predictability	Rate Monotonic Scheduling

3.6 Multiprocessor Scheduling Algorithm

Multiprocessor systems are increasingly commonplace, and have found their way into desktop machines, laptops, and even mobile devices. In this system, Multiple CPUs are available, load sharing is possible. Multiprocessor Scheduling can be divided in following two types:

3.6.1 Symmetric MultiProcessor SMP

The most common multiprocessor system in use today is the Symmetric MultiProcessor (SMP). All of the CPUs in an SMP system are identical. All processes put into a common ready queue, or each processor may have its own private queue of ready processes. Whenever a CPU needs a process to run, it takes the next task from the ready list. The scheduling queue must be accessed in a critical section. Busy waiting is usually used.

Some consideration for SMP scheduling:

1. Selection of the next task is not as important. With multiple CPUs, it is not as likely that a short task will wait for a long task to complete.
2. Any task can run on any CPU thereby allowing load balancing.
3. Tasks should stay with a single CPU to take advantage of cache loading.
4. It can use *Gang scheduling*.

[Gang Scheduling] It schedule all of the threads of a process together.

In an SMP, this isn't much of a problem since any CPU can execute any thread. In systems with distributed memory (each CPU has its own memory and cannot access the RAM of another CPU), load sharing is a major consideration. This is often more of an application concern than an OS concern.

3.6.2 Asymmetric multiprocessing

In this approach,*master server* handles all scheduling decisions, I/O processing, and other system activities. The other processors execute only user code. This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing. In an asymmetric multiprocessing system, not all CPUs are treated equally; for example, a system might only allow (either at the hardware or operating system level) one CPU to execute operating system code or might only allow one CPU to perform I/O operations.

3.7 Scheduling Algorithm Evaluation

How can we select a CPU scheduling algorithm for a particular system. We can evaluate scheduling algorithms by establishing criteria. The criteria may be like Maximizing CPU utilization, Guaranteed response time, Maximizing throughput, etc or combination of this. There are many scheduling algorithms. We can evaluate CPU algorithm based on criteria by following methods:

1. Deterministic modeling : We can takes a particular predetermined workload and defines the performance of each algorithm for the workload. For example Consider the FCFS, SJF, and RR(time quantum=10) scheduling algorithms for set of processes. which will give the min avg. WT? We have done deterministic modeling by using Gantt chart in scheduling algorithm examples. Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. But, it requires exact numbers for input, and its answers apply only to those cases.

2. Queuing models : It uses probabilistic approach to model arrival of processes, and CPU and I/O bursts. It Computes average throughput, utilization, waiting time, etc.In it, computer system described as network of servers, each with queue of waiting processes. This area of study is called queueing-network analysis. Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. The classes of algorithms and distributions that can be handled with this model are limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

3. Simulations : Simulations are more accurate than queuing model. They use Programmed model of computer system. They Gather statistics indicating algorithm performance. Simulation uses Random number generator according to probabilities Distributions defined mathematically or empirically to provide input. Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also requires more computer time. Finally, the design, coding, and debugging of the simulator can be a major task.

4. Implementation : Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The major difficulty with this approach is the high cost. Another difficulty is that the environment in which the algorithm is used will change.

Conclusion: In conclusion, the chapter provides a thorough examination of the various scheduling algorithms used in operating systems to manage process execution. Each algorithm has its own set of advantages and limitations, making them suitable for different types of systems and workloads. For instance, FCFS is simple and fair but can lead to long waiting times, especially in the presence of CPU-bound processes. SJF minimizes waiting time but requires precise knowledge of job lengths, which is often impractical. Priority Scheduling ensures critical tasks are prioritized but can cause starvation for low-priority processes. Round Robin Scheduling, suitable for time-sharing systems, balances the needs of all processes but can lead to high context-switching overhead. Multiprocessor scheduling introduces additional complexity but is essential for optimizing performance in systems with multiple CPUs.

3.8 Exercise

1. Explain the concept of process scheduling in an operating system.

2. What are the primary goals of a process scheduler, and how do they impact system performance?

3. How does the context switching overhead factor into scheduling decisions?

4. How can operating systems optimize scheduling to minimize context switches?

5. Describe the different types of scheduling queues used in operating systems (e.g., ready queue, waiting queue).

6. Explain the role of a dispatcher in the scheduling process.

7. Define preemptive and non-preemptive scheduling. Provide examples of scheduling algorithms that fall under each category.

8. Define preemptive and non-preemptive scheduling. Provide examples of scheduling algorithms that fall under each category.

9. Discuss the advantages and disadvantages of preemptive and non-preemptive scheduling.

10. Explain the concepts of starvation and convoying in the context of scheduling algorithms. Provide examples of scheduling scenarios where these issues might arise. How can we design scheduling algorithms to mitigate starvation and convoying?

11. Describe the concept of a Multilevel Feedback Queue (MLFQ) scheduling algorithm.

12. The concept of aging can be used to improve the responsiveness of scheduling algorithms. Explain how aging works and how it can be implemented in scheduling decisions.

13. Consider three process, all arriving at time zero, with total execution time of 10, 20 and 30 units respectively. Calculate Average Waiting Time for:

    (a) FCFS

    (b) SJF

14. Consider the set of 5 processes whose arrival time and burst time are given in Table 3.8. Calculate Average waiting time and turn around time for following scheduling algorithm:

    (a) FCFS

    (b) SJF

    (c) SRTF

    (d) Round Robin with time quantum=2

Table 3.8: Scheduling Problem

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

3.9 Multiple Choice Questions

1. The ____ is the number of processes completed per unit time.
   (a) Capacity
   (b) Output
   (c) Throughput
   (d) Efficiency

2. Long-term scheduler is____ ?
   (a) It selects which process has to be brought into the ready queue
   (b) It selects which process has to be executed next and allocates CPU
   (c) It selects which process to remove from memory by swapping
   (d) None of the above

3. A process terminates when_____:
   (a) It is removed the job queue
   (b) Its process control block is de-allocated
   (c) It is removed from waiting queue
   (d) None of the above

4. Medium-term scheduler is _____ ?
   (a) It selects which process has to be brought into the ready queue
   (b) It selects which process to remove from memory by swapping
   (c) It selects which process has to be executed next and allocates CPU
   (d) None of the mentioned

5. Short-term scheduler selects _____ ?
   (a) Which process has to be executed next and allocates CPU
   (b) Which process to remove from memory by swapping
   (c) Which process has to be brought into the ready queue
   (d) None of the mentioned

6. The interval from the time of submission of a process to the time of completion is termed as _____
   (a) waiting time
   (b) turnaround time
   (c) response time
   (d) throughput

7. Which scheduling algorithm allocates the CPU first to the process that requests the CPU first?
   (a) FCFS(first-come, first-served scheduling)

(b) Shortest job scheduling

(c) Priority scheduling

(d) None of the mentioned

8. In priority scheduling algorithm, CPU is allocated to _____

(a) Process with highest priority

(b) Process with lowest priority

(c) Process with shortest burst time

(d) None of the mentioned

9. If the time-slice used in the round-robin scheduling policy is more than the maximum time required to execute any process, then the policy will

(a) degenerate to shortest job first

(b) degenerate to priority scheduling

(c) degenerate to first come first serve

(d) none of the above

10. Which of the following scheduling algorithms must be non-preemptive?

(a) Round Robin

(b) Priority algorithm

(c) FCFS

(d) SJF

11. The _____ scheduling algorithm is designed especially for time sharing systems.

(a) First Come First Served

(b) Shortest Remaining Time First

(c) Shortest Job First

(d) Round Robin

12. Assume there are 10 processes in the system running or ready to run. If round robin scheduling algorithm with a time quantum of 2 ms is used, then a waiting process will not wait more than _____?

(a) 12 ms

(b) 15 ms

(c) 18 ms

(d) 20 ms

13. Consider three CPU intensive processes, which require 10, 20, 30 units and arrive at times 0,2,6 respectively. How many context switches are needed if shortest remaining time first is implemented? Context switch at 0 is included but context switch at end is ignored

(a) 1

(b) 2

(c) 3

(d) 4

14. There are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below. What will be the average waiting time in Shortest Job First Scheduling:

    (a) 4.2

    (b) 4.9

    (c) 5.4

    (d) 5.8

15. There are five jobs named as P1, P2, P3, P4, P5 and P6. Their priority, arrival time and burst time are given in the table below. What will be the turnaround time for process P5 in Non-Preemptive Priority Scheduling?

    (a) 9

    (b) 16

    (c) 21

    (d) 30

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

DeepScience
Open Access Books

Chapter 4

Inter Process Communication

Abstract: This chapter explores various software solutions for achieving synchronization and mutual exclusion in operating systems. It delves into classic algorithms such as Peterson's solution and semaphores, explaining their mechanisms and how they prevent race conditions. The chapter also addresses the producer-consumer problem, highlighting the need for synchronization in concurrent process execution and providing practical examples to illustrate key concepts.

Keywords: Synchronization, Mutual Exclusion, Peterson's Solution, Semaphores, Producer-Consumer Problem, Race Condition, Concurrent Processes.

4.1 Introduction

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is *independent* if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is *cooperating* if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. Figure 4.1 shows Inter Process Communication Mechanism for processes.

Inter-process communication (IPC) is the mechanism for sharing data between multiple processes.

Figure 4.1: Inter Process Communication Mechanism

4.1.1 Benefits

There are several reasons for providing an environment that allows process: cooperation:

- Information sharing: Since several users may be interested in the same piece of information (for instance, a shared file). For example web servers use IPC to share web documents and media with users through a web browser.

- Computation speedup: If we want a particular task to run faster, we must break it into subtasks, then execute them in parallel. We will get speedup only if the computer has multiple processing elements (such as CPUs or I/O channels). In client server environment like when we access google, one server process can not handle all request. These types of websites uses multiple servers that communicate with one another using IPC to process user requests.

- Modularity: We can develop large application with large number of small task. IPC helps coordination among processes to accomplish large task.

- Convenience: Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

4.1.2 Categories

We can categorize IPC into two classes:

1. Communication mechanism: Communication mechanism are used to communicate with other process.There are two fundamental models of inter-process communication:

   (a) Shared memory

   (b) Message passing

   In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing model, communication takes place by means of messages exchanged between the cooperating processes. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for inter computer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time consuming task of kernel intervention. In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. In the remainder of this section, we explore each of these IPC models in more detail.

2. Synchronization mechanism: Synchronization mechanism are used to provide synchronization between processes. There are various mechanism like Semaphores, Monitors, Barriers, etc provides synchronization.

4.1.3 Producer Consumer Problem (Bounded Buffer Problem)

The producer-consumer problem raise the need for synchronization in systems where many processes share a resource. In this problem, two processes share a fixed-size buffer. One process(Producer) produces information and puts it in the buffer, while the other process(Consumer) consumes information from the buffer. Producer and Consumer both work

concurrently.These processes can access the buffer concurrently. Concurrent access to buffer leads to nondeterministic results.
What happens if the producer tries to put an item into a full buffer?
What happens if the consumer tries to take an item from an empty buffer?
In order to synchronize these processes, we will block the producer when the buffer is full, and we will block the consumer when the buffer is empty. So the two processes, Producer and Consumer, should work as follows:

```
BufferSize = 3;
count = 0;

Producer()
{
        int widget;
        WHILE (true) {              // loop forever
                make_new(item); //create a new item to put in the buffer
                IF(count==BufferSize)
                Sleep();               // if the buffer is full, sleep
                put_item(item);        // put the item in the buffer
                count = count + 1;     // increment count of items
                IF (count==1)
                Wakeup(Consumer);      // if the buffer was previously
                    empty, wake
                  }                      //  the consumer
}

Consumer()
{
        int item;
        WHILE(true) {             // loop forever
                IF(count==0)
                Sleep();              // if the buffer is empty, sleep
                remove_item(item);    // take an item from the buffer
                count = count + 1;    // decrement count of items
                IF(count==N-1)
                Wakeup(Producer);     // if buffer was previously full,
                             //wake the producer

                Consume_item(item);  // consume the item
                }
}
```

4.1.4 Race Condition

Concurrency occurs when two or more separate execution flows are able to run simultaneously. Examples of independent execution flows include threads, processes, and tasks. Concurrent execution of multiple flows of execution is an essential part of a modern computing environment. We can show that the value of counter may be incorrect as follows. Note that the statement "counter++" may be implemented in machine language (on a typical machine) as

```
S1: R1 <- counter //Load value of Counter in R1
S2: R1 <- R1 + 1
S3: counter <- R1 //Store value of R1 in Counter
```

where R1 is a local CPU register. Similarly, the statement "counter–" is implemented as follows:

```
S4: R2 <- counter //Load value of Counter in R2
S5: R2 <- R2 - 1
S6: counter <- R2 //Store value of R2 in Counter
```

where again R2 is a local CPU register. The concurrent execution of "counter++" and "counter–" is implemented by sequential execution of above lower-level statements. These statements may be executed in any order. The order for correct results is (S1,S2,S3,S4,S5,S6). We can take example, Suppose We have 5 items in buffer and value of counter=5. Producer produce an item and increment counter. Consumer consumes an item and decrement counter. The value of counter will be 5, if all statements execute in correct order. We are considering one possible order which may produce unexpected results:

```
S1: R1 <-counter // R1=5,counter=5
S2: R1 <- R1 + 1  // R1=6
S4: R2 <- counter // R2=5,counter=5
S5: R2 <- R2 - 1  // R2=4
S3: counter <- R1 // counter=6
S6: counter <- R2 // counter=4
```

We will get value of counter=6, which is incorrect. This situation is *Race Condition* which may be defined as:

[Race Conditions] A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. Race condition is a software defect and should be avoided. Race conditions result from runtime environments, including operating systems, that must control access to shared resources, especially through process scheduling.

Race condition is arise because we allowed both processes to manipulate the variable counter concurrently. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way. Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Clearly, we want the resulting changes not to interfere with one another.

4.2 The Critical-Section Problem

Race condition may arise in many other situations involving shared memory, shared files, and shared everything else. We can eliminate it by prohibiting more than one process from reading and writing the shared data at the same time. This solution is *mutual exclusion* (Singhal, 2001). It ensures that if one process is using a shared variable or file, the other processes will not be able to access the same

[Critical Section] The part of the program where the shared memory is accessed is called the critical region or critical section. If we could ensure that no two processes were ever in their critical regions at the same time, we could avoid races.

Consider a system consisting of n processes P1,P2...Pn. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when

one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The critical-section problem is designing a protocol for process synchronization. Each process must request permission to enter its critical section. The section of code implementing this request is the *entry section*. The critical section may be followed by an *exit section*. The remaining code is the remainder section. The general structure of a typical process P,can be shown as:

```
do{
        entry section

        critical section

        exit section

        remainder section

} while (TRUE);
```

4.2.1 Requirement for Solving Critical Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion: If process P; is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.(This is also known as the NO DEADLOCK requirement.)

3. Bounded waiting: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.(This is also called the NO STARVATION or the FAIRNESS requirement.)

4. We can make no assumption concerning the relative speed of the n processes.

4.2.2 Solutions

Solutions to the critical section problem are of two general types:

1. Hardware Solution: Solutions depending on special hardware facilities.

2. Software Solution: Solutions that are strictly software based. Only characteristic of the hardware they rely on is that if two processes attempt to store a value in the same memory cell, then the hardware will guarantee that the final value will be the same as that written by one of the two, though nothing is guaranteed regarding the order.

4.3 Hardware Solutions

4.3.1 Disabling Interrupts

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts. If interrupts are disabled CPU will not be switched to another process. After disabling interrupt,Process can access shared memory with ensuring that no other process will be able to access it.

This method gives user processes the power to turn off interrupts, which may not re enable it again. That is the reason this solution is not used for user processes. This approach is generally used in kernel code.

```
InterruptDisable()
// critical section
InterruptEnable()
```

Disadvantages of Disabling Interrupts:

1. It does not work for user-mode programs. So the Unix print spooler, which is a user-mode program would need another solution.
2. We do not want to block interrupts for too long or the system will seem unresponsive.
3. Disabling interrupts is insufficient if the system has several processors. The main line can be executing on both processors simultaneously so interrupts are not involved. One processor cannot block interrupts on the other.

4.3.2 TSL Instructions

The test-and-set instruction(TSL) is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
It performs following two operations atomically:

1. It reads the contents of the memory word **lock** into register.
2. It stores a nonzero value at the memory address **lock**.

No other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus. That is the reason other process will not be able to access memory. Typically, the value 1(true) is written to the memory location. If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done. One way to implement mutual exclusion by using test-and-set is:

```
boolean lock = false
function Critical(){
                while TestAndSet(lock)
                ; // skip check until lock is acquired
                critical section // only one process can be in this
                                          //section at a time
                    lock = false // release lock when finished with the
                        critical section
        }
```

4.4 Software Solutions

4.4.1 Strict Alternation

It use integer variable turn. Variable **turn** keeps track of whose turn it is to enter the Critical Section(CS). If turn=0, then process 0 will enter in critical section. In case turn=1 process 1 will enter in critical section.after leaving critical section process alter value of turn. If process 0 completes its CS, it will change turn to 1.After that process 1 can enter in its CS.

```
        while (TRUE) {
                while (turn != 0)
                critical_section{};
                        turn = 1;
                        noncritical_section();
                        /* loop */;
                }
                PROCESS 0

                while (TRUE) {
                        while (turn != 1)
                        critical _region();
                        turn = 0;
                        noncritical _region();
                        /* loop */
                }
                PROCESS 1
```

We can again explain how mutual exclusion is achieved with this method:

1. Initially, process 0 inspects turn, finds it to be 0, and enters its CS.
2. process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
3. When process 0 leaves the CS, it sets turn to 1, to allow process 1 to enter its CS.
4. Suppose that process 1 finishes its CS quickly, so both processes are in their non CS (with turn set to 0).

This solution ensures Mutual exclusion but do not fulfill Progress. For Example: If process 0 finishes its non CS and goes back to the top of its loop. Process 0 executes its whole loop quickly, exiting its CS and setting turn to 1.
At this point turn is 1 and both processes are executing in their non CS.process 0 finishes its non CS and goes back to the top of its loop.
It is not permitted to enter its CS, because turn is 1.Even process 1 is not in CS, process 0 is waiting for process 1 to set turn to 0.

4.4.2 Peterson's Solution

Peterson's algorithm (Peterson's solution) is used to achieve mutual exclusion that allows two processes to share a single-use resource without conflict. It was formulated by Gary L. Peterson in 1981.His original solution worked with only two processes. The algorithm can be generalized for more than two processes. The algorithm uses two variables, flag and turn. A flag[n] value of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.

The pseudo-code for Peterson's solution is given as:

```
bool flag[0]   = false;
bool flag[1]   = false;
int turn;
```

Variables

```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)
{
        // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

Process P0

```
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0)
{
        // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

Process P1

The algorithm does satisfy the three essential criteria(mutual exclusion, progress, and bounded waiting) to solve the critical section problem.

4.4.3 Semaphores

Semaphores are useful in preventing race conditions. A semaphore is a variable or abstract data type that is used for controlling access to a common resource. It is a synchronization tool to access shared memory and resources. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal (). The value of the semaphore S is the number of units of the resource that are currently available Semaphores which allow synchronization for more than one Resources are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

```
Signal(semaphore S){
        S = S + 1; // Release Resource
}
Wait(semaphore S){
        while S <= 0 //If Resource is not Available Wait
        ; // no-op
        S=S-1;
}
```

Wait and signal pseudo codes for Binary Semaphores We can achieve mutual exclusion with semaphore in following way:

```
do
{
```

```
                wait(s);
                // critical section
                signal(s);
                // remainder section
        } while
```

Disadvantages

In this implementation, a process wanting to enter its critical section, has to get binary semaphore. Semaphore is available after signals is done. For example, we have semaphore s, and two processes, P1 and P2 that want to enter their critical sections at the same time. P1 first calls wait(s). The value of s is decremented to 0 and P1 enters its critical section. While P1 is in its critical section, P2 calls wait(s), but because the value of s is zero, it must wait until P1 finishes its critical section and executes signal(s). When P1 calls signal, the value of s is incremented to 1, and P2 can then proceed to execute in its critical section (after decrementing the semaphore again). Mutual exclusion is achieved because only one process can be in its critical section at any time.

[Spinlock] Processes waiting on a semaphore must constantly check to see if the semaphore is not zero. This continual looping is clearly a problem in a real multiprogramming system (where often a single CPU is shared among multiple processes). This is called *busy waiting* and it wastes CPU cycles. This type of semaphore is called a spinlock.

Second problem with semaphore is possibility of incorrect programming. The correct use of semaphore is:

```
        do
        {
                wait(s);
                // critical section
                signal(s);
                // remainder section
        } while
```

But if programmer changes order of wait and signal like:

```
        do
        {
                signal(s);
                // critical section
                wait(s);
                // remainder section
        } while
```

This code will not ensure critical section and may produce incorrect result.

To eliminate busy waiting, we can modify the definition of the wait () and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

```
wait(Semaphore s){
        s=s-1;
        if (s<0) {
                // add process to queue
                block();
        }
}

signal(Semaphore s){
        s=s+1;
        if (s<=0) {
                // remove process p from queue
                wakeup(p);
        }
}
```

4.4.4 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

A monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met. A monitor consists of a mutex (lock) object and condition variables. A condition variable is basically a container of threads that are waiting on a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Another definition of monitor is a *thread-safe class, object, or module* that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor").

```
Sample Monitor Code
==================

        name : monitor
        some local declarations
        initialize local data
        procedure name(arguments)
        do some work
        .
        .
```

```
                .
                other procedures
```

Benefits of Monitors

1. Monitors are A higher level synchronization mechanism provided in various languages.
2. They are easier to use, provides better abstraction, better encapsulation in comparison of semaphores/locks.

4.5 Classical IPC Problems

4.5.1 Readers and Writers Problem

The Readers and Writers problem is useful for modeling processes which are competing for a limited shared resource. A practical example of a Readers and Writers problem is a railway reservation system consisting of a huge data base with many processes that read and write the data. Reading information from the data base will not cause a problem since no data is changed. The problem lies in writing information to the data base. If no constraints are put on access to the data base, data may change at any moment. By the time a reading process displays the result of a request for information to the user, the actual data in the data base may have changed. What if, for instance, a process reads the number of available seats in a train, finds a value of one, and reports it to the customer. Before the customer has a chance to make their reservation, another process makes a reservation for another customer, changing the number of available seats to zero.

We can solve this problem by using semaphores:

```
                semaphore mutex = 1;        // Controls access to the
                    reader count
                semaphore db = 1;           // Controls access to the
                    database
                int reader_count;           // The number of reading
                    processes                                   //accessing
                     the data

    Reader()
                {
                        while (TRUE) {         // loop forever
                                wait(&mutex);     // gain access to
                                    reader_count
                                reader_count = reader_count + 1;
                                  // increment the reader_count
                                if (reader_count == 1)
                                wait(&db);
                            // if this is the first process to read the
                                   // database, a down on db is executed
                            // to prevent access to the database
                            // by a writing process
                                signal(&mutex);
                            // allow other processes to access
                                reader_count
                                read_db();  // read the database
                                wait(&mutex);    // gain access to
                                    reader_count
                                reader_count = reader_count - 1;    //
                                    decrement reader_count
                                if (reader_count == 0)
```

```
                        signal(&db);
            // if there are no more processes reading from the
                        // database, allow writing process to
                            access the data
                        signal(&mutex);
// allow other processes to access reader_countuse_data();
                        // use the data read from the database (
                            non-critical)
                    }

                    Writer()
                    {
                        while (TRUE) {                    //
                            loop forever
                            create_data();       // create
                                data to enter into
                                                          //
                                database (non-critical)
                            wait(&db);          // gain access
                                 to the database
                            write_db();        // write
                                information to the database
                            signal(&db);       // release
                                exclusive access to the
                                database
                        }
```

4.5.2 Dining Philosopher Problem

Dining Philosophers. There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more.

Figure 4.2: Dining Philosopher Problem

In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.

Thus, each philosopher is represented by the following pseudo code:

```
                    procedure philosopher(i)
                    {
                        while TRUE do
                        {
                            THINKING;
                            take_chopsticks(i);
                            EATING;
                            drop_chopsticks(i);
                        }
                    }
```

A philosopher may THINK indefinitely. Every philosopher who EATs will eventually finish. Philosophers may PICKUP and PUTDOWN their chopsticks in either order, or non deterministically, but these are atomic actions, and, of course, two philosophers cannot use a single CHOPSTICK at the same time.

The problem is to design a protocol to satisfy the liveliness condition: any philosopher who tries to EAT, eventually does.

Solutions: There are many solutions for this problem. This book is considering one solution given in *"Modern operating System"*. Philosopher can take chopstick before eating. While for taking chopstick philosopher needs to check whether he is hungry and both philosopher seated on his left and right side are not eating. The test function performs this task. If philosopher succeeds then he can start eating. After completion of eating, he drops his both sticks.

```
system DINING_PHILOSOPHERS

VAR
me:    semaphore, initially 1;
                          /* for mutual
    exclusion */
s[5]:  semaphore s[5], initially 0;
                    /* for synchronization
    */
pflag[5]: {THINK, HUNGRY, EAT}, initially
     THINK;  /* philosopher flag */

procedure take_chopsticks(i)
{
         wait(me);                /*
             critical section */
         pflag[i] := HUNGRY;
         test(i);         //test
         signal(me);                  /*
             end critical section */
         wait(s[i])               /* Eat if
              enabled */
}

void test(i)             /* Let phil[i]
    eat, if waiting */
{
         if ( pflag[i] == HUNGRY
         && pflag[i-1] != EAT
         && pflag[i+1] != EAT)
         then
         {
                  pflag[i] := EAT;
                  signal(s[i])
         }
}

void drop_chopsticks(int i)
{
         wait(me);                 /*
             critical section */
         test(i-1);                /* Let
             phil. on left eat if possible
              */
         test(i+1);                /* Let
             phil. on rght eat if possible
```

```
                                        */
                                    signal(me);                    /*
                                        up critical section */
                                }
```

Conclusion: In conclusion, synchronization mechanisms are crucial for ensuring the correct and efficient execution of concurrent processes in operating systems. Peterson's solution, while effective for two processes, can be extended for multiple processes, demonstrating the flexibility of algorithmic approaches to mutual exclusion. Semaphores, both binary and counting, offer robust solutions for managing access to shared resources. The producer-consumer problem underscores the necessity of synchronization to prevent nondeterministic results and maintain process integrity. Understanding and implementing these solutions is fundamental for developing reliable and efficient operating systems.

4.6 Exercise

1. What is IPC? Explain the use of Semaphore to solve Producer-Consumer problem.
2. What is Monitor in IPC? Solve the Bounded buffer problem using Monitor.
3. Describe the three main benefits of using an operating system in a computer system.
4. Briefly differentiate between the two main categories of operating systems based on user interaction.
5. Briefly explain the dining philosopher's problem?
6. Describe a possible solution to prevent deadlock in the dining philosopher's problem. List the three essential requirements for solving the critical-section problem effectively. Briefly describe two different approaches to achieve process synchronization for the critical-section problem.
7. Explain how disabling interrupts can be used for critical section implementation.
8. Choose one software-based solution (e.g., Strict Alternation, Peterson's Solution, Semaphores, Monitors) and explain its core principles for achieving process synchronization.
9. Compare and contrast the use of semaphores and monitors for process synchronization. Highlight the advantages and disadvantages of each approach.

4.7 Multiple Choice Questions

1. Which one is a synchronization tool of the following?
   (a) Pipe
   (b) Thread
   (c) Socket
   (d) Semaphore

2. The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows:

```
void enter_CS(X)
{
while test-and-set(X) ;
}
void leave_CS(X)
{
X = 0;
}
```

In the above solution, X is a memory location associated with the CS and is initialized to 0.

Now consider the following statements:

I. The above solution to CS problem is deadlock-free

II. The solution is starvation free.

III. The processes enter CS in FIFO order.

IV. More than one process can enter CS at the same time.

Which of the above statements is TRUE?

(a) I only
(b) I and II
(c) II and III
(d) IV only

3. Semaphore is a/an _____ to solve the critical section problem.

(a) integer variable
(b) special program for a system
(c) special H/W
(d) none of the mentioned

4. TestAndSet(TSL) instruction is executed _____

(a) periodically
(b) after every process
(c) atomically
(d) none of above

5. What are the two types of Semaphore?

(a) Digital Semaphores and Binary Semaphores
(b) Analog Semaphores and Octal Semaphores
(c) Counting Semaphores and Binary Semaphores
(d) Critical Semaphores and System Semaphores

6. What are the two atomic operations allowed on semaphores?

   (a) wait and stop
   (b) stop and hold
   (c) wait and signal
   (d) none of the mentioned

7. What are the requirements for the solution to critical section problem?

   (a) Mutual Exclusion
   (b) Progress
   (c) Bounded Waiting
   (d) All of Above

8. A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?

   (a) -2
   (b) -1
   (c) 1
   (d) 2

9. What are Spinlocks?

   (a) semaphores that work better on multiprocessor systems
   (b) semaphores that avoid time wastage in context switches
   (c) semaphores in which CPU cycles wasting in getting locks over critical sections of programs
   (d) All of the mentioned

10. If the semaphore value is negative _____

    (a) its magnitude is the number of processes waiting on that semaphore
    (b) it is invalid
    (c) no operation can be further performed on it until the signal operation is performed on it
    (d) none of the mentioned

11. The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0, S2 = 0.

```
Process P_O
while(true)
{
wait (S0);
print '0';
signal (S1);
signal (S2);
}
Process P_1
wait (S1);
signal (S0);
Process P_2
wait (S2);
signal (S0);
```

How many times will P_O print '0'?

(a) exactly 2 times

(b) at least 2 times

(c) 1 time

(d) not printed

12. Each process P_i, i = 0,1,2,3,. . . . . ,9 is coded as follows.

```
Process P_i
while(true)

    wait(mutex)
    {
    Critical Section
    }
    signal(mutex)
```

The code for P10 is identical except that it uses signal (mutex) instead of wait(mutex). What is the largest number of processes that can be inside the critical section at any moment (the mutex being initialized to 1)?

(a) 1

(b) 2

(c) 3

(d) None of the mentioned

13. State whether the following statements are correct for the characteristics of the monitor:
i) The local data variables are accessible not only by the monitors' procedures but also by the external procedure.
ii) A process enters the monitor by invoking one of its procedures.
iii) Only one process may be excluded in the monitor at a time.

(a) i and ii only

(b) ii and iii only

(c) i and iii only

(d) All i, ii and iii

14. Consider the following proposed solution for the critical section problem. There are n processes: P_0,P_1 ....P_{n-1}. In the code, function P_{max} returns an integer not smaller than any of its arguments. For all i, t[i] is initialized to zero.

```
do {
c[i]=1; t[i] = pmax (t [0],..., t[n-1])+1; c[i]=0;
for every j!=i in {0,...,n-1} {
while (c[j]);
while (t [j] != 0 && t[j]<=t[i]);
}
Critical Section;
t[i]=0;
Remainder Section;
} while (true);
```

Which one of the following is TRUE about the above solution?

(a) At most one process can be in the critical section at any time

(b) The bounded wait condition is satisfied

(c) The progress condition is satisfied

(d) It cannot cause a deadlock

15. Consider the following solution to the producer-consumer synchronization problem. The shared buffer size is N. Three semaphores empty, full and mutex are defined with respective initial values of 0, N and 1. Semaphore empty denotes the number of available slots in the buffer, for the consumer to read from. Semaphore full denotes the number of available slots in the buffer, for the producer to write to. The placeholder variables, denoted by P, Q, R and S, in the code below can be assigned either empty or full. The valid semaphore operations are: wait() and sigmal().

Producer:	Consumer:

do{
wait(P);
wait(mutex);
//Add item to buffer
signal(mutex);
signal(Q);
}while(1);
	do{
	wait(R);
	wait(mutex);
	//Consume item from buffer
	signal(mutex);
	signal(S);
	}while(1);

Which one of the following assignments to P, Q, R and S will yield the correct solution?

(a) P: full, Q: full, R: empty, S: empty

(b) P: empty, Q: empty, R: full, S: full

(c) P: full, Q: empty, R: empty, S: full

(d) P: empty, Q: full, R: full, S: empty

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

DeepScience
Open Access Books

Chapter 5

Deadlock

Abstract: In multiprogramming environments, resource allocation among competing processes often leads to deadlocks, where processes are indefinitely blocked, waiting for resources held by each other. This chapter explores various aspects of deadlocks, including their definition, necessary conditions, and modeling techniques. It also delves into methods for deadlock detection, prevention, and avoidance, such as the Banker's Algorithm. By understanding these concepts and applying appropriate strategies, system reliability and efficiency can be enhanced, preventing the significant operational delays caused by deadlocks.

Keywords: Deadlock, Operating systems, Resource allocation, Deadlock detection, Deadlock prevention, Deadlock avoidance, Banker's Algorithm, Multiprogramming.

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and 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(Stallings, 2009). For example consider two processes A and B that each want to print a file currently on a CD-ROM Drive (Figure5.1).

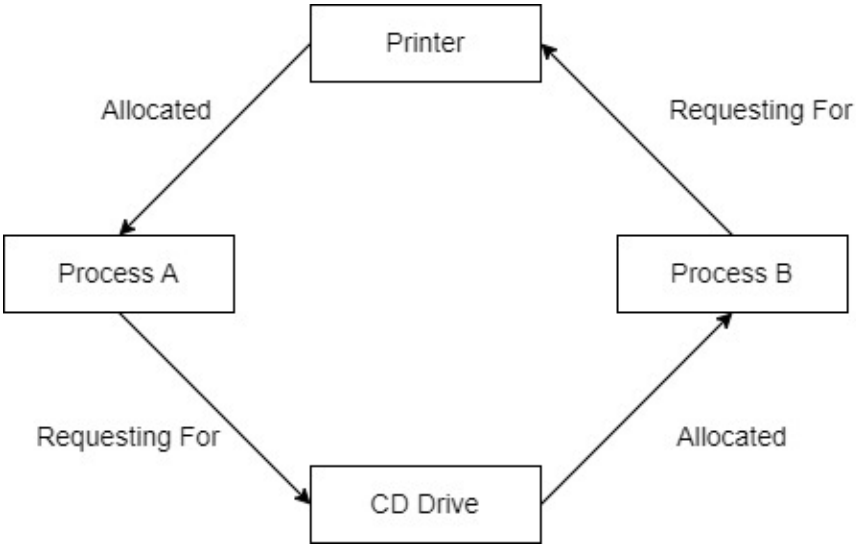


Figure 5.1: Deadlock Situation

1. A has obtained ownership of the printer and will release it after getting the CD Drive and printing one file.
2. B has obtained ownership of the CD drive and will release it after getting the printer and printing one file.
3. A tries to get ownership of the drive, but is told to wait for B to release it.
4. B tries to get ownership of the printer, but is told to wait for A to release it.

Result is deadlock!

5.1 Resources

A resource is an object granted to a process.Examples are Kernel Data Structures (Process Control Blocks, Threads, ...),Locks/semaphores to protect critical sections,Memory (page frames, buffers, etc.),Files,I/O Devices(printers, ports, tape drives, speaker, etc.).

5.1.1 Preemptable and Nonpreemptable Resources

Resources come in two types

1. Preemptable: meaning that the resource can be taken away from its current owner (and given back later). An example is memory.
2. Non-preemptable: meaning that the resource cannot be taken away. An example is a printer.

Remark. Deadlocks occur when processes are granted exclusive access to non-preemptable resources and wait when the resource is not available

The resources are used in following order:

1. Request
2. Allocate
3. Use
4. Release

Processes request the resource, use the resource, and release the resource. The allocate decisions are made by the system and we will study policies used to make these decisions.

5.2 Deadlock Modeling

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all the processes are waiting. No one of them will ever cause any of the events that could wake up any other process. So all the processes continue to wait forever. In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software. Figure 5.2 is giving examples of deadlocks in traffic scenario.

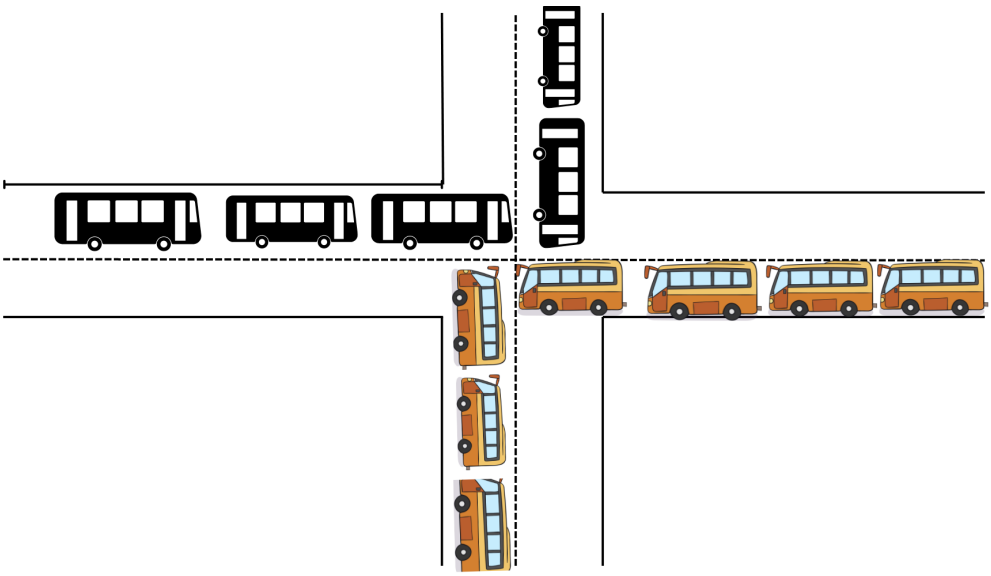


Figure 5.2: Deadlock Examples

5.2.1 Starvation vs. Deadlock

Starvation and Deadlock are two different things!
Deadlock: No work is being accomplished for processes that are deadlocked, because processes are waiting for each other (Deitel, 2004). If deadlock exist, no process can complete.
Starvation: Starvation is delay in getting resources for particular process. Work (progress) is still in progress. However, a particular set of processes may not be getting any work done because they cannot obtain the resources they need. May only last a short time. Starvation may removed after certain time.

5.2.2 Necessary Conditions for Deadlock

The following four conditions are necessary but not sufficient for deadlock. They are not sufficient.

- Mutual exclusion: A resource can be assigned to at most one process at a time (no sharing).
- Hold and wait: A processing holding a resource is permitted to request another.
- No preemption: A process must release its resources; they cannot be taken away.

- Circular wait: There must be a chain of processes such that each member of the chain is waiting for a resource held by the next member of the chain.

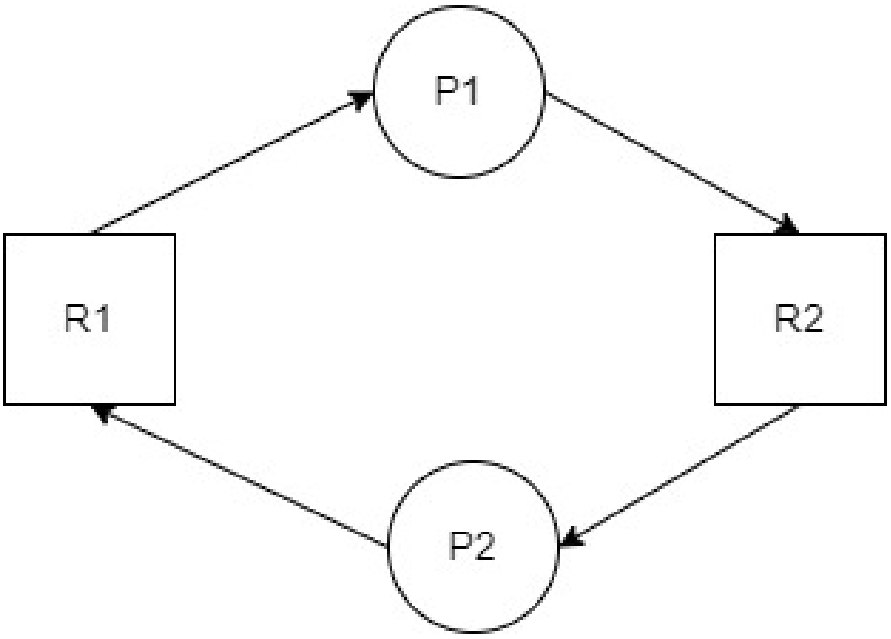


Figure 5.3: Circular Wait

One can say "If you want a deadlock, you must have these four conditions.". But of course you don't actually want a deadlock, so you would more likely say "If you want to prevent deadlock, you need only violate one or more of these four conditions.".

The first three are static characteristics of the system and resources. That is, for a given system with a fixed set of resources, the first three conditions are either always true or always false: They don't change with time. The truth or falsehood of the last condition does indeed change with time as the resources are requested/allocated/released.

5.2.3 Deadlock Representation

We can use Resource Allocation Graph, which also called a Reusable Resource Graph to model resource allocation.It uses following notation:

1. The processes are circles.
2. The resources are squares.
3. An arc (directed line) from a process P to a resource R signifies that process P has requested (but not yet been allocated) resource R.
4. An arc from a resource R to a process P indicates that process P has been allocated resource R.

5.3 Deadlock Solutions

There are four strategies used for dealing with deadlocks.

1. Ignore the problem

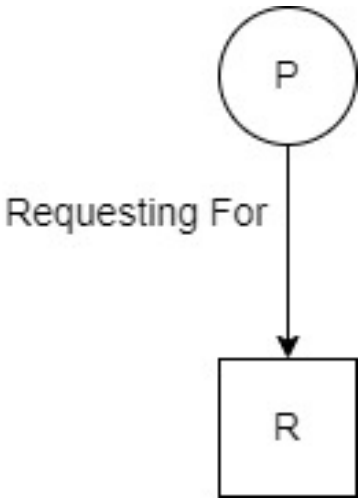


Figure 5.4: Resource Request

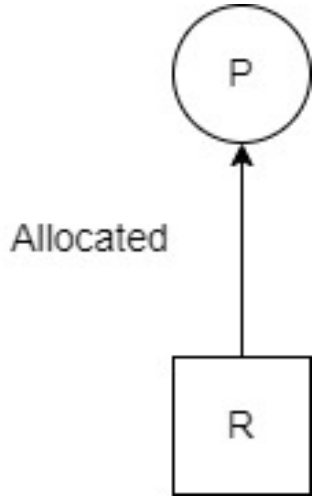


Figure 5.5: Resource Allocation

2. Detect deadlocks and recover from them
3. Prevent deadlocks by violating one of the 4 necessary conditions.
4. Avoid deadlocks by carefully deciding when to allocate resources.

5.3.1 Deadlock Ignorance

This is also called as *Ostrich Algorithm*. In it we assume that deadlock doesn't happen. In general, this is a reasonable strategy. Deadlock is unlikely to occur very often. A system can run for years without deadlock occurring. If the operating system has a deadlock prevention or detection system in place, this will have a negative impact on performance (slow the system down) because whenever a process or thread requests a resource, the system will have to check whether granting this request could cause a potential deadlock situation.
If deadlock does occur, it may be necessary to bring the system down, or at least manually kill a number of processes, but even that is not an extreme solution in most situations. Most operating systems, including UNIX, MINIX 3, and Windows, just ignore the problem.The

most users would prefer an occasional deadlock rather than restrictions like all users to one process, one open file, and one of everything. Solutions to deadlock may be complex that is the reason sometimes we have to prefer between convenience and correctness

5.3.2 Deadlock Detection and Recovery

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock.

This solution consists of:

- An algorithm to determine whether a deadlock has occurred.*Detection*
- An algorithm to recover from the deadlock.*Recovery*

Single Instance of Each Resource Type If there is only one instance of each resource, it is possible to detect deadlock by constructing a resource allocation/request graph and checking for cycles. There are number of algorithms exist to detect cycles in a graph. We can define 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.

Remark. an edge from ^{P}i to ^{P}j in a wait-for graph implies that process^{P}i is waiting for process ^{P}j, to release a resource that ^{P}i needs.

1. Initialize L to the empty list and designate all edges as unmarked
2. Add the current node to L and check to see if it appears twice. If it does, there is a cycle in the graph.
3. From the given node, check to see if there are any unmarked outgoing edges. If yes, go to the next step, if no, skip the next step
4. Pick an unmarked edge, mark it, then follow it to the new current node and go to step 3.
5. We have reached a dead end. Go back to the previous node and make that the current node. If the current node is the starting Node and there are no unmarked edges, there are no cycles in the graph. Otherwise, go to step 3.

Recovery When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to 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

1. Abort all deadlocked processes to break the deadlock cycle. the deadlocked processes may have computed for a long time, and all the work needs to be discarded.
2. Abort one process at a time until the deadlock cycle is eliminated. This method is less costly than first because it will kill fewer process.

We should abort those processes whose termination will not be costly(Will waste huge amount of work, important process). Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. In Resource Preemption we have to consider:

1. Selecting a process : From which resources are to be preempted.We preempt resource in a way to minimize cost. Cost of preemption is work done and number of resources a deadlocked process is holding.
2. Rollback:After preemption of resource, we must roll back the process to some safe state and restart it from that state. Simplest solution is a total rollback: Abort the process and then restart it. We also implement partial

5.3.3 Deadlock Prevention

A deadlock can occur, if each of the four necessary conditions are satisfied. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We can examine each condition and way to eliminate it:

- Mutual Exclusion: The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer,can not be allocated in shareable mode. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.
- Hold and Wait: There are two possibilities for elimination of Hold and Wait. The first alternative is that a process specify all of the resources it needs at once, prior to execution. By ensuring that its all resources can be given at that time, it do not have to wait for resources. The second alternative is to allow a process to request resources only when it has no resource. A process may request any additional resources only after it release all the resources that are currently held by it.

Remark. Both strategies suffers from problems.

First strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on all or none basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the **Hold and Wait** condition is denied and deadlocks simply cannot occur. This strategy can lead to *lower resources utilization*. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.
Starvation is also possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

- No Preemption: We can eliminate no preemption in some cases.If a process is holding some resources and requests another resource that cannot be immediately allocated to it. In that case all resources allocated to process are preempted. The preempted resources are made available for other waiting process. The process will be restarted only when it can regain its old resources,as well as the new ones that it is requesting.

  When a process is holding some resources and do not get additional resources it needs. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the *no-preemptive* condition effectively.
  This strategy is not useful in case of resources **where preemption is not possible**. For example printers, tape drives, and CD/DVD in write mode can not be preempted. It also requires **High Cost**. When a process release resources the process may lose all its work to that point.

- Circular Wait: The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as shown in table 5.1 We can now consider the following protocol to

Table 5.1: Resource Numbering

Resource number	Resource
1	Card reader
2	Printer
3	Plotter
4	Tape Drive
5	Card Punch

prevent deadlocks: Each process can request resources only in an increasing order of

enumeration.A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). All requests for synchronization objects must be made in increasing order.It is up to application developers to write programs that follow the ordering. If these protocol is used, then the circular-wait condition cannot hold.

5.3.4 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

It is still possible to avoid deadlock by being careful when resources are allocated. The most famous deadlock avoidance algorithm is **Banker's algorithm** given by Dijkstra. It is same like used by a banker in deciding if a loan can be safely made. It uses :

- Each process declare the maximum number of resources of each type that it may need.
- The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular wait condition.
- Resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.
- **Safe State** System is in safe state if there exists a safe sequence of all processes. When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- **Safe Sequence** Sequence ¡P1, P2, ..., Pn¿ is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j¡I. In other words there is a way of allocating resources to process in order ¡P1,P2,P3,..,Pn¿ to complete execution of all processes.
- **Unsafe State** A state which is not safe is called unsafe state. System is in unsafe state if we do not have safe sequence. An unsafe state may lead to deadlock.

Example 1. Consider a system with 12 tape drives. Assume there are three processes : P1,P2,P3. Assume we know the maximum number of tape drives that each process may request:
P1 : 10, P2 : 4, P3 : 9 Suppose at time t, 9 tape drives are allocated as follows :
P1: 5,P2: 2, P3: 2 So, we have three more tape drives which are free. This system is in a safe state because it we sequence processes as: $<$ P2, P1, P3 $>$, then P2 can get two more tape drives and it finishes its job, and returns four tape drives to the system. Then the system will have 5 free tape drives. Allocate all of them to P1, it gets 10 tape drives and finishes its job. P1 then returns all 10 drives to the system. Then P3 can get 7 more tape drives and it does its job.

Banker's Algorithm It assumes a process declares the maximum number of instances of each resource type that it may need at the time of entering the system. The algorithm decides, for each resource request, whether granting it would put the system in an unsafe state. It uses following data structures shown in table5.2

Safety Algorithm The safety algorithm for finding out whether or not a system is in a safe state. This algorithm can be described, as follows:

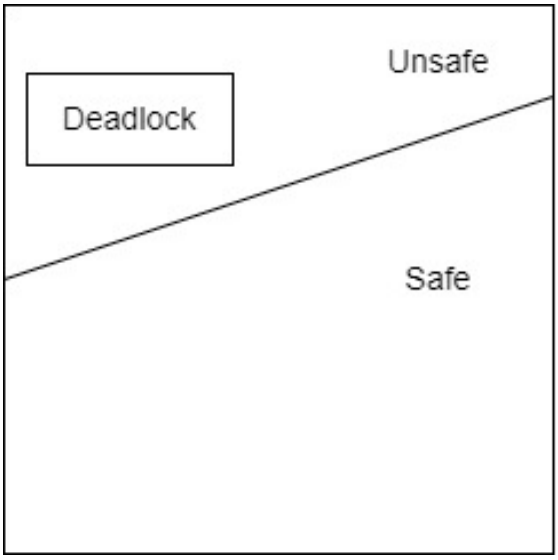


Figure 5.6: Safe and Unsafe State

Table 5.2: Data structures used in Deadlock Handling

Data Structure	Use	Description
Available[1..m]	resource availability	A vector of length m indicates the number of available resources of each type.
Max[1..n, 1..m]	maximum demand	An n x m matrix defines the maximum demand of each process.
Allocation[1..n, 1..m]	current allocation	An n x m in matrix defines the number of resources of each type currently allocated to each process.
Need[1..n, 1..m]	potential need	An n x m matrix indicates the remaining resource need of each process.

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

Resource - Request Algorithm This algorithm determines if requests can be safely granted.Let ^{Request}i be the request vector for process P. If Request[j]==k, then process P, wants k instances of resource type R;. When a request for resources is made by process P, the following actions are taken:

```
1. If Request < Need , go to step 2. Otherwise, raise an error
    condition, since
the process has exceeded its maximum claim.
2. If Request < Available, go to step 3. Otherwise, Ps must wait,
     since the
resources are not available.
3. If requested resources are allocated to
process P- Modify system state as follows:
Available = Available - Request;
Allocation  = Allocation  + Request;
Need  = Need - Request;
```

If the resulting resource-allocation state is safe, process P is allocated its resources. If the new state is unsafe, then P, must wait for Request. System is restored to old state(previous to resource-allocation) state.

Example

We are taking following example to illustrate Banker's algorithm.Assume that there are 5 processes, P0 through P4, and 4 types of resources. At some time we have the following system state represented by table 5.3,5.4 and 5.5. We can use the safety algorithm to test

Table 5.3: Allocation Matrix

	Allocation			
Process	A	B	C	D
P0	0	1	1	0
P1	1	2	3	1
P2	1	3	6	5
P3	0	6	3	2
P4	0	0	1	4

if the system is in a safe state. We will first define Work(table 5.6) and Finish(table 5.7).

Table 5.4: Max requirement

	Max			
Process	A	B	C	D
P0	0	2	1	0
P1	1	6	5	2
P2	2	3	6	6
P3	0	6	5	2
P4	0	6	5	6

Table 5.5: Available Resource

Available			
A	B	C	D
1	5	2	0

Table 5.6: Work matrix

Work			
A	B	C	D
1	5	2	0

Table 5.7: Finish matrix

Finish	
Process	
P0	FALSE
P1	FALSE
P2	FALSE
P3	FALSE
P4	FALSE

For completion of P0 the need of P0 (0,1,0,0) should be less than or equal to work. The need of P0 can be satisfied, so P0 can be completed and after completion of P0 it will release all resources. The work matrix can be updated by adding the allocated resources(0,1,1,0) for that process to work(table5.8). The Finish after P0 will be shown in table5.9. In similar

Table 5.8: Work

Work			
A	B	C	D
1	6	3	0

Table 5.9: Finish after completion of P0

Finish	
Process	
P0	TRUE
P1	FALSE
P2	FALSE
P3	FALSE
P4	FALSE

way the completion of P1 can be checked. We have to check each element of the vector Need of P1(0,4,2,1) against the corresponding element in Work(1,6,3,0). Because P1 need 1 resource of D type and which is not available currently (the fourth element of Work), we need to move on to P2. Need of 2 (1,0,0,1) is not less than work, so must move on to P3. Need of P3 (0,0,2,0) is less than work, so we can update work and finish. In the same way we can check completion of P4. The P4 can be completed with resources available in Work. After that P1 and P2 can be completed.
Hence the safe Sequence will be: P0,P3,P4,P1,P2.

Conclusion: Deadlocks present a significant challenge in multiprogramming systems, as they can halt system operations by blocking processes indefinitely. This chapter provided a comprehensive overview of deadlocks, detailing the conditions required for their occurrence and the various strategies to manage them. Deadlock detection involves identifying cycles in resource allocation graphs, while prevention and avoidance techniques focus on structuring resource requests and allocations to prevent circular waits. The Banker's Algorithm, a well-known avoidance strategy, was discussed in detail. Understanding and implementing these methods can greatly reduce the risk of deadlocks, ensuring smoother and more efficient system operations.

5.4 Exercise

1. What are the four necessary conditions for deadlock?
2. What are the various strategies for handling for deadlock?
3. Explain Bankers Algorithm for Deadlock avoidance?
4. Describe Deadlock Prevention? What are main disadvantages of implementing Prevention?
5. What are various issues in deadlock recovery?
6. Explain Deadlock detection for single resource type?

5.5 Multiple Choice Questions

1. What is the primary difference between preemptable and nonpreemptable resources?
   (a) Preemptable resources can be forcibly reclaimed, while nonpreemptable resources cannot
   (b) Nonpreemptable resources can be forcibly reclaimed, while preemptable resources cannot
   (c) Preemptable resources lead to deadlock, while nonpreemptable resources do not
   (d) Nonpreemptable resources lead to deadlock, while preemptable resources do not
2. Which of the following statements best defines deadlock in the context of operating systems?
   (a) Deadlock is a situation where a process is waiting indefinitely for resources held by other processes
   (b) Deadlock is a situation where a process is waiting for a resource that is available
   (c) Deadlock is a situation where all processes are actively utilizing resources
   (d) Deadlock is a situation where resources are dynamically allocated and deallocated
3. What distinguishes starvation from deadlock?
   (a) Starvation occurs when a process is blocked indefinitely, whereas deadlock involves multiple processes
   (b) Starvation involves a single process, while deadlock involves multiple processes
   (c) Starvation occurs due to resource allocation policies, while deadlock occurs due to resource contention
   (d) Starvation can be resolved through deadlock detection techniques, whereas deadlock cannot
4. Which of the following is NOT a necessary condition for deadlock?
   (a) Mutual exclusion
   (b) Hold and wait
   (c) No preemption

(d) Circular wait

5. Deadlock detection and recovery involve:

   (a) Proactively preventing deadlocks before they occur
   (b) Detecting the presence of a deadlock and taking corrective actions
   (c) Ignoring the existence of deadlocks and focusing on system performance
   (d) Avoiding the situations that lead to deadlocks in the first place

6. Which approach to handling deadlock involves periodically checking the system for deadlock conditions?

   (a) Deadlock ignorance
   (b) Deadlock detection and recovery
   (c) Deadlock prevention
   (d) Deadlock avoidance

7. What is the primary goal of deadlock prevention techniques?

   (a) To detect deadlocks as soon as they occur
   (b) To avoid the conditions that can lead to deadlock
   (c) To recover from deadlocks efficiently
   (d) To ignore deadlocks and prioritize system performance

8. Which of the following is a characteristic of deadlock avoidance?

   (a) Deadlocks are not detected
   (b) Deadlocks are detected but not recovered from
   (c) Deadlocks are prevented before they occur
   (d) Deadlocks are allowed to occur and then resolved

9. Which deadlock solution technique is based on ensuring that the system remains in safe states?

   (a) Deadlock ignorance
   (b) Deadlock detection and recovery
   (c) Deadlock prevention
   (d) Deadlock avoidance

10. In deadlock avoidance, what is the significance of the safe state?

    (a) It indicates a state where deadlock is inevitable
    (b) It indicates a state where deadlock is impossible
    (c) It indicates a state where deadlock may occur but can be avoided
    (d) It indicates a state where deadlock has already occurred

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 6

Memory Management

Abstract: This chapter delves into the fundamental aspects of memory management in operating systems. It covers the definition and importance of memory management, including logical and physical address maps. The chapter explores various memory allocation methods, such as contiguous memory allocation with fixed and variable partitions, addressing issues like internal and external fragmentation and the concept of compaction. Detailed discussions on paging, including its operation, hardware support, protection, sharing, and its advantages and disadvantages, are presented. The chapter also examines segmentation and the comparison between paging and segmentation, highlighting the benefits and drawbacks of each approach. Additionally, paged segmentation is introduced, along with its respective advantages and disadvantages. The chapter concludes with exercises and multiple-choice questions to reinforce the concepts discussed.

Keywords: Memory Management, Logical Address, Physical Address, Contiguous Memory Allocation, Internal Fragmentation, External Fragmentation, Compaction, Paging, Hardware Support, Protection, Shared Pages, Segmentation, Paged Segmentation.

Basic Memory Management: Definition ,Logical and Physical address map , Memory allocation : Contiguous Memory allocation – Fixed and variable partition – Internal and External fragmentation and Compaction , Paging : Principle of operation, Page allocation, Hardware support for paging,Protection and sharing, Disadvantages of paging.

6.1 Introduction

With CPU scheduling, we can improve CPU utilization and response to its users. We allow several processes in main memory and CPU switches among them for multitasking(timesharing). We can say all processes shares main memory.There are various ways to manage memory for example bare-machine approach, paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on hardware design of the system.
Memory consists of a large array of words or bytes, each with its own address. The CPU brings instructions from memory. Program counter(PC) points to the address of next instructions to be executed. These instruction may cause additional loading from and storing to specific memory addresses. CPU may generate address of instruction executed and also for data which is loaded or stored from main memory.

The memory unit sees only memory addresses, it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).

6.1.1 Basic Hardware

CPU can access Main memory(also cache memory) and the registers directly. There are machine instructions contains memory addresses as arguments, but no instruction points to disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in main memory and registers, they must be copied from secondary storage to main memory before the CPU can operate on them. Registers operates at the speed of CPU. Main Memory access may take many cycles of the CPU clock to complete. Main memory is slower than CPU. To improve performance we add a high speed memory which is called as *cache*. To ensure correct operation,There must be protection between user processes from one another. This protection must be provided by the hardware. Memory management provides protection by using two registers, a base register and a limit register. The base register holds the smallest legal physical memory address and the limit register specifies the size of the range. For example, if the base register holds 300000 and the limit register is 1209000, then the program can legally access all addresses from 300000 through 411999. Protection of memory space is accomplished by having the CPU hardware compare even/

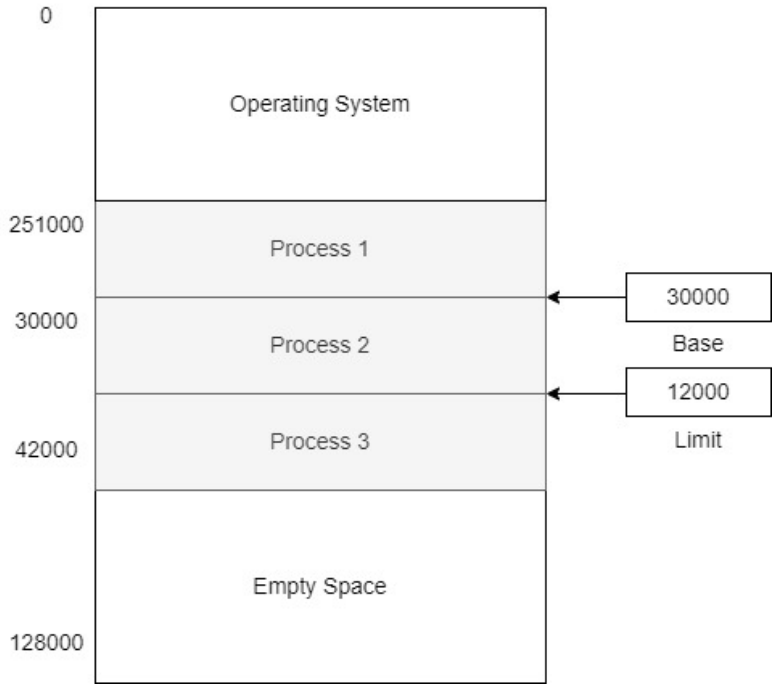


Figure 6.1: Memory Management System

address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error. This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions

can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

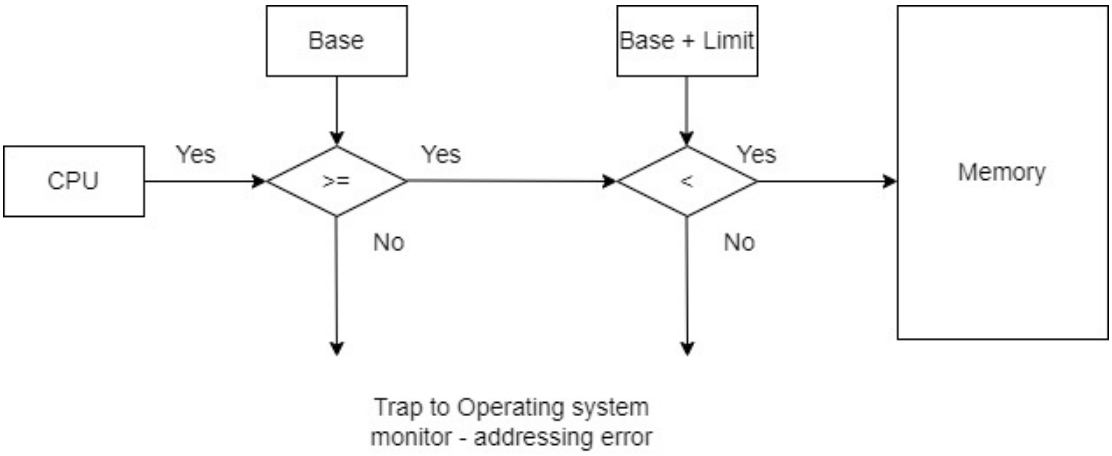


Figure 6.2: Hardware Address Protection

6.1.2 Address Binding

User programs typically refer to memory addresses with symbolic names such as "i", "count", and "averageTemperature". These symbolic names must be mapped or bound to physical memory addresses, which typically occurs in several stages:

1. Compile Time: If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.

2. Load Time: If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.

3. Execution Time: If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.

6.1.3 Logical and Physical Address

The address generated by the CPU is a *logical address*, whereas the address actually seen by the memory hardware is a *physical address*. Addresses bound at compile time or load time have identical logical and physical addresses. Addresses created at execution time, however, have different logical and physical addresses. In this case the logical address is also known as a *virtual address*, and the two terms are used interchangeably.
The set of all logical addresses used by a program composes the logical address space, and the set of all corresponding physical addresses composes the physical address space. The

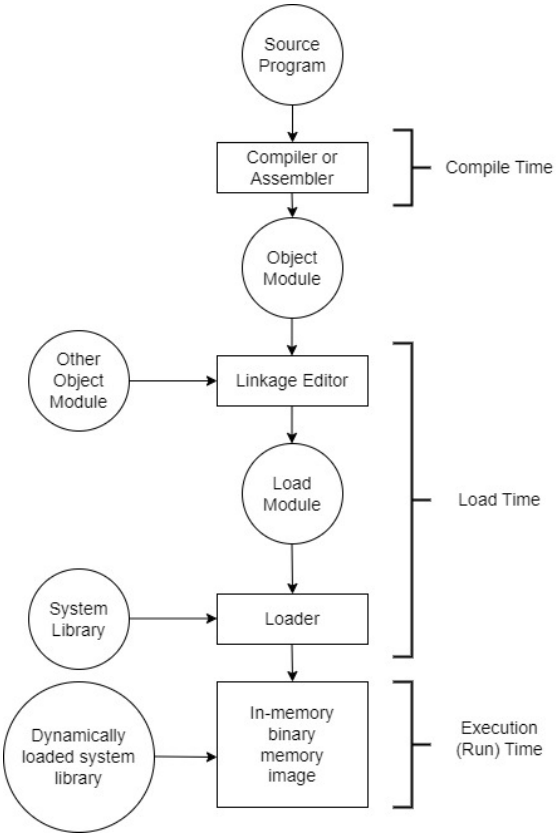


Figure 6.3: Address Binding

run time mapping of logical to physical addresses is handled by the **Memory-management unit**, MMU (Dhamdhere, 2003). The MMU can take on many forms. One of the simplest is extension of the base-register scheme described earlier. The base register is now termed a relocation register, whose value is added to every memory request at the hardware level. Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses, all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. We now have two different types of addresses: logical addresses (in the range *0* to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

6.1.4 Dynamic Loading and Dynamic Linking

Dynamic loading loads up each routine as it is called, rather than loading an entire program into memory at once. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and

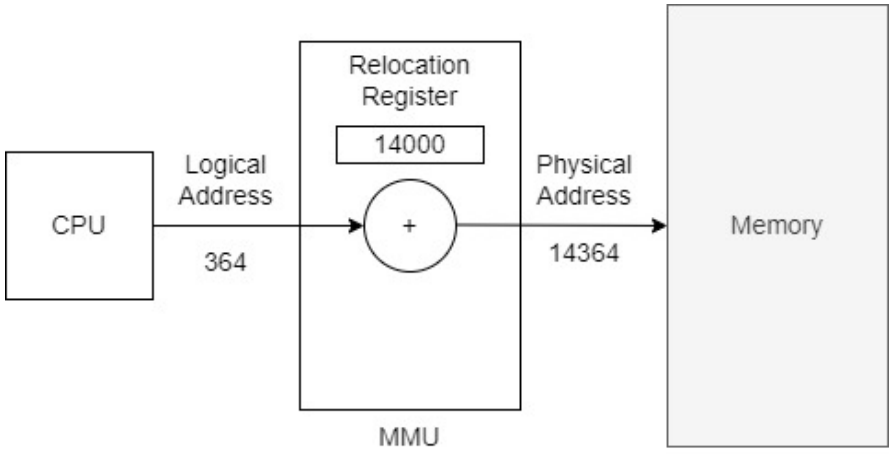


Figure 6.4: Dynamic Relocation

then then loading it up if it is not already loaded. In static linking library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code. In **dynamic linking**, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time. This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs. If the code section of the library routines is *reentrant*, (meaning it does not modify the code while it runs, making it safe to re-enter it), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it. (Each process would have their own copy of the data section of the routines, but that may be small relative to the code segments.) Obviously the OS must manage shared routines in memory. An added benefit of **dynamically linked libraries** (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates. When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes. However if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.

6.2 Swapping

A process must be loaded into memory in order to execute.If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the backing store. Backing store is capable of providing direct access to these memory images. Major time consuming part of swapping is transfer time. Total transfer time is directly proportional to the amount of memory swapped. Let us assume that the user process is of size 100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second. The actual transfer of the 100K process to or from memory will take 100KB / 1000KB per second = 1/10 second = 100 milliseconds It is important to swap processes out of memory only when they are idle, or more to the point, only when

there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes. Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available.(e.g. Paging)

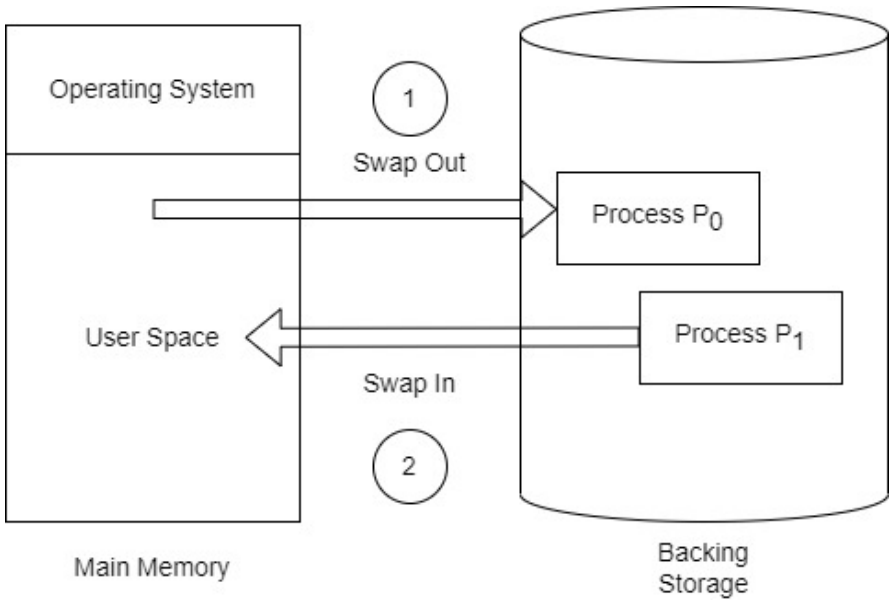


Figure 6.5: Swapping

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called roll out, roll in.

6.3 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate the parts of the main memory in the most efficient way possible. One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed.
The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. The OS is usually loaded low, because that is where the interrupt vectors are located. We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses. With relocation and limit registers, each logical address must be less than the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

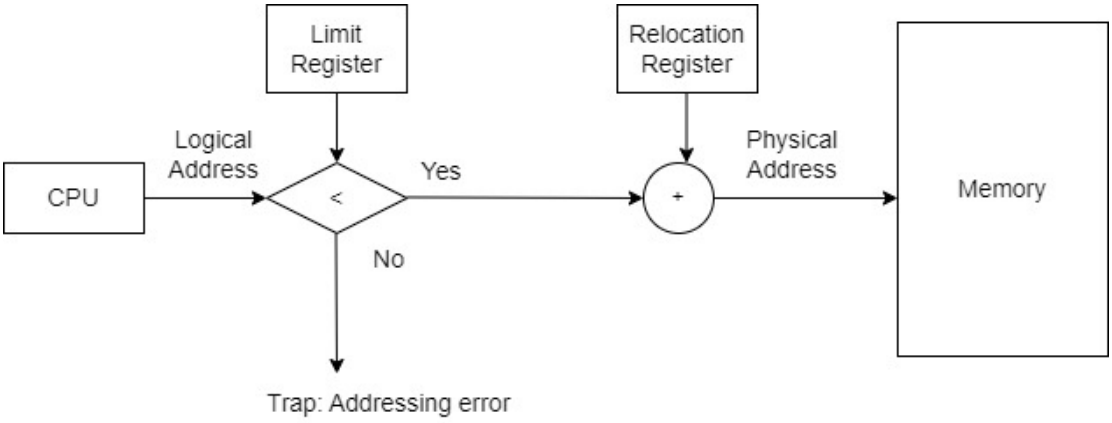


Figure 6.6: Hardware Support for Relocation

6.3.1 Memory Allocation

Simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used. An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. First fit : Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.

2. Best fit : Allocate the smallest hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.

3. Worst fit : Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests. Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

6.3.2 Fragmentation

there are two types of fragmentation:

1. External Fragmentation All the memory allocation strategies suffer from external fragmentation, though first and best fits experience the problems more so than worst fit. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.
   The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list. Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation. If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via *compaction*, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses. Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

2. Internal Fragmentation It occurs with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average $1/2$ block will be wasted per memory request, because on the average the last allocated block will be only half full.
   Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.
   Some systems use variable size blocks to minimize losses due to internal fragmentation.

6.4 Paging

Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as pages.

Paging eliminates most of the problems of the other methods discussed previously, and is the frequently used memory management technique used today.

The basic idea behind paging is to divide physical memory into a number of equal sized blocks called *frames*, and to divide a programs logical memory space into blocks of the same size called *pages*. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames. Any page (from any process) can be placed into any available frame. The page table is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory: A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.) The page table maps the page number to a frame number, to yield a physical address which also has

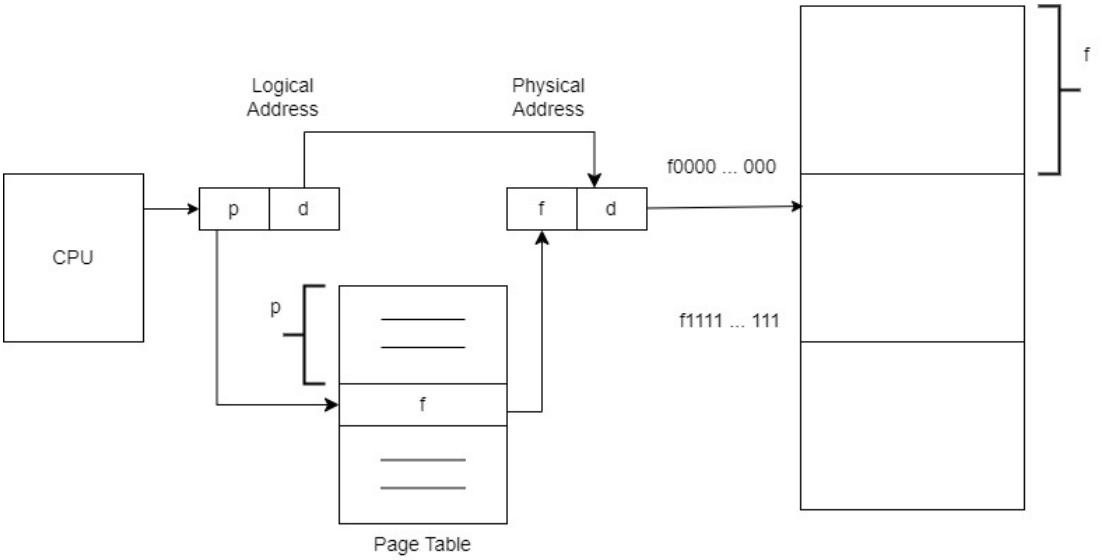


Figure 6.7: Paging Hardware

two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame. Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space. There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory. There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process. Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.

Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^{32} physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. ($32 + 12 = 44$ bits of physical address space.) When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table. Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space. The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

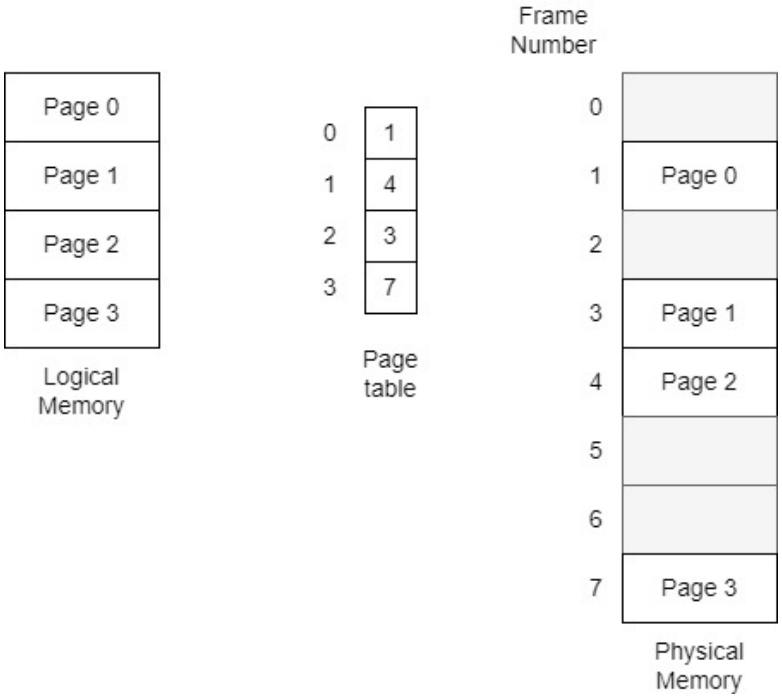


Figure 6.8: Paging Tables

6.4.1 Hardware Support for Paging

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The use of registers for the page table is satisfactory if the page table is very small. Currently computers allows the page table to be very large. For that page table is kept in main memory, and a page-table base register (PTBR) points to the page table. With this scheme, two memory accesses are needed to access a byte (one for the page-table entry, one for the byte).
The standard solution to this problem is to use a special, small, fastlookup hardware cache, called a *translation look-aside buffer* (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast, but the hardware is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024. The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device. Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated. If the TLB is full, then replacement strategies range from least-recently used, LRU to random. Some TLBs allow some entries to be wired down, which means that they cannot be removed from the TLB. Typically these would be kernel frames. Some TLBs store address-space identifiers, ASIDs, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting

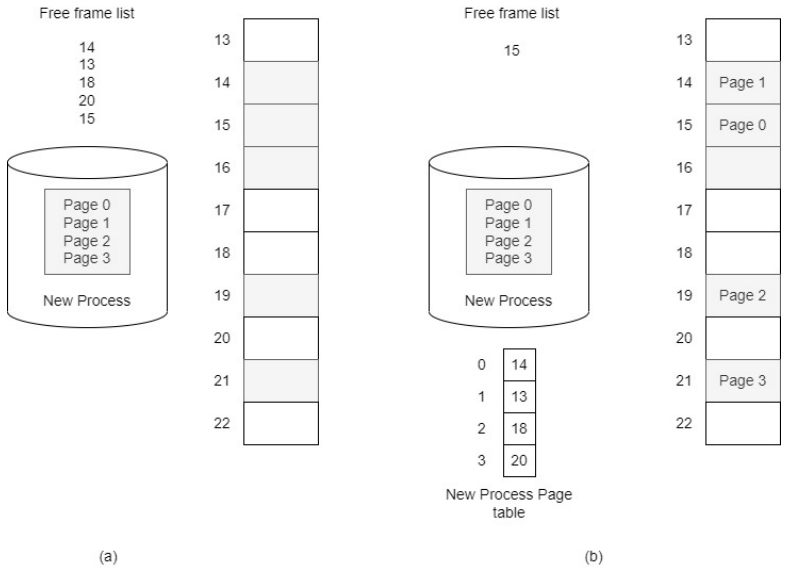


Figure 6.9: Free Frame before and after Allocation

one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch. The percentage of time that the desired information is found in the TLB is termed the *hit ratio*.

6.4.2 Protection

The page table can also help to protect processes from accessing memory which belongs to OS or other process. A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.
Valid / invalid bits can be used in the page table to verify pages which can be accessible by current process.When this bit is set to **valid** the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to **invalid** the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid-invalid bit.

Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. (Areas of memory in the last page that are not entirely filled by the process, and may contain data left over by whoever used that frame last.) Many processes do not use all of the page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a page-table length register, PTLR, to specify the length of the page table.

6.4.3 Shared Pages

An advantage of paging is the possibility of sharing common code. Consider a system that supports 40 users, each of them uses a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is

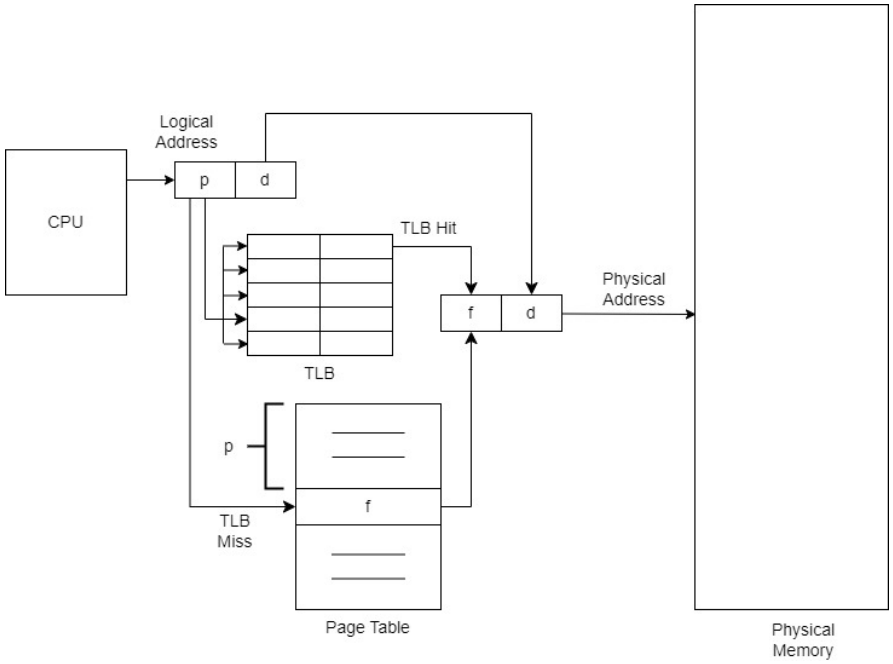


Figure 6.10: Paging Hardware with TLB

reentrant code (or pure code), then it can be shared.

Reentrant code is non-self-modifying code. It never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.

For given scenario, only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB.
Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. Generally operating systems implement shared memory using shared pages.

6.4.4 Advantages

1. Allocating memory is easy.
2. Eliminates external fragmentation
3. Supports Noncontiguous Allocation. Data (page frames) can be scattered all over PM
4. Can be extended to implement Virtual Memory by demand paging.
5. More efficient than basic swapping.

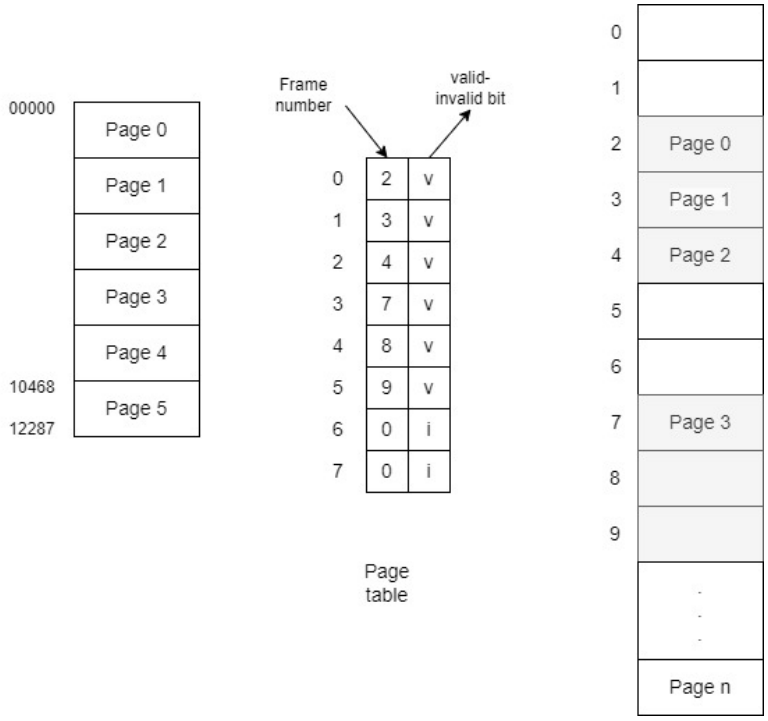


Figure 6.11: Valid Invalid bits in Paging

6. No need for considerations about fragmentation.

7. Provide efficient utilization of main memory by swapping out page least likely to be used.

6.4.5 Disadvantages

1. Longer memory access times (page table lookup), Access to page table can be optimized by using TLB.

2. Space overhead of page table.

3. Memory requirements (one entry per VM page)

4. Page Table Length Register (PTLR) to limit virtual memory size

5. Suffers from Internal fragmentation. Average Internal fragmentation=np/2, n=number of process,p=page size.

6.5 Segmentation

Segmentation is the division of a computer's primary memory into segments or sections. In segmentation, a reference to a memory location includes segment number and an offset(distance within that segment). Segments are also used in object files of compiled programs when they are linked together into a program image and when the image is loaded into memory.
Segments usually correspond to natural divisions of a program such as individual routines or data tables so segmentation is generally more visible to the programmer. Certain segments

may be shared between programs. Unlike paging, segment are having varying sizes and thus eliminates internal fragmentation. External fragmentation still exists.

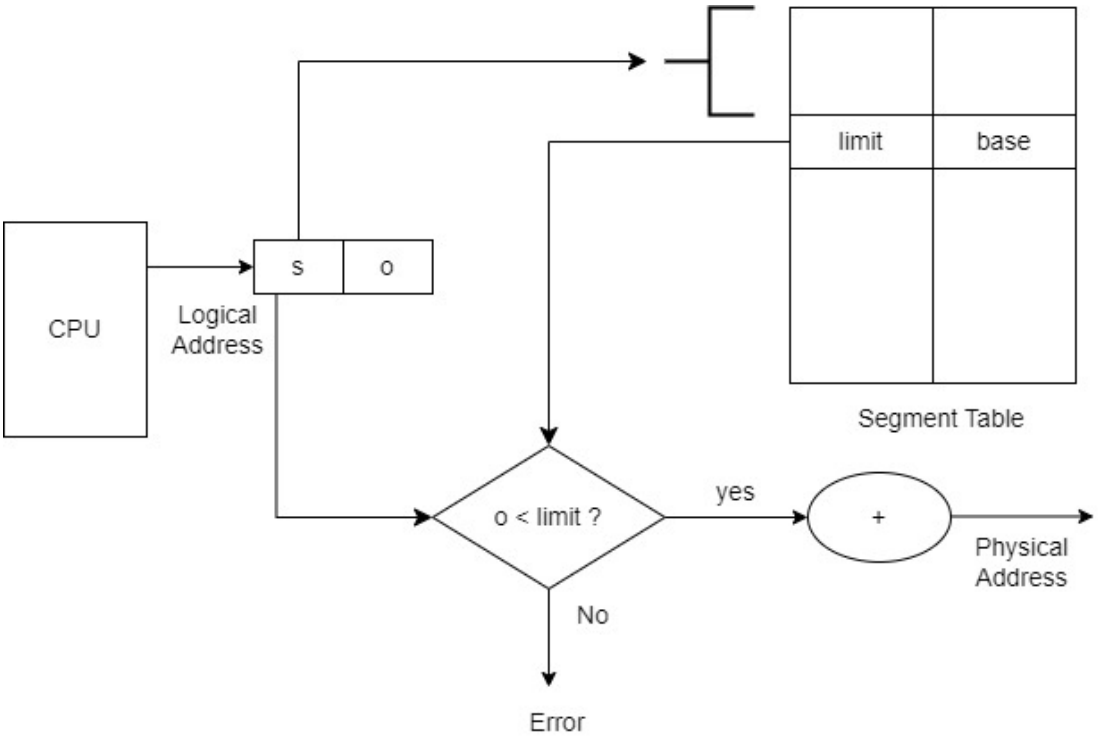


Figure 6.12: Segmentation

Segmentation is a memory management technique used in operating systems to divide a process's logical address space into variable-sized segments. Each segment represents a logical unit of a program, such as code, data, stack, or heap.

6.5.1 Advantages

Segmentation offers several advantages in memory management:

1. Segmentation allows programs to be divided into logical units or segments based on their functionality (e.g., code, data, stack, heap).

2. segmentation accommodates segments of variable sizes. This flexibility enables efficient memory allocation, as segments can dynamically adjust to the memory requirements of individual processes or programs.

3. Segmentation facilitates the management of growing data structures, such as arrays or linked lists, by allowing the allocation of additional memory segments as needed.

4. Segmentation provides a level of memory protection by assigning access rights to individual segments.

5. Segmentation simplifies memory addressing by breaking down the address space into logical segments, each with its own base address.

6. Segmentation enables efficient sharing of data between processes or programs by allowing multiple processes to access the same segment.

6.5.2 Disadvantages

1. Segmentation can lead to external fragmentation.

2. Managing variable-sized segments can be more complex compared to fixed-size blocks used in paging. Segmentation requires maintaining segment tables or segment descriptors to map logical addresses to physical addresses, which adds overhead to memory management operations and may degrade system performance.

3. Address translation in segmentation involves two steps: first, translating logical addresses to segment numbers using the segment table, and then translating segment numbers to physical addresses using segment descriptors. This two-step translation process can be more time-consuming and complex than the single-step translation used in paging.

4. Segmentation may not efficiently handle large processes that require contiguous memory allocation. As segments are allocated independently, large segments may not find contiguous free memory space, leading to fragmentation issues and reduced performance for large processes.

5. Segmentation introduces additional complexity to memory protection mechanisms.

6.5.3 Comparison of Paging and Segmentation

paging divides memory into fixed-size blocks, while segmentation divides the logical address space into variable-sized segments. Both techniques have their advantages and disadvantages.The comparison of paging and segmentation is given in table 6.1.

6.6 Paged Segmentation

Paged segmentation combines aspects of both paging and segmentation memory management techniques. In this approach, the logical address space of a process is divided into variable-sized segments, as in segmentation, but each segment is further divided into fixed-size blocks or pages, similar to paging. The mapping between logical addresses and physical addresses is done in two stages:

1. Segmentation:

   - The logical address consists of a segment number and an offset within that segment.
   - Each segment has its own base address and limit stored in a segment table. The base address indicates where the segment starts in physical memory, and the limit specifies the size of the segment.
   - When a process references a memory address, the segment number is used to index the segment table to obtain the base address of the segment.

2. Paging:

Table 6.1: Comparison of Paging and Segmentation

Features	**Paging**	**Segmentation**
Memory Size	In Paging, a process address space is broken into fixed sized blocks called pages.	In Segmentation, a process address space is broken in varying sized blocks called Segments.
Accountability	Operating System divides the memory into pages.	Compiler is responsible to calculate the segment size, the virtual address and actual address.
Size	Page size is determined by available memory.	Segment size is determined by the user.
Speed	Paging technique is faster in terms of memory access.	Segmentation is slower than paging.
Fragmentation	Paging can cause internal fragmentation as some pages may go underutilized.	Segmentation can cause external fragmentation as some memory block may not be used at all.
Logical Address	During paging, a logical address is divided into page number and page offset.	During segmentation, a logical address is divided into section number and section offset.
Table	During paging, a logical address is divided into page number and page offset.	During segmentation, a logical address is divided into section number and section offset.
Data Storage	Page table stores the page data.	Segmentation table stores the segmentation data.

- Once the base address of the segment is obtained, the offset within the segment is used to locate the specific page in physical memory.
- A page table is used to map pages within each segment to physical page frames in memory.

6.6.1 Advantages

1. Flexibility: Paged segmentation combines the flexibility of segmentation with the efficiency of paging. It allows for variable-sized segments, which can grow dynamically, accommodating the changing memory requirements of processes.

2. Efficient Memory Management: Paging reduces internal fragmentation by dividing memory into fixed-size pages, while segmentation provides logical organization and flexibility. This combination optimizes memory utilization and allocation.

3. Protection and Sharing: Segmentation allows for protection and sharing of segments, while paging facilitates efficient sharing of individual pages. This combination enables both coarse-grained and fine-grained memory protection and sharing mechanisms.

6.6.2 Disadvantages

1. Complexity: Paged segmentation introduces additional complexity to the memory management system due to the combined mechanisms of segmentation and paging. Managing both segment tables and page tables can be more complex than using either technique individually.
2. Overhead: Maintaining both segment tables and page tables requires additional memory overhead and computational overhead for address translation, which can impact system performance.
3. Fragmentation: Paged segmentation may still suffer from fragmentation issues, including both internal fragmentation within pages and external fragmentation between segments.

Paged segmentation offers a balance between the flexibility of segmentation and the efficiency of paging, but it comes with added complexity and overhead in memory management. It's a trade-off between flexibility and performance that operating systems must consider based on the specific requirements of their applications and hardware.

Conclusion: In summary, this chapter has provided a comprehensive overview of memory management techniques crucial for the efficient operation of an operating system. By understanding the principles of memory allocation, paging, and segmentation, along with their respective advantages and disadvantages, one can appreciate the complexities involved in managing system memory. The exercises and multiple-choice questions at the end of the chapter serve to solidify the reader's grasp of these essential concepts, ensuring a well-rounded understanding of memory management strategies.

6.7 Exercise

1. Segmentation suffers from internal fragmentation or external fragmentation? Justify.
2. In a paging scheme,16-bit addresses are used with a page size of 512 bytes. If the logical address is 0000010001111101, how many bits are used for the page number and offset? What will be the physical address, if the frame address corresponding to the computed page number is 15.
3. Distinguish between:

   (i) Static and dynamic allocation of memory

   (ii) Swapping and paging

   (iii) Page, frame and segment
4. Describe the two difficulties with the use of equal size fixed partitions.
5. Differentiate between internal vs external fragmentation.
6. Describe the process of address binding in memory management, including its significance in the context of operating systems.
7. Discuss the different types of address binding and how they relate to dynamic loading and dynamic linking.
8. What is a significant advantage of paging over contiguous memory allocation?

6.8 Multiple Choice Questions

1. Logical memory is broken into blocks of the same size called

   (a) frames
   (b) pages
   (c) backing store
   (d) None of these

2. Every address generated by the CPU is divided into two parts: (choose two)

   (a) frame bit
   (b) page number
   (c) page offset
   (d) frame offset

3. The _____ is used as an index into the page table.

   (a) frame bit
   (b) page number
   (c) page offset
   (d) frame offset

4. The _____ table contains the base address of each page in physical memory.

   (a) Process
   (b) memory
   (c) page
   (d) frame

5. The size of a page is typically

   (a) can by anything
   (b) power of 2
   (c) power of 4
   (d) None of these

6. Which one of the following is the address generated by CPU?

   (a) physical address
   (b) absolute address
   (c) logical address
   (d) none of the mentioned

7. The address of a page table in memory is pointed by

   (a) stack pointer
   (b) page table base register

(c) page register

(d) program counter

8. Operating System maintains the page table for

(a) each process

(b) each thread

(c) each instruction

(d) each address

9. What is compaction?

(a) a technique for overcoming internal fragmentation

(b) a paging technique

(c) a technique for overcoming external fragmentation

(d) a technique for overcoming fatal error

10. With paging there is no _____ fragmentation.

(a) internal

(b) external

(c) either type of

(d) none of the mentioned

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

DeepScience
Open Access Books

Chapter 7

Virtual Memory

Abstract: This chapter provides an in-depth examination of page replacement algorithms within the context of virtual memory management in operating systems. It discusses the principles and implementations of various algorithms, including First-In, First-Out (FIFO), Optimal Page Replacement (OPT), Least Recently Used (LRU), Second Chance, and Not Recently Used (NRU). The chapter explores the theoretical underpinnings of these algorithms, their practical implementations, and their performance implications. Through comparative analysis, the chapter highlights the trade-offs involved in choosing different algorithms and their impact on system efficiency.

Keywords: Virtual Memory, Page Replacement Algorithms, FIFO, LRU, Optimal Page Replacement (OPT), Second Chance Algorithm, NRU, Memory Management.

Virtual memory is a technique which gives an the impression that programmer have contiguous working memory (an address space), while in fact memory allocated may be physically fragmented and may be part of program is stored in disk. Systems that use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual 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 be loaded fully in main memory

1. User written error handling routines are used only when an error occurred 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.
4. The ability to execute a program that is only partially in memory would counter many benefits. Less number of I/O would be needed to load or swap each user program into memory.
5. A program can be written without considering the amount of physical memory that is available.
6. 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.

7.1 Locality of Reference

Locality of reference, also known as the *principle of locality*, is the phenomenon of the same value or related storage locations being frequently accessed.
There are two basic types of reference locality:

1. Temporal locality refers to the reuse of specific data and/or resources within relatively small time durations.

2. Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, e.g., traversing the elements in a one-dimensional array.

7.2 Page Fault

A page is a fixed-length block of memory that is used as a unit of transfer between physical memory and external storage like a disk.

A page fault is an interrupt (or exception) raised by the hardware, when a program accesses a page that is mapped in address space, but not loaded in physical memory. Page Fault occurs when we access page which is currently not in main memory.

The hardware that detects this situation is the memory management unit in a processor. The exception handling software that handles the page fault is generally part of an operating system. The operating system tries to handle the page fault by making the required page accessible at a location in physical memory or kills the program in case it is an illegal access. Contrary to what their name might suggest, page faults are not errors and are common and necessary to increase the amount of memory available to programs in any operating system that utilizes virtual memory, including Microsoft Windows, Mac OS X. **Hardware generates a page fault for page accesses in following situation:**

1. Page corresponding to the requested address is not loaded in memory.

2. Page corresponding to the memory address accessed is loaded, but its present status is not updated in hardware.

3. In exception known as the protection fault is generated for page accesses where,the page is not part of the program, and so is not mapped in program memory.

4. In exception, program does not have sufficient privileges to read or write the page. the page access is legal, but it is mapped with demand paging.

7.3 Page Replacement Algorithm

Page replacement algorithms are the techniques using which Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again then it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm. A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults.

7.3.1 The First-In, First-Out (FIFO) Page Replacement Algorithm

Simplest paging algorithm is the FIFO (First-In, First-Out) algorithm. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page which is oldest is removed and the new page added to the tail of the list. As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults: Although FIFO

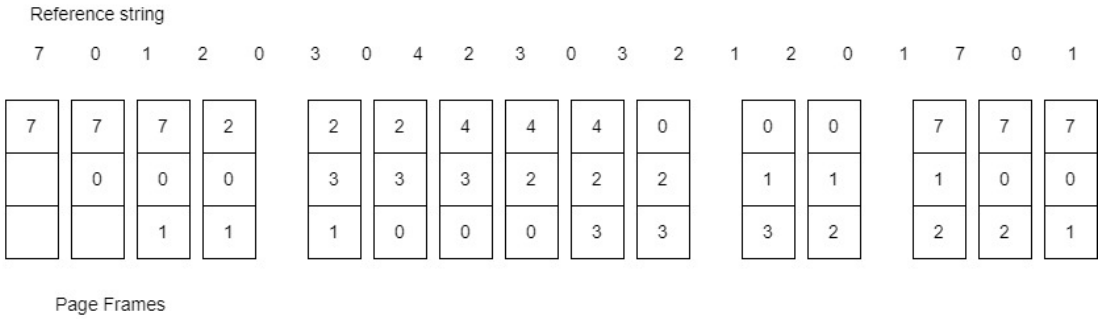


Figure 7.1: FIFO Page Replacement Algorithm

is simple and easy, it is not always optimal, or even efficient. An interesting effect that can occur with FIFO is Belady's anomaly, in which increasing the number of frames available can actually increase the number of page faults that occur.

7.3.2 Optimal Page Replacement

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. Replace the page that will not be used for the longest period of time . Use the time when a page is to be used. Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms. In practice most page-replacement algorithms try to approximate OPT by predicting (estimating) in one fashion or another what page will not be used for the longest period of time. The basis of FIFO is the prediction that the page that was brought in the longest time ago is the one that will not be needed again for the longest future time, but as we shall see, there are many other prediction methods, all striving to match the performance of OPT.

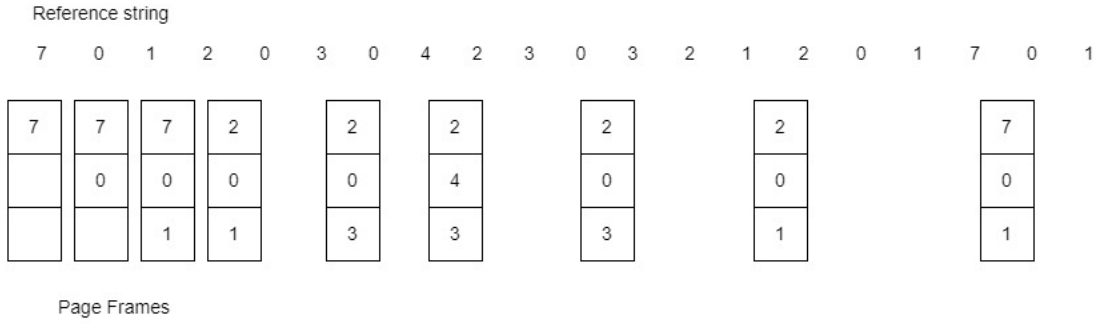


Figure 7.2: Optimal Page Replacement Algorithm

7.3.3 LRU Page Replacement Algorithm

The prediction behind LRU, the Least Recently Used, algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest load time, and the latter looks at the oldest use time.) Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.) LRU is considered a good

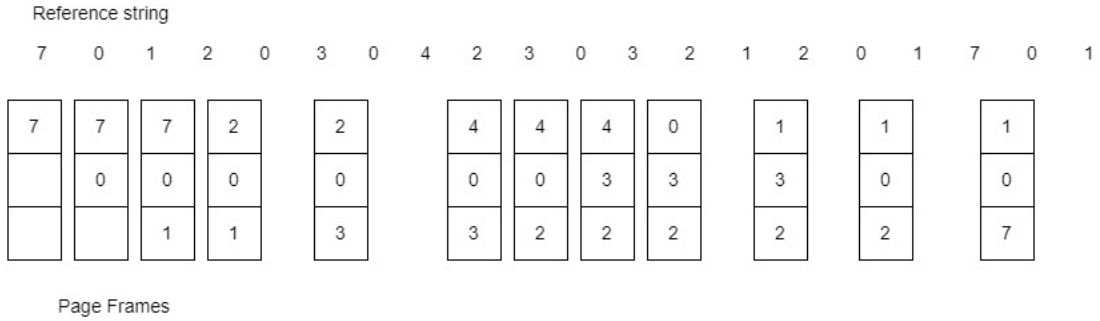


Figure 7.3: LRU Page Replacement Algorithm

replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used: Counters. Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered. Stack. Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure. Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for every memory access. Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called stack algorithms, which can never exhibit Belady's anomaly.

7.3.4 Second chance algorithm

The second chance algorithm is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table. When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner. If a page is found with its reference bit not set, then that page is selected as the next victim. If, however, the next page in the FIFO does have its reference bit set, then it is given a second chance: The reference bit is cleared, and the FIFO search continues. If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table. If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass. If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement. As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely. This algorithm is also known as the clock algorithm, from the hands of the clock moving around the circular queue.

7.3.5 NRU Page Replacement Algorithm

The not recently used (NRU) page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the software level as well.
At a certain fixed time interval, the clock interrupt triggers and clears the referenced bit of all the pages, so only pages referenced within the current clock interval are marked with a referenced bit. When a page needs to be replaced, the operating system divides the pages into four classes:

- class 3. referenced, modified
- class 2. referenced, not modified
- class 1. not referenced, modified
- class 0. not referenced, not modified

Although it does not seem possible for a page to be not referenced yet modified, this happens when a class 3 page has its referenced bit cleared by the clock interrupt. The NRU algorithm picks a random page from the lowest category for removal. So out of the above four pages, the NRU algorithm will replace the not referenced, not modified. Note that this algorithm implies that a modified but not referenced (within last clock interval) page is less important than a not modified page that is intensely referenced.

7.4 Demand Paging

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a lazy swapper, although a pager is a more accurate term.

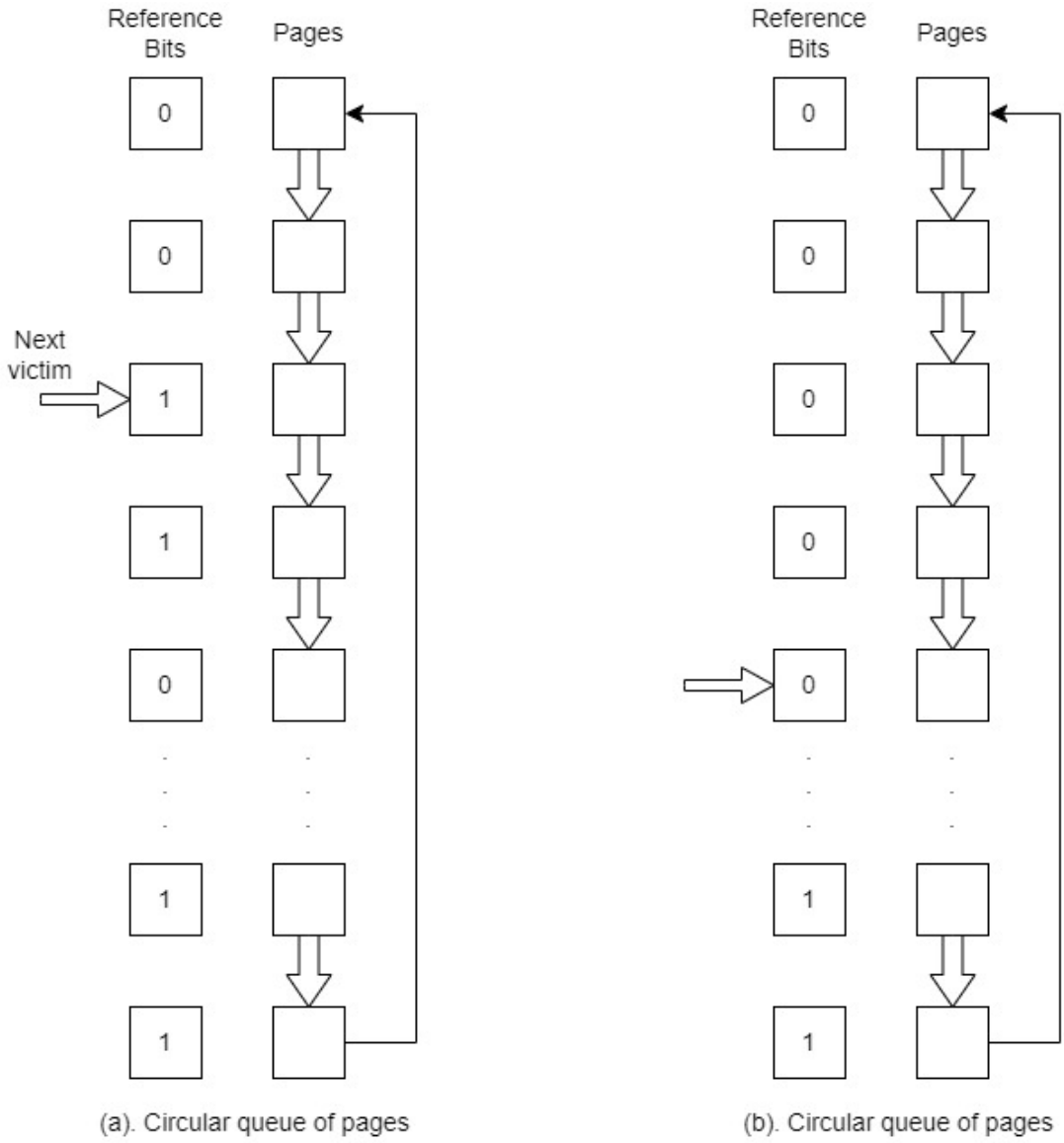


Figure 7.4: Second Chance Page Replacement Algorithm

Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)

If the process only ever accesses pages that are loaded in memory (memory resident pages), then the process runs exactly as if all the pages were loaded in to memory. If a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in a series of steps:

1. The memory address requested is first checked, to make sure it was a valid memory request.

2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.

3. A free frame is located, possibly from a free-frame list.

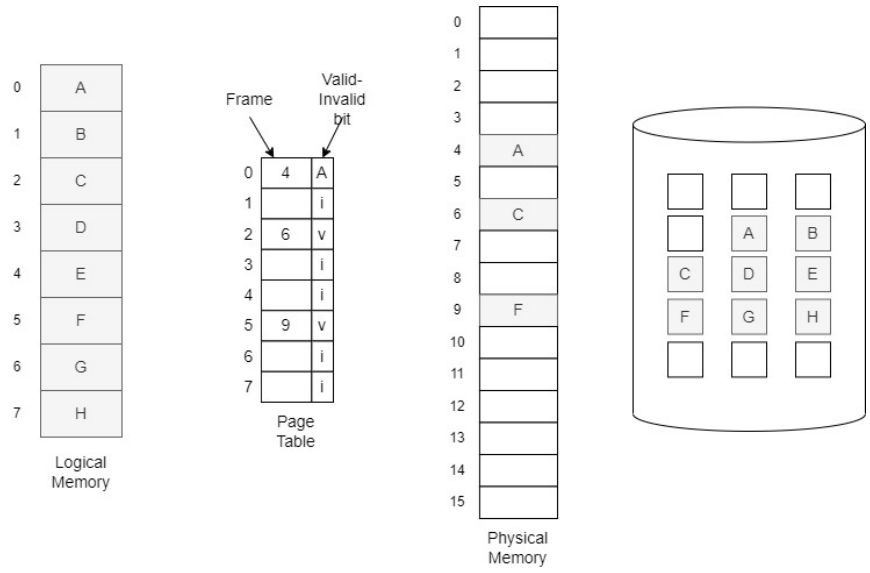


Figure 7.5: Demand Paging

4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)

5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.

6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as pure demand paging. In theory each instruction could generate multiple page faults. In practice this is very rare, due to locality of reference. The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory. A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

Conclusion: In conclusion, effective memory management is crucial for the performance and efficiency of operating systems. Page replacement algorithms play a vital role in managing the limited physical memory and optimizing the use of virtual memory. While the Optimal Page Replacement algorithm provides the theoretical best performance, practical implementations such as LRU and FIFO are commonly used due to their balance between simplicity and efficiency. The Second Chance and NRU algorithms offer additional strategies for handling page replacement, each with unique advantages. Understanding these

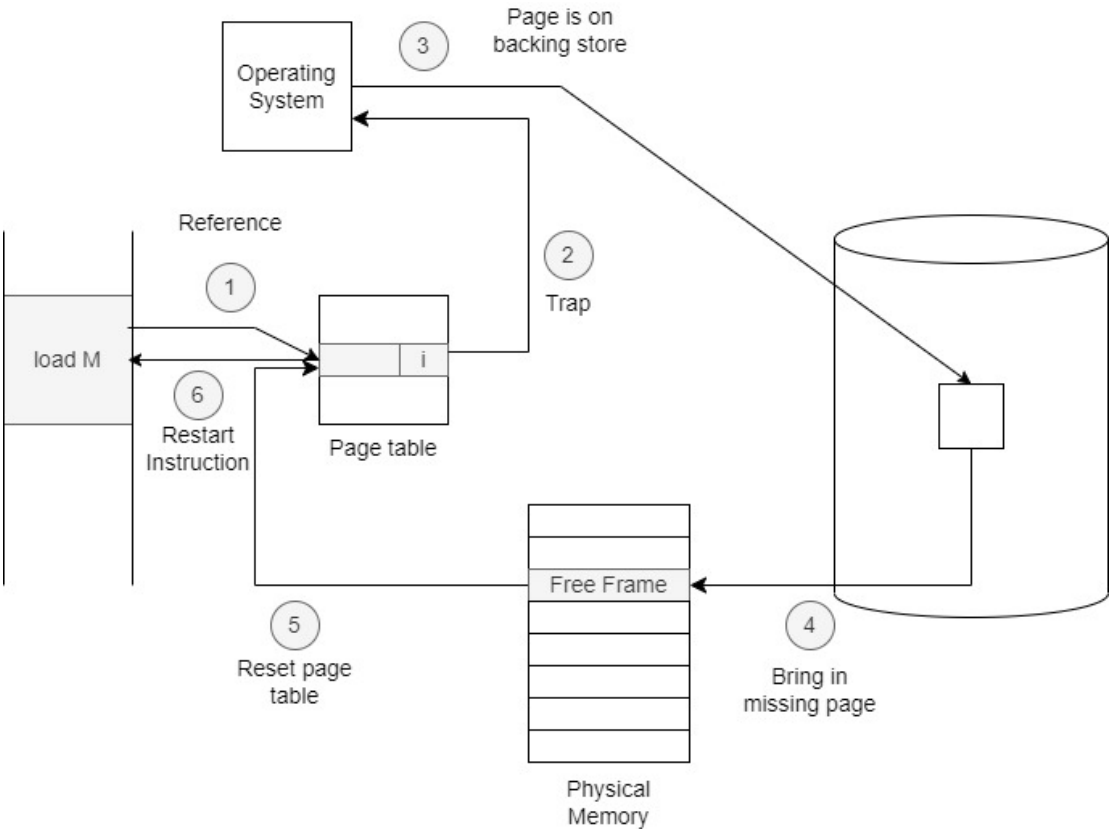


Figure 7.6: Steps in Handling Page Faults

algorithms and their implications helps in designing better operating systems that can handle modern computing demands efficiently.

7.5 Exercise

1. Discuss Demand Paging.
2. Write Second Chance LRU approximation page replacement algorithm in detail. Also write enhanced LRU approximation algorithm.
3. Consider the following page reference string:

   7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

   How many page faults would occur for following page replacement algorithm, considering 3 frames and 4 frames.

   i. FIFO

   ii. LRU

   iii. Optimal

4. Write about TLB.

5. What do you understand by thrashing?

6. Consider the following page reference string:

   7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.
   Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

i. LRU replacement

ii. Second chance LRU replacement

7.6 Multiple Choice Questions

1. The address generated by the CPU is referred to as

   (a) physical address
   (b) logical address
   (c) register address
   (d) Neither a nor b

2. Virtual memory allows _____

   (a) execution of a process that may not be completely in memory
   (b) a program to be smaller than the physical memory
   (c) a program to be larger than the secondary storage
   (d) execution of a process without being in physical memory

3. The address loaded into the memory address register of the memory is referred to as:

   (a) physical address
   (b) logical address
   (c) relative address
   (d) none of above

4. The run time mapping from virtual to physical addresses is done by a hardware device called the

   (a) Virtual to physical mapper
   (b) memory management unit
   (c) memory mapping unit
   (d) None of these

5. The size of a process is limited to the size of

   (a) physical memory
   (b) external storage
   (c) secondary storage

(d) None of these

6. Each entry in a Translation look-aside buffer (TL consists of):

   (a) Key
   (b) value
   (c) bit value
   (d) constant

7. If a page number is not found in the TLB, then it is known as a

   (a) TLB miss
   (b) buffer miss
   (c) TLB hit
   (d) TLB hit

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

DeepScience
Open Access Books

Chapter 8

Storage Management

Abstract: This chapter delves into various disk scheduling algorithms critical for optimizing disk I/O performance. It explores the foundational concepts and differences between algorithms like FCFS (First-Come, First-Served), SSTF (Shortest Seek Time First), SCAN, C-SCAN, LOOK, and C-LOOK. Each algorithm is analyzed for its operational principles, advantages, and disadvantages. Additionally, the chapter compares different RAID configurations, highlighting their fault tolerance, performance characteristics, and suitability for various applications. This comprehensive examination provides insights into the strengths and weaknesses of each approach, offering a nuanced understanding of their impact on system performance.

Keywords: Disk Scheduling, FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK, RAID, Fault Tolerance, Disk I/O Performance, Data Protection, Performance Overhead.

8.1 DISKS

Magnetic disks have been used as non-volatile storage for computer systems. Non-Volatile Storage Devices are storage devices that retain data even when they are turned off. Every computer needs a storage device.

Each disk platter is a flat, circular disc, similar to a CD. Hard disk drive components include the spindle, disk platter, actuator, actuator arm and read/write head. Typical platter diameters are 1.8-5.25 inches. Both sides of the platter are coated with a magnetic material. Data is stored on the platters magnetically.

A read / write head is placed just above each platter. The heads are connected to a disk arm, which moves all heads as one unit. The platter is logically divided into circular tracks, each track is sub divided into sectors. The set of tracks at one arm position forms a cylinder. In a disk drive, there may be thousands of circular cylinders, each track containing hundreds of sectors. The driver motor spins the disk at high speeds while in use. The most common speed is 15,000 rotations per minute.

The two important criteria for evaluating performance of disk are: **access time** and **disk bandwidth**.

Disk access time has two major components:

- Seek time: refers to the time it takes for the disk to move the heads to the cylinder that contains the sector you want.
- Rotational latency refers to the extra time it takes for your disk to rotate the sector you want around the disk head.

Disk bandwidth is the number of bytes transferred divided by the time elapsed between the initial request for service and the final transfer.

Seek time is the largest time than rotational latency and transfer time.

$$AccessTime = SeekTime + RotationalLatency + TransferTime \tag{8.1}$$

8.2 RAID

Redundant Array of Inexpensive Disks (RAID) is a data storage technology used to increase performance, reliability by combining several physical hard drives into one logical unit. The distribution and storage of data across these drives is dependent on the level or configuration of RAID. The RAID appears like a single disk to the software, while all RAIDs have the property that the data are distributed over the various drives, to allow parallel operation. RAID also use controlled redundancy (duplication of data) to achieve better reliability and availability. RAID is popularly used in High Performance Computing (HPC)(Wolf, 2014).

In the early days, there were only five common RAID levels, but over the years, there have been many variations are available, most common are shown in Figure 8.1:

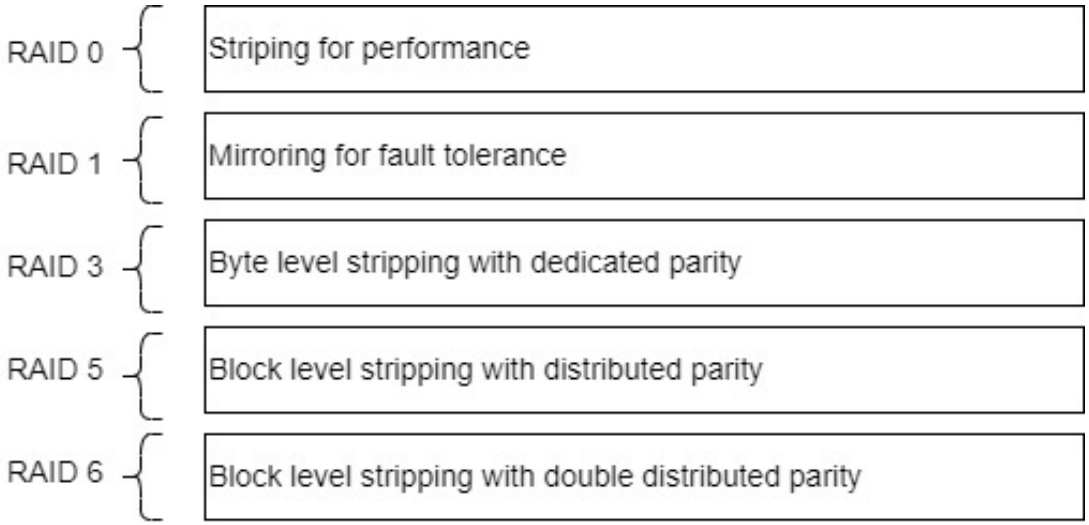


Figure 8.1: RAID levels

8.2.1 RAID 0

RAID 0 is based on **striping** but there is no mirroring and there is no parity. Stripping splits data across multiple disks which increase speed and capacity. Mirroring creates an exact copy of a disk on another disk. The capacity of a volume in RAID 0 is equal to the capacity of the total number of drives in a set. However, because striping distributes each file's content among all the drives, if any drive fails, the entire volume and all the files will be lost. RAID 0 level is shown in Figure 8.2.

The advantage of RAID 0, however, is that the read and write throughput of any file is increased by a factor equal to the number of units because, unlike in spanned vols, reads and write are done at the same time. However, the disadvantage of this is that the volume is more vulnerable to drive failures, since every drive in the RAID 0 setup causes the volume to fail, and the average volume failure rate increases as more units are added.

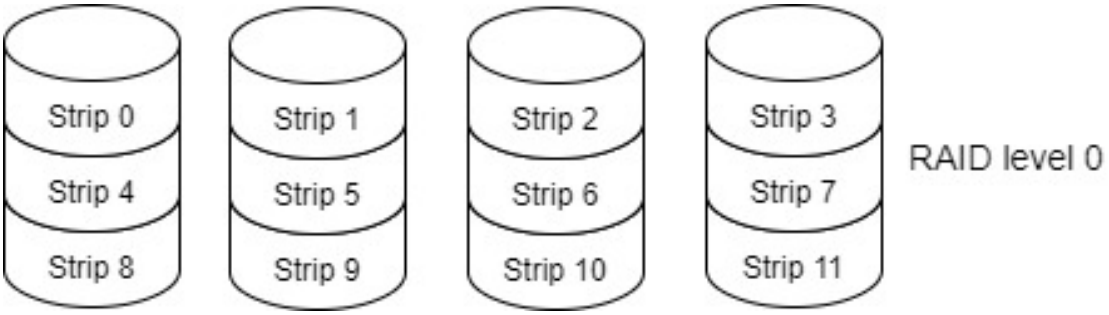


Figure 8.2: RAID 0

8.2.2 RAID 1

RAID 1 uses data mirroring without parity and striping. Same data is written on 2 or more drives. This creates a "mirrored set" of drives. Read requests can be served by any drive within the set. This improves read performance. Read throughput, if optimized by the controller or the software, is close to the sum of the throughput of all the drives within the set, as in RAID 0. Read throughput is slower than actual read throughput in most implementations of RAID 1. Write performance is slow because every strip is written twice. RAID 1 level is shown in Figure 8.3.

Figure 8.3: RAID 1

8.2.3 RAID 2

RAID 2 is a less commonly used RAID level compared to RAID 1 and others. It uses a technique called "bit-level striping with dedicated parity". In RAID 2, data is divided into individual bits, and each bit is written across multiple disks in the array in a systematic manner. Additionally, RAID 2 uses a separate dedicated disk (or disks) to store parity information. This parity information is used for error detection and correction in case of disk failure. RAID 2 is designed to provide high data transfer rates and error correction capabilities, primarily suitable for applications that require high throughput and data integrity, such as large-scale scientific computing or data-intensive processing tasks. RAID 2 structure is shown in Figure 8.4. Despite its advantages in certain scenarios, RAID 2 is not widely adopted due to the complexity and cost associated with the dedicated parity disks and specialized hardware required.

8.2.4 RAID 3

After RAID 0 and RAID1, RAID 2 specification was formed. Unlike RAID 0 and RAID 1, which operate on a strip-by-strip basis, RAID 2 works on a word-by-word, or even a byte-by-byte basis. It uses HAMMING code for error correction code.

RAID 3 is a simplified version of RAID level 2.

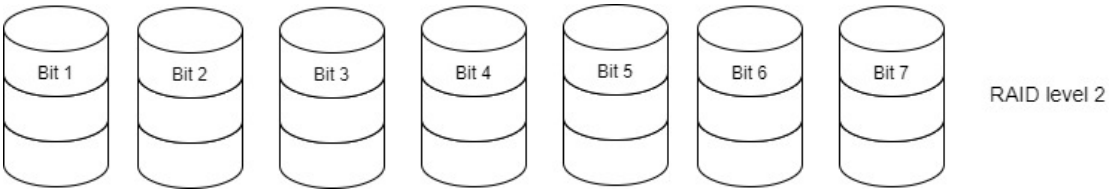


Figure 8.4: RAID 2

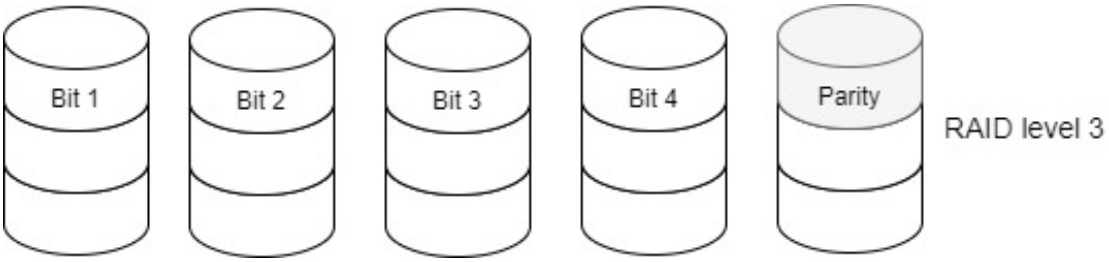


Figure 8.5: RAID 3

RAID 3 is based on the idea of byte level striping with dedicated parity. In this way, all disk spindle rotations are synchronized and data is striped so that each sequential bit is on a different disk drive. The parity is calculated across the corresponding bytes and then stored on the dedicated parity drive. While there are implementations of RAID 3, it is not widely used. The Figure 8.5 is depicting RAID 3 structure.

8.2.5 RAID 4

In RAID 4, data is striped across multiple disks, meaning it's divided into blocks and each block is written to a different disk. As shown in figure 8.6, RAID 4 uses a dedicated disk, called a parity disk, to store information that allows the system to recover data if one of the other disks fails. This means that even if one disk crashes, the data can still be reconstructed using the information on the parity disk. RAID 4 is like having several hard drives working together, with one disk keeping track of extra information to make sure your data stays safe even if one of the drives stops working.

8.2.6 RAID 5

In RAID 5, instead of having a dedicated parity disk like in RAID 4, the parity information is distributed across all the disks in the array. Each block of data is striped across the disks just like in RAID 4, but instead of having one disk dedicated solely to parity, the parity information for each stripe is distributed across all the disks(figure 8.7).

This distribution of parity information across all the disks in RAID 5 provides better performance for both reading and writing data compared to RAID 4 because multiple disks can work in parallel to perform these operations.

One important consequence of this distribution is that RAID 5 can withstand the failure of any single disk in the array without losing any data. When a disk fails, the data can be reconstructed using the parity information distributed across the remaining disks.

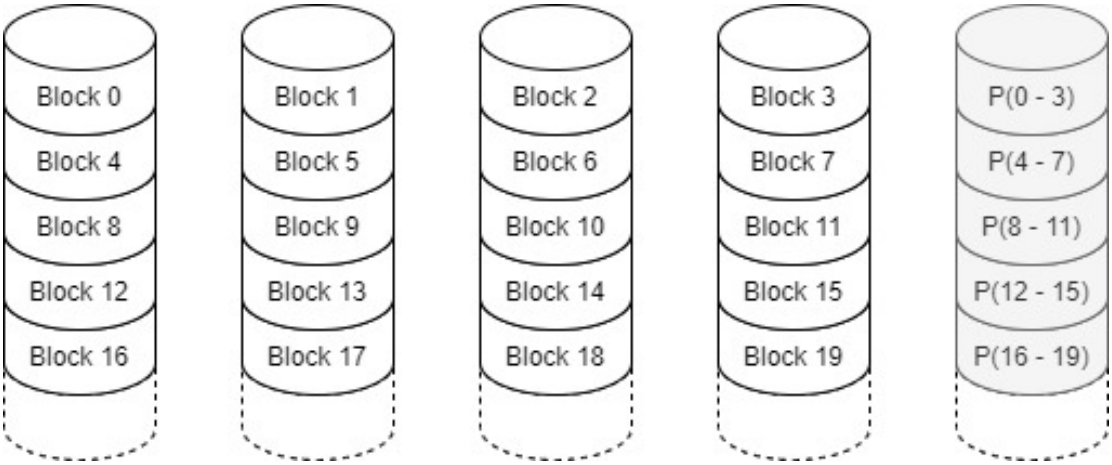


Figure 8.6: RAID 4

RAID 5 differs from RAID 4 in that it distributes parity information across all disks instead of having a dedicated parity disk. This provides better performance and fault tolerance, making RAID 5 a popular choice for many storage systems.

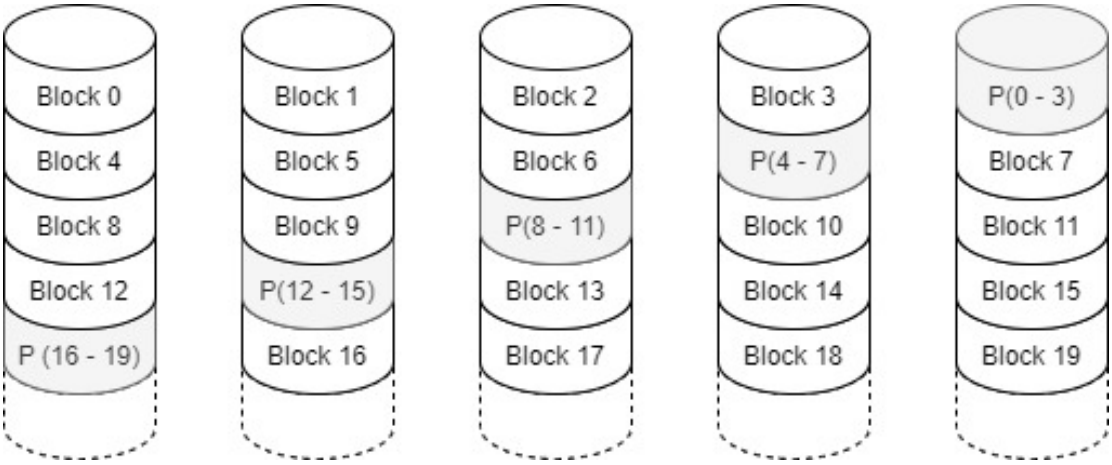


Figure 8.7: RAID 5

8.2.7 RAID 6

RAID 6, like RAID 4 and RAID 5, is a method for storing data across multiple hard drives in a way that improves performance and/or redundancy. However, RAID 6 offers an additional level of fault tolerance compared to RAID 5. In RAID 6, similar to RAID 5, data is striped across multiple disks, and parity information is also generated and stored. However, in RAID 6 as shown in figure 8.8, two sets of parity information are generated for each block of data, instead of just one as in RAID 5. This means that RAID 6 can withstand the failure of up to two disks simultaneously without losing any data. Even if two disks fail, the data can still be reconstructed using the parity information stored on the remaining disks.
The downside of RAID 6 compared to RAID 5 is that it requires more storage overhead

for storing the additional parity information, which can reduce the overall usable storage capacity of the array. In summary, RAID 6 provides higher fault tolerance compared to RAID 5, as it can withstand the failure of up to two disks in the array without losing data. However, it comes at the cost of higher storage overhead. RAID 6 is often chosen for critical data storage where data integrity and fault tolerance are paramount.

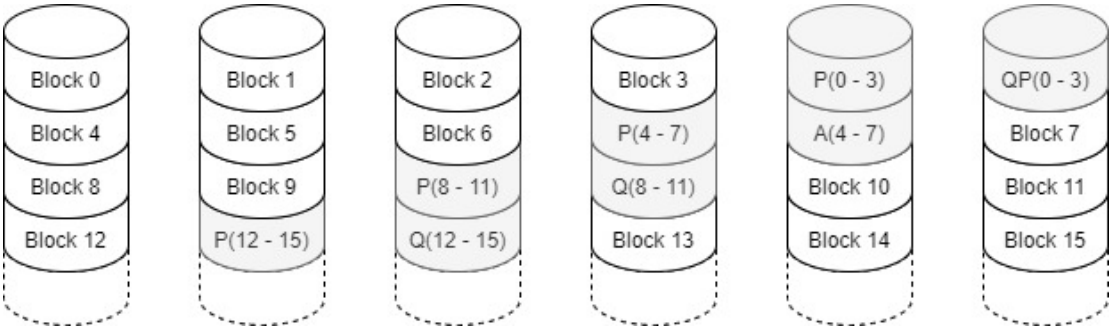


Figure 8.8: RAID 6

8.3 Disk Scheduling Algorithms

Many processes request to operating system for performing I/O(Input/Output). Disk scheduling algorithm decides which request will be handled next with satisfying fairness, speed and efficiency. Due to the time-consuming nature of seeking disk requests, the disk scheduling algorithm tries to minimize latency. **Latency** is the time that it takes to complete a single I/O operation on a block device.

Request is served immediately if the desired disk drive/controller is available. If the disk is busy, a new request is placed in a queue of pending requests. When a request is complete, the operating system has to decide which pending request to process next. Disk scheduling algorithm choose which specific disk request is next to process. The goal of using disk scheduling algorithms is to keep Head movements to a minimum. The smaller the head, the shorter the seek time. As the seek time is largest among others, scheduling algorithm tries to minimize it.

8.3.1 First Come First Served (FCFS)

This is the most basic type of disk scheduling algorithm. The requests are served or processed in the order in which they arrive. The request that arrives first is accessed and served first. As it it follows the arrival order, this scheduling algorithm observes large jumps from the innermost to the outermost track of the disk, and vice versa.

Example

For following disk requests, compute for the Total Head Movement of the read/write head:
95, 180, 34, 119, 11, 123, 62, 64
Consider that the read/write head is positioned at location 50.
The FCFS choose next request for location 95 and the head movement =95-50=45
next request for location 180 and the head movement =180-95=85
next request for location 34 and the head movement =180-34=146
next request for location 119 and the head movement =119-34=85

next request for location 11 and the head movement =119-11=108
next request for location 123 and the head movement =123-11=112
next request for location 62 and the head movement =123-62 =61
next request for location 64 and the head movement =64-62=2
The figure 8.9 shows head movement in FCFS.
Total Head Movement=45+85+146+85+108+112+61+2=644

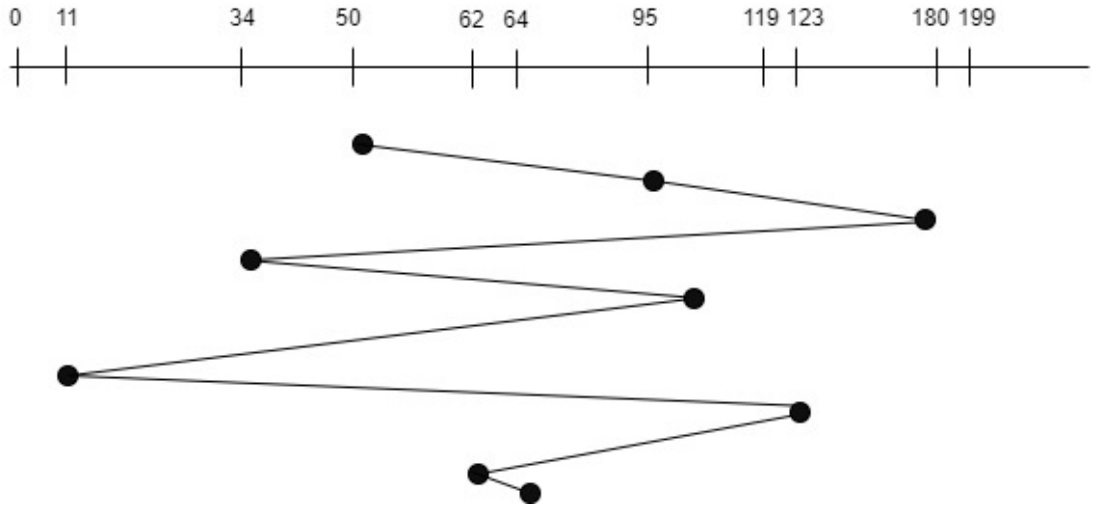


Figure 8.9: FCFS Disk Scheduling

Advantages

1. There is no indefinite delay in FCFS disc scheduling.

2. There is no starvation with FCFS disc scheduling because each request is given an equal opportunity.

3. FCFS is easy to implement and understand.

Disadvantage

1. Scheduling disc time in FCFS is not optimized.

2. FCFS services requests in the order they arrive, it may lead to longer seek times on average.

3. It may not make efficient use of the disk's bandwidth or minimize the amount of disk arm movement

8.3.2 Shortest Seek Time First Scheduling (SSTF)

SSTF is based on the concept that the head of the algorithm should go to the track that is nearest to its current position. This process will continue until all track requests have been processed.

Example

Consider the same example given in FCFS, the order of disk request is:
95, 180, 34, 119, 11, 123, 62, 64
Current position of head is 50.
SSTF will serve the closest request to head which is 62. Next it will serve 64 and then it will serve 34,11,95,119,123 and 180. The figure 8.10 shows head movement in SSTF.
Total head movement =(62-50)+(64-62)+(64-34)+(34-11)+(95-11)+(119-95)+(123-119)+(180-123)
=236

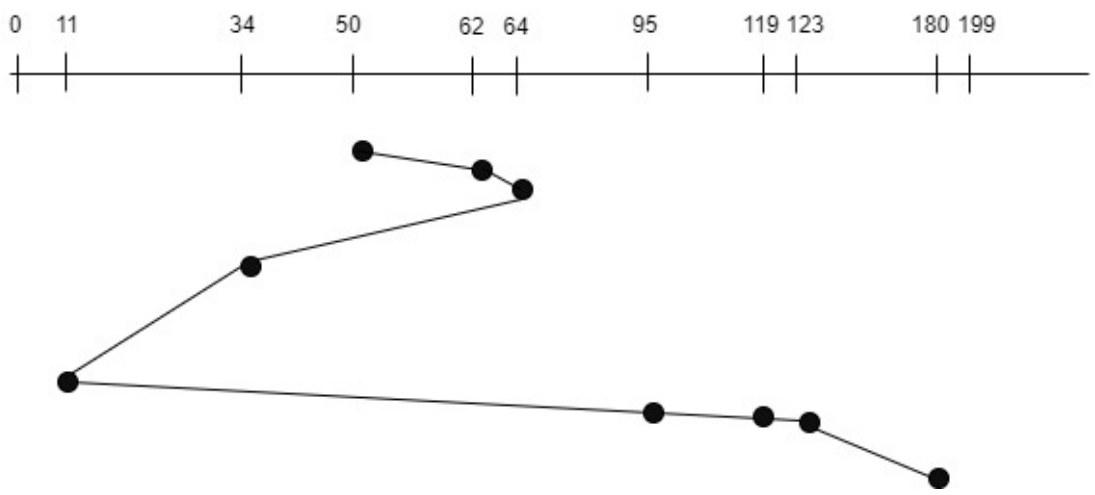


Figure 8.10: SSTF Disk Scheduling

Advantages

- SSTF aims to minimize the average seek time by servicing the request with the shortest seek time first. The average response time is less than FCFS.
- SSTF minimizes the distance the disk head needs to travel between consecutive requests, reducing mechanical wear and tear on the disk drive.
- Increased throughput.

Disadvantages

- Not the optimal algorithm.
- May cause starvation of some requests.
- High variance of response time as SSTF favours only some requests.

8.3.3 SCAN Scheduling

The SCAN algorithm starts with the disk arm at one end and moves it towards the other end. As it gets closer to each cylinder, the head moves back and forth, servicing requests. When it reaches the other end, it moves in the opposite direction and keeps servicing. The head keeps scanning back and forth across the disk. It is also called the **elevator algorithm**

because the disk arm works like a lift in a building. It services all the requests that go up and then turning around to service requests that go down.

Example

The requests for disk cylinders comes in following order:
98, 183, 37, 122, 14, 124, 65, and 67
And the disk arm is placed at 53 and it is moving toward 0.
The head will next service 37 (53-37=16) not 65(65-53=12), as it is moving towards 0. Then it will serve 14. Head will move at cylinder 0 after it. Then arm will move in reverse direction and move toward other end of the disk, servicing requests at 65, 67, 98, 122, 124, and 183.
The head movement in SCAN Total head movement = (53-37) + (37-14) + (14-0) + (65-0) + (67-65) + (98-67) + (122-98) + (124-122)+(183-124)
=236

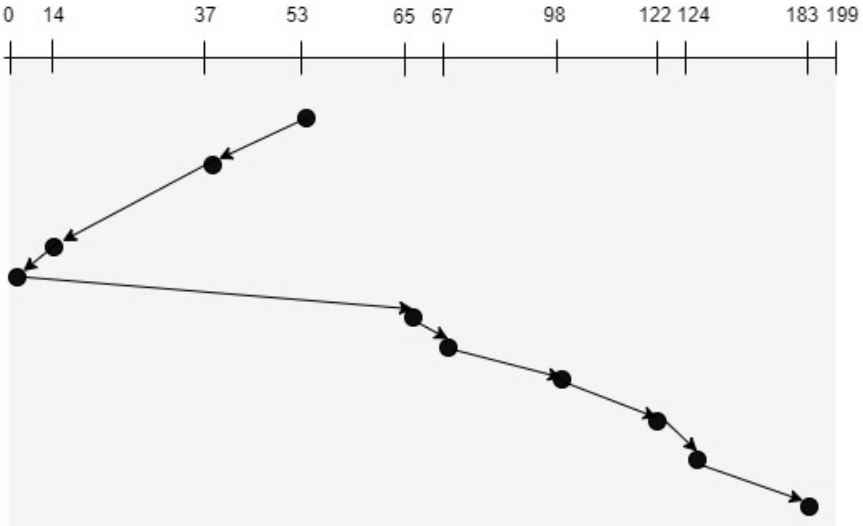


Figure 8.11: SCAN Disk Scheduling

Advantages

- SCAN aims to minimize the average seek time by moving the disk arm in one direction until it reaches the end of the disk and then reversing direction.
- SCAN provides fairness in servicing disk I/O requests by serving requests in the order they are encountered while traversing the disk surface.
- SCAN ensures a balanced response time for I/O requests by servicing requests in a systematic manner while traversing the disk surface.

Disadvantages

- SCAN can potentially result in starvation for requests located at the edges of the disk surface, particularly if there's a continuous stream of requests closer to the disk arm's current position.
- Long waiting time for requests for locations just visited by disk arm.

8.3.4 Circular SCAN (C-SCAN) scheduling

Circular scheduling is a variation of SCAN. It is designed to provide a more consistent wait time. Just as with SCAN, the head moves from one end of the disk to another, servicing requests on the way. The C-SCAN algorithm behaves differently when the head reaches the other end of the disk. It will not service requests on the way back like SCAN Algorithm. It immediately goes back to the start of the disk (figure 8.12). The scheduling algorithm of C-scans treats the cylinders like a circular list that wraps from the end of the cylinder to the first.

Example

For the previous example given in SCAN scheduling, C-SCAN will serve request in following order:
65,67,98,122,124,183,14 and 37. (After serving 183, the head will move directly to 0 and then the request for 14 will be served).

Total head movement = (65-53)+(67-65)+(98-67)+(122-98)+(124-122)+(183-124)+(199-183)+(14-0)+(37-14)
Total head movement = 137

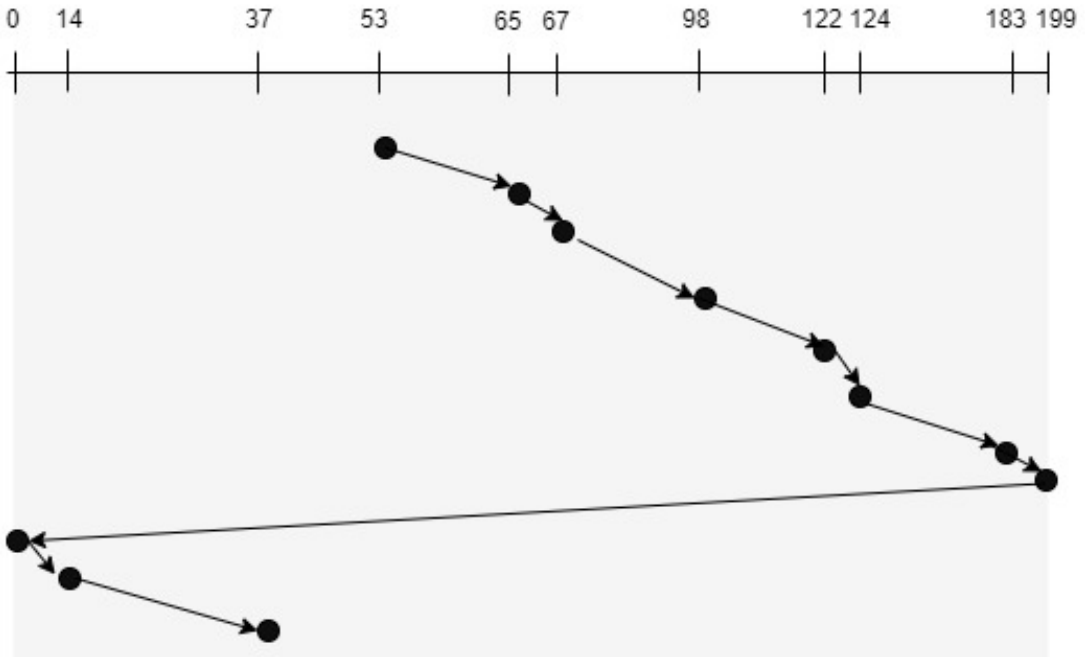


Figure 8.12: C-SCAN Disk Scheduling

Advantages

- By always moving the disk arm in the same direction (circularly), C-SCAN minimizes seek time, as it only scans in one direction without reversing the direction unless it reaches the end of the disk.

- Provides more uniform wait time compared to SCAN.

- C-SCAN scheduling can provide better performance compared to other disk scheduling algorithms like FCFS (First-Come, First-Served) or SCAN, especially in scenarios where most I/O requests are located near the edge of the disk.

- C-SCAN scheduling ensures fairness by treating all requests equally. Every request eventually gets serviced, preventing any request from being indefinitely delayed.

Disadvantages

- It may result in increased waiting time for I/O requests located on the opposite side of the disk's arm movement direction. Requests at the far end of the disk may experience longer wait times compared to those near the current position of the disk arm.

- It requires the disk arm to move circularly across the disk surface continuously. This constant movement may result in increased wear and tear on the disk drive components, potentially reducing the disk's lifespan.

- It may not be suitable for workloads with irregular access patterns or frequent requests distributed across different parts of the disk.

- It does not adapt to changes in the workload or disk access patterns dynamically.

8.3.5 LOOK Disk Scheduling Algorithm

LOOK is an advanced version of the SCAN scheduling algorithm. This algorithm provides slightly better lookup time than any of the other algorithms (FCFS, SRTF, SCAN and C-SCAN). LOOK processes requests same as SCAN, but it also looks ahead as if there are more tracks to service in that direction. If no pending requests are in the moving direction, the head will turn around and start servicing in the opposite direction. The LOOK algorithm is name as "look" because they look for a request before continuing to move in a given direction. The LOOK performs better than SCAN because in LOOK algorithm, the head does not move until the disk is at the end (figure 8.13).

Example

For example: if request comes in order 176, 79, 34, 60, 92, 11, 41, 114 and initial head position is 50. The first request will be handled for 60. Then R/W head moves to 79. After that request handled in following order: 92, 114, 176, 41, 34, 11.

Total head movement = (60-50) + (79-60) + (92-79) + (114-92) + (176-114) + (176-41) + (41-34) + (34-11) =291

Advantages

- LOOK minimizes unnecessary movement of the disk arm by only scanning as far as necessary to satisfy pending I/O requests. Unlike the SCAN algorithm, which always scans the entire disk in both directions, LOOK stops scanning in a direction when no pending requests are left in that direction.

- It provides better response time compared to the SCAN algorithm because it avoids unnecessary sweeps of the disk in both directions. This leads to faster processing of I/O requests and reduced waiting times for the requests.

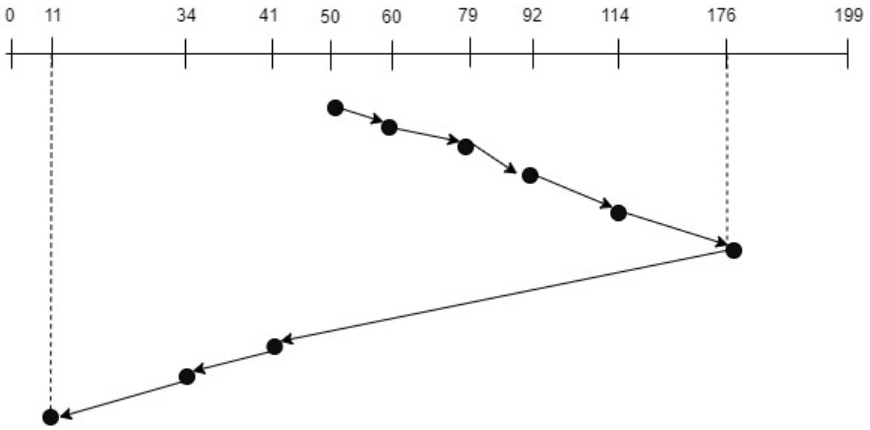


Figure 8.13: LOOK Disk Scheduling

- It optimizes disk utilization by scanning only the portions of the disk where pending I/O requests exist. It does not waste time scanning empty portions of the disk, which can improve overall system throughput.

- LOOK ensures fairness by treating all pending I/O requests equally. It services requests in the order of their arrival, preventing any request from being indefinitely delayed.

Disadvantages

- Similar to the SCAN algorithm, LOOK may still result in starvation for I/O requests located at the extremes of the disk's range. Requests located farthest from the current position of the disk arm may experience longer wait times, especially if there is a continuous stream of requests in the opposite direction.

- It does not dynamically adapt to changes in the workload or access patterns. It follows a predefined scanning pattern based on pending I/O requests at the time of invocation. This lack of adaptability may lead to sub optimal performance in certain scenarios.

- It may not be ideal for workloads with highly irregular access patterns or a mix of short and long-distance requests.

8.3.6 C-LOOK (Circular LOOK) Disk Scheduling Algorithm:

C-LOOK is an improved version of both the SCAN algorithm and the LOOK disk scheduling algorithm. This algorithm also wraps the tracks in a circular cylinder like the C-SCAN algorithm, but the seek time is faster than C-SCAN. As we know, C-SCAN is designed to prevent starvation and services all requests more evenly, the same is true for C-Look. In C-Look, the head only services requests in one direction (left or right), until all the requests in that direction are not being serviced. We can observe in figure 8.14, it jumps to the furthest request on the opposite direction and services the remaining requests. This not only provides a more uniform service, but also avoids wasting seek time by going to the end of disk.

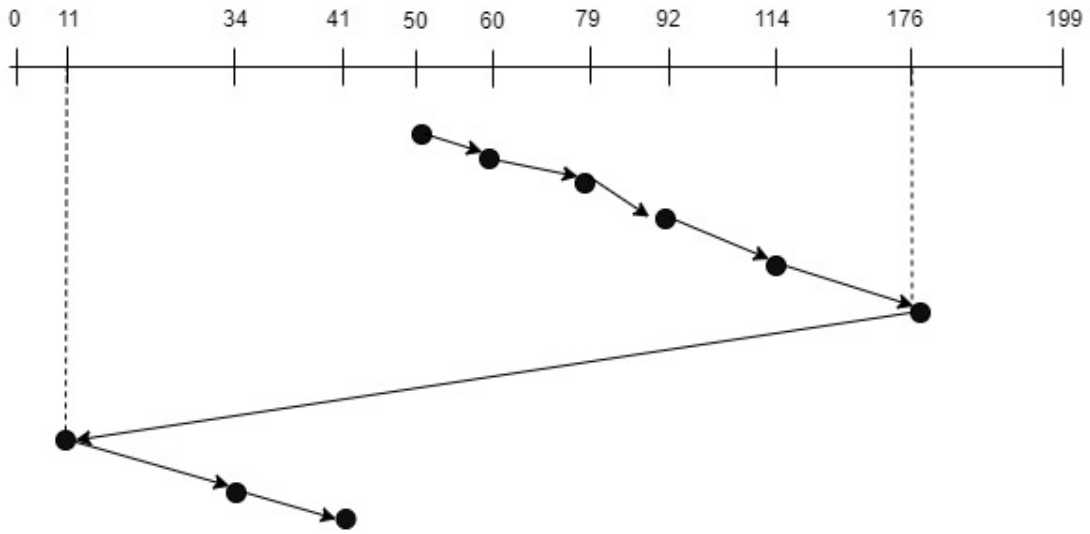


Figure 8.14: C-LOOK Disk Scheduling

Example

We can take same example given in LOOK scheduling. The C-LOOK will handle request of 60, 79, 92, 114, 176 in same order like LOOK scheduling. Then it will handle request of 11 as it is closest to other side. Then it will handle request of 34 and 41.
Total head movement = (60-50) + (79-60) + (92-79) + (114-92) + (176-114) + (176-11) + (34-11) + (41-34) =321

Advantages

In C-LOOK the head does not have to move till the end of the disk if there are no requests to be serviced.

- There is less waiting time for the cylinders which are just visited by the head in C-LOOK.
- C-LOOK provides better performance when compared to LOOK Algorithm.
- Starvation is avoided in C-LOOK.
- Low variance is provided in waiting time and response time.

Disadvantages

- In C-LOOK an overhead in servicing end requests is present.

Conclusion: In conclusion, the chapter provides a thorough analysis of several disk scheduling algorithms and RAID configurations, emphasizing their significance in storage management systems. The comparison between FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK algorithms highlights the trade-offs in terms of seek time optimization, fairness, and potential for request starvation. Meanwhile, the detailed discussion on RAID levels underscores the balance between performance and data protection. By understanding these

mechanisms, system administrators can make informed decisions to enhance disk performance and reliability in various computing environments.

8.4 Exercise

1. Explain the concept of disk storage and its role in storage management systems. Discuss the key characteristics and components of disk storage systems.

2. Compare and contrast the various RAID levels (0, 1, 2, 3, 4, 5, and 6) in terms of their architectures, fault tolerance, performance, and suitability for different applications.

3. Describe the RAID 0 configuration in detail. Discuss its advantages, disadvantages, and scenarios where it is commonly used. How does RAID 0 improve performance?

4. Analyze the RAID 1 configuration, highlighting its redundancy features and performance implications. What are the benefits and limitations of RAID 1 compared to other RAID levels?

5. Investigate RAID 5 and RAID 6 configurations, focusing on their fault tolerance mechanisms and performance characteristics. How do they differ in terms of data protection and performance overhead?

6. Explain the concept of disk scheduling algorithms and their significance in optimizing disk I/O performance. Discuss the FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK disk scheduling algorithms, highlighting their strengths and weaknesses.

7. Compare and contrast the FCFS and SSTF disk scheduling algorithms. What are the major differences in their approaches to disk I/O optimization? Provide examples to illustrate their usage scenarios.

8. Discuss the SCAN disk scheduling algorithm in detail, including its operation principle and performance characteristics. How does SCAN scheduling mitigate the disk head movement overhead compared to other algorithms?

9. Evaluate the C-SCAN disk scheduling algorithm's advantages and disadvantages compared to SCAN scheduling. In what scenarios is C-SCAN scheduling more suitable for disk I/O optimization?

10. Explain how the LOOK disk scheduling algorithm differs from the SCAN algorithm. What are the key differences in their approaches to disk head movement optimization? Provide examples to illustrate their respective benefits.

11. A computer has four page frames. The time of loading, time of last access
    and the R and M bit for each page given below:
    Which page NRU, FIFO, LRU will replace.

12. Consider the following page reference string:

Page	Loaded	Last Ref.	R	M
0	126	280	1	0
1	230	265	0	01
2	140	270	0	0
3	110	285	1	1

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
How many page faults would occur for the following replacement algorithms, assuming four frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.

- LRU replacement
- FIFO replacement

13. Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is:
86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130
Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the following disk scheduling:

- FCFS
- SCAN

14. Disk requests come in to the disk for cylinders 10, 22, 20, 2, 40, 6 and 38. A seek takes 6 msec per cylinder move. How much seek time is for Closest cylinder next algorithm? Initially arm is at cylinder 20.

8.5 Multiple Choice Questions

1. Which of the following is NOT a characteristic of disk storage systems?
   (a) Volatile storage
   (b) Random access
   (c) Non-volatile storage
   (d) Sequential access
2. Which RAID level provides improved performance through data striping without redundancy?
   (a) RAID 0
   (b) RAID 1
   (c) RAID 5

(d) RAID 6

3. RAID 1 is primarily known for:

   (a) Disk striping with parity
   (b) Mirroring of data
   (c) Distributed parity calculation
   (d) Data striping across multiple disks

4. RAID 5 and RAID 6 are similar in that they both:

   (a) Utilize mirroring for data redundancy
   (b) Use parity to provide fault tolerance
   (c) Incorporate disk striping without parity
   (d) Have no redundancy features

5. Which disk scheduling algorithm may result in starvation for some requests?

   (a) First Come First Served (FCFS)
   (b) Shortest Seek Time First (SSTF)
   (c) SCAN Scheduling
   (d) Circular SCAN (C-SCAN) scheduling

6. Which disk scheduling algorithm optimizes disk I/O by servicing requests closest to the disk head's current position?

   (a) FCFS
   (b) SSTF
   (c) SCAN
   (d) LOOK

7. Which RAID level offers the highest level of fault tolerance?

   (a) RAID 0
   (b) RAID 1
   (c) RAID 5
   (d) RAID 6

8. C-LOOK scheduling is an enhancement over which other disk scheduling algorithm?

   (a) FCFS
   (b) SSTF
   (c) SCAN
   (d) LOOK

9. Which RAID level provides both data striping and parity for fault tolerance?

   (a) RAID 0
   (b) RAID 1

(c) RAID 5

(d) RAID 6

10. Which disk scheduling algorithm may cause excessive head movement leading to reduced performance on certain workloads?

(a) FCFS

(b) SSTF

(c) SCAN

(d) LOOK

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 9

File Management

Abstract: This chapter explores various file allocation techniques used in operating systems, focusing on contiguous, linked, and indexed allocation methods. Each technique is analyzed for its implementation simplicity, performance, and impact on disk fragmentation and space utilization. The chapter provides a comprehensive comparison, highlighting the advantages and disadvantages of each method. Contiguous allocation, with its straightforward management and minimal overhead, is balanced against the challenges of external fragmentation and file size limitations. Linked allocation, which mitigates some fragmentation issues, introduces performance drawbacks due to pointer overhead. Indexed allocation offers efficient file access and reduced fragmentation but at the cost of additional overhead for small files. The discussion is aimed at providing insights into the trade-offs involved in choosing appropriate file allocation strategies for different operating systems.

Keywords: File Allocation Techniques, Contiguous Allocation, Linked Allocation, Indexed Allocation, Disk Fragmentation, Operating Systems, File System Management, Performance Overhead, Disk Space Utilization, File Growth Management.

9.1 File Concept

Imagine a file as a digital container or a virtual folder that can hold different types of information. Just like you use physical folders to organize your papers, a file is a way to organize and store information on a computer. A file can be used to store many things, such as text documents (like letters and essays), pictures, music, videos, or even computer programs. It's like having a special place on your computer where you can keep all your important stuff neatly organized.

A file is a collection of data or information that is stored together under a single name and is managed by the operating system. It serves as a basic unit for organizing and storing data on a storage device such as a hard drive or SSD.

A file is the smallest unit of secondary storage(figure 9.1) (Arpaci-Dusseau and Arpaci-Dusseau, 2018). Operating systems use file systems to organize files on storage devices like hard drives, solid-state drives, or network storage. The file system provides a structured way to store, retrieve, and manage files efficiently. Different operating systems use various file systems, but they all follow similar principles for file organization.

Figure 9.1: Files

9.1.1 File Types

Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined structure, which depends on its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute. The following types of files are common for different uses:

1. Text Files (.txt): Plain text files containing human-readable text without any formatting or special features. They are widely used for storing textual information and can be opened by basic text editors.

2. Document Files (.docx, .pdf, .odt): These files store formatted text, images, and other multimedia elements. Popular formats include Microsoft Word (.docx), Portable Document Format (.pdf), and OpenDocument Text (.odt).

3. Spreadsheet Files (.xlsx, .csv): Used for organizing tabular data, calculations, and formulas. Common formats include Microsoft Excel (.xlsx) and Comma-Separated Values (.csv).

4. Presentation Files (.pptx, .odp): Contain slides with text, images, and multimedia for creating presentations. Formats include Microsoft PowerPoint (.pptx) and OpenDocument Presentation (.odp).

5. Image Files (.jpg, .png, .gif, .bmp): Store digital images with various levels of compression and quality. Common formats include JPEG (.jpg), Portable Network Graphics (.png), Graphics Interchange Format (.gif), and Bitmap (.bmp).

6. Audio Files (.mp3, .wav, .flac): Store audio recordings, music, and sound. Common formats include MP3 (.mp3), WAV (.wav), and Free Lossless Audio Codec (.flac).

7. Video Files (.mp4, .avi, .mkv): Store video recordings and movies. Formats include MPEG-4 (.mp4), Audio Video Interleave (.avi), and Matroska Video (.mkv).

8. Archive Files (.zip, .rar, .tar): Compressed files that can contain one or more files and folders. They are used for efficient storage and distribution. Common formats include ZIP (.zip), RAR (.rar), and TAR (.tar).

9. Database Files (.db, .sqlite, .mdb): Store structured data in a format that allows efficient storage, retrieval, and manipulation. Examples include SQLite (.sqlite) and Microsoft Access Database (.mdb).

10. Program Files (.exe, .app, .sh): Executable files that contain instructions for a computer to run a program. Common formats include Windows Executable (.exe), macOS Application (.app), and Shell Script (.sh).

11. Font Files (.ttf, .otf): Store font data used for displaying text in various styles. Formats include TrueType Font (.ttf) and OpenType Font (.otf).

These are just a few examples of the many file types used in computing. Each file type serves a specific purpose and is associated with particular software or applications designed to read, display, or manipulate the data they contain.

9.1.2 File Attributes

File systems use various file attributes to store and manage information about files. File attributes are important metadata about files about each file, including its type, size, permissions, timestamps, and many more. Here are some common file attributes used by file systems:

- File Name: The name of the file, which serves as its unique identifier within a directory.
- File Extension: A part of the file name that follows a period (.) and indicates the file type or format.
- File Size: The size of the file in bytes or kilobytes, indicating the amount of data it contains.
- File Type: An identifier that specifies the type of file, such as text, image, audio, video, executable, etc.
- File Location: The physical location of the file's data on the storage device, usually specified by a block or cluster address.
- File Permissions: Access rights that control who can read, write, or execute the file. Common permission levels include read, write, execute, and special permissions like setuid, setgid, and sticky bit.
- File Owner: The user account or group that owns the file and has certain rights to modify its permissions.
- Timestamps: Records of various time-related information associated with the file, including:
  - Creation Time: The time when the file was initially created.

- Modification Time: The time when the file's content was last modified.
- Access Time: The time when the file was last accessed (read).
- Metadata Change Time: The time when the file's metadata (e.g., permissions) was last modified.

Different operating system and file system uses additional attributes that provide specific information about the file, such as read-only, hidden, archive, compressed, encrypted, etc. The file attributes are essential for data integrity, access restriction, and file administration. They facilitate effective file organization, protection, and retrieval for the operating system and file system.

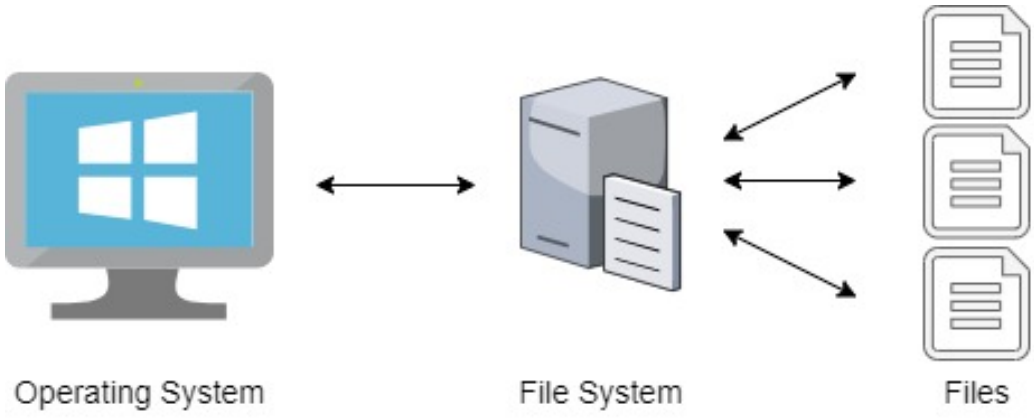


Figure 9.2: File System

9.1.3 File Operation

File systems offer a variety of actions to effectively handle files and directories. Users and programs can create, read, write, delete, and change files and directories using these operations. Following operations are supported on files:

1. Create: This operation allows users or applications to create new files and directories. When creating a file, the user specifies the name and location within the directory structure.

2. Read: The read operation enables users and applications to retrieve the content of a file. It allows them to access and read the data stored in the file.

3. Write: With the write operation, users and applications can modify the content of a file. It allows them to add, change, or delete data within the file.

4. Delete: The delete operation allows users and applications to remove files and directories from the file system. Once deleted, the file's data is typically marked as free space, and the file's directory entry is removed.

5. Open: The open operation is used to establish a connection between the file and the requesting process or application. Once a file is opened, the process can perform read and write operations on the file.

6. Close: It is used to terminate the connection between the file and the requesting process. After closing a file, the process can no longer perform read or write operations on it.

7. Rename: It allows users and applications to change the name of a file or directory within the file system.

8. Seek: It is used to shift the file pointer to a specific location within the file, use the seek operation. This makes it possible to access random sections of the file.

9. Truncate: It allows users and applications to reduce the size of a file. By doing this, the file's extra data is removed from the end.

10. Set Permissions: File systems provide operations to set access permissions on files and directories. Access permission indicates who can read, write, and execute files.

11. Copy: The copy operation allows users and applications to duplicate a file, creating a new file with same content.

12. Move: This operation is used to move a file or directory from one location to another location within the file system.

9.2 File Allocation Techniques

File systems employ file allocation techniques to allocate storage space for storing files. For best storage utilization and access speed, these techniques manage how files are physically kept and arranged on disc. There are various file allocation strategies, each with pros and cons. The primary file allocation methods are:

1. Contiguous Allocation

2. Linked Allocation

3. Indexed Allocation

9.2.1 Contiguous Allocation

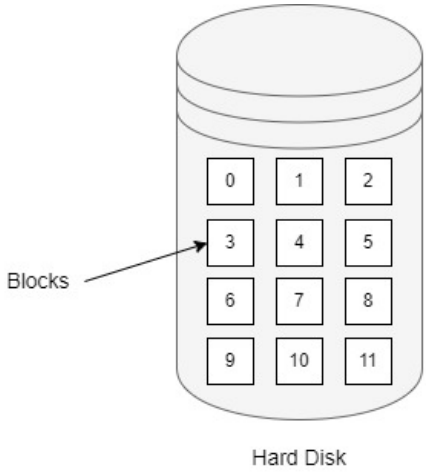
In contiguous allocation, each file is stored in a continuous block of disk space. When you create a file, the operating system looks for a big enough block of space to hold the entire file. This method allows you to access files quickly because the whole file is saved in one disk block. On the other hand, it can cause file fragmentation as you create and delete files, leaving you with tiny chunks of disk space that you may not be able to use for larger files.

Contiguous allocation is a method of allocating disk space for files where each file occupies a set of consecutive disk blocks.

Example: Let's say we have three files: abc.text, videos.mp4, and jtp.docx. Each file needs a certain number of blocks to store its data.

1. abc.text requires 3 blocks.

2. videos.mp4 requires 2 blocks.



File Name	Start	Length	Allocated Blocks
abc.text	0	3	0, 1, 2
video.mp4	4	2	4, 5,
jtp.docx	9	3	9, 10, 11

Directory

Figure 9.3: Contiguous Allocation

3. jtp.docx requires 3 blocks.

The disk space allocation for these files using contiguous allocation can be done in following way:

1. We start by allocating the first available 3 consecutive blocks(0,1,2) on the disk for abc.text
2. Next, we allocate the next available 2 consecutive blocks(4,5) for videos.mp4.
3. Finally, we allocate the next available 3 consecutive blocks(9,10,11) for jtp.docx.(blocks 6,7,8 are already occupied by other file)

Figure 9.3 shows contiguous allocation for above files.

Advantages

1. Sequential Access: Contiguous allocation allows for efficient sequential access to files since the data blocks are stored adjacently on the disk. This can result in faster read and write operations, particularly for large files.
2. Simple Implementation: Contiguous allocation is straightforward to implement and manage. The operating system only needs to track the starting block and the size of each file, simplifying file system management.
3. Minimal Overhead: There is minimal overhead associated with file allocation and management since the operating system only needs to store information about the starting block and size of each file.
4. Reduced Disk Fragmentation: Contiguous allocation can help minimize disk fragmentation since files are stored in contiguous blocks. This can lead to improved disk performance over time.

Disadvantages

1. Fragmentation: While contiguous allocation minimizes internal fragmentation (unused space within a block), it can lead to external fragmentation (unused space scattered

throughout the disk). Over time, as files are created, deleted, and resized, free blocks become fragmented, making it challenging to allocate contiguous blocks for new files.

2. File Size Limitation: Contiguous allocation imposes a limitation on the maximum size of files that can be stored on the disk. If there is not enough contiguous free space available, large files cannot be accommodated, leading to file allocation failures.

3. Wastage of Space: Contiguous allocation can result in wasted space due to internal fragmentation. If a file does not perfectly fit into a block, the remaining space in that block is wasted, reducing overall disk utilization efficiency.

4. Difficulty in File Growth: It can be challenging to accommodate the growth of files stored using contiguous allocation. If a file needs to be expanded and there is not enough contiguous free space available after the existing allocation, the file may need to be relocated to a different area of the disk, leading to additional overhead and fragmentation.

9.2.2 Linked Allocation

Linked allocation uses pointers to connect non-contiguous blocks of disk space. Each block has a reference to the following block in the linked list. When you create or extend a file, the operating system looks for free blocks in the file and updates the pointers in the linked list accordingly. Linked allocation reduces fragmentation, but it also introduces overhead because it requires the storage of pointers for each block. This can result in slower access times because accessing a particular block requires going through the linked list.

In linked allocation, each file is a collection of linked disk blocks that do not have to be contiguous. Disk blocks can be distributed anywhere on disk.

Key features of linked allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- Simple access – need only starting address.
- No wastage of space.
- Random access is not allowed.
- It is used by MS-DOS and OS/2.

Example: For example file jeep requires 5 disk blocks start from block 9 and the last block of file will be stored in block 25, as shown in Figure 9.4.

Advantages

1. No Fragmentation: Linked file allocation eliminates the issue of fragmentation entirely since each block of a file can be located anywhere on the disk. This means that even if free blocks are scattered, they can still be utilized efficiently.

2. Flexibility in File Size: Linked allocation allows for files to grow dynamically as needed. New blocks can be added to a file without the need for contiguous space, making it suitable for handling files of varying sizes.

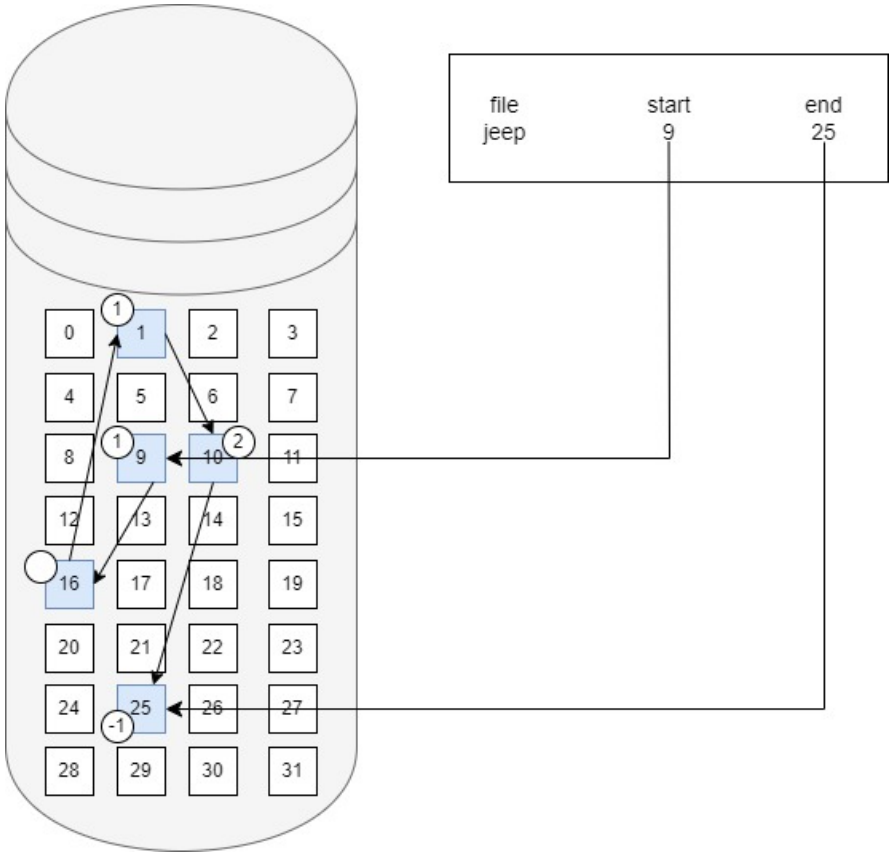


Figure 9.4: Linked Allocation

3. Efficient Disk Space Utilization: Linked allocation minimizes wasted space since each block is used optimally. There is no internal fragmentation, as each block is only as large as necessary to accommodate the data it contains.

4. Simple Implementation: Linked allocation is relatively simple to implement, as each block only needs to store a pointer to the next block in the file. This simplicity can lead to faster file system operations.

Disadvantages

1. Poor Performance: Linked allocation can lead to slower performance compared to other allocation methods, especially for large files or files with high levels of fragmentation. This is because accessing linked blocks requires following pointers, which can result in increased seek times.

2. Fragmentation of File Allocation Table (FAT): In systems that use a File Allocation Table (FAT) to keep track of file blocks, the FAT itself can become fragmented over time. This fragmentation can make it more difficult for the operating system to locate and access file blocks efficiently.

3. Wasteful of Disk Space: While linked allocation minimizes internal fragmentation, it can lead to wasted disk space due to overhead associated with storing pointers. Each block of a file requires additional space to store a pointer to the next block, which can add up, particularly for small files.

4. Difficulty in File Recovery: Linked allocation can complicate file recovery operations in the event of disk errors or corruption. If a pointer to a block becomes corrupted or lost, accessing the remaining blocks of the file can become challenging or impossible.

9.2.3 Indexed Allocation

In indexed allocation a specific block (index block) stores pointers to data blocks in memory. Each file contains an index block that contains pointers to all of the disk blocks that store the data in the file. The index block(9.4) serves as an indirect address for accessing the file's data blocks. This approach reduces file fragmentation and provides efficient file access because the index block is cacheable. However, it can introduce extra overhead, particularly for small files that need a full index block.

In indexed allocation, each file contains an index block. An index block contains the references to all the blocks of file.

One common implementation of indexed file allocation is the use of index nodes (i-nodes) in UNIX-like file systems. Each file in the file system has its own i-node, which contains metadata about the file, such as its size, permissions, and pointers to the data blocks.

Key features of indexed allocation:

- In indexed allocation, a data structure called an *i-node(index-node)*, which lists the attributes and disk addresses of the file's blocks
- Indexed allocation improves access time over linked allocation by using the index block to access file.
- Each file has its own index block, which is an array of disk-block addresses.
- To read and write the i^{th} block, the i^{th} entry in index-block is used. This is similar to paging scheme.
- The main advantage of this scheme compared to linked files using in memory table is that i-node only needs to be in memory at the time the file is opened.

Example: We have a file named "Pics" that needs to be stored on the disk. The file consists of 5 data blocks. First, allocate a block to serve as the index block for "Pics". This index block will contain pointers to the data blocks of the file. As shown in figure 9.5, block number 11 is used as indexed block for file. In next step Next, indexed allocation will allocate 5 data blocks for the file "Pics". The index block contains pointers to each data block allocated for "Pics".
When the operating system needs to access file, it first reads the index block to determine the locations of all the data blocks belonging to the file. Then, it can sequentially read each data block to retrieve the contents of the file.

Advantages

1. Efficient File Access: Indexed file allocation provides efficient file access because all the block pointers are stored in the index block. This allows for direct access to any block of the file without the need to traverse through the entire file.

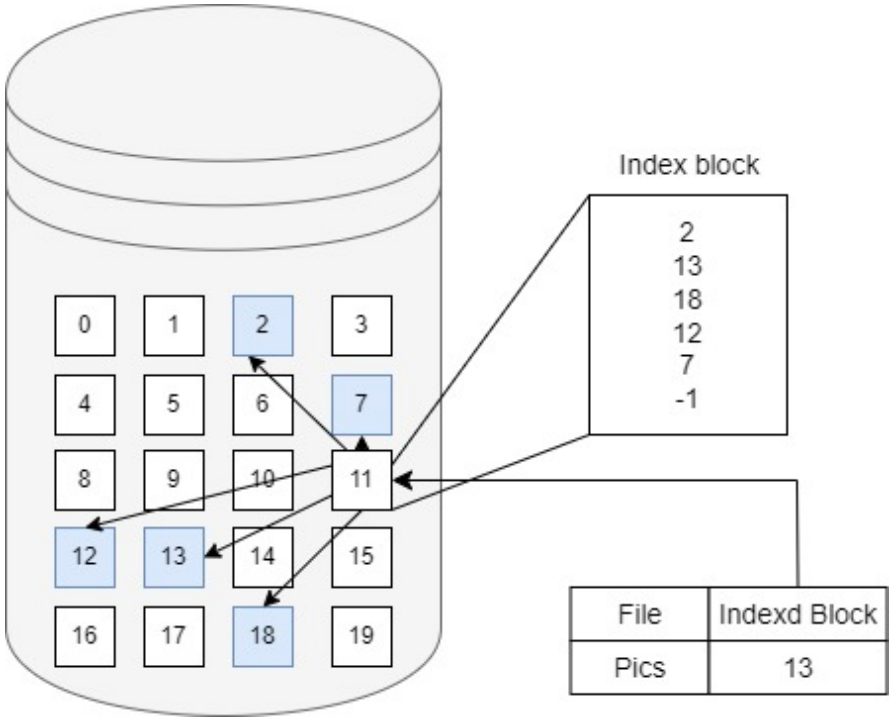


Figure 9.5: Indexed Allocation

2. Support for Large Files: Indexed allocation can efficiently handle large files as the index block can accommodate a large number of pointers to data blocks. This makes it suitable for managing files of varying sizes.

3. Minimized Fragmentation: Indexed allocation minimizes fragmentation since it does not require contiguous allocation of data blocks. Files can be stored non-contiguously on the disk, reducing the likelihood of fragmented free space.

4. Improved Performance: Compared to linked allocation, indexed allocation typically offers better performance for file access, especially for large files. It avoids the overhead of following pointers for each block access, resulting in faster read and write operations.

Disadvantages

1. Index Block Overhead: Indexed allocation incurs overhead in terms of the space required for the index block. For files with a small number of blocks, the overhead of the index block can be significant, leading to inefficient disk space utilization.

2. Limited Index Block Size: The size of the index block imposes a limit on the number of pointers it can contain. If the file exceeds this limit, indexed allocation may not be suitable, leading to potential inefficiencies or the need for alternative allocation methods.

3. Increased Complexity: Indexed allocation introduces additional complexity to file system management. The maintenance and updating of index blocks require more sophisticated algorithms compared to other allocation methods, increasing the complexity of file system operations.

4. Single Point of Failure: The index block serves as a single point of access for the file's data blocks. If the index block becomes corrupted or lost, it can result in the loss of access to the entire file, making file recovery challenging. This potential single point of failure can pose risks to data integrity and reliability.

9.3 File Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. File access methods are techniques or approaches used to read, write, and manipulate data stored in files. These methods vary based on the type of data, the structure of the file, and the requirements of the application using the file. Some common file access methods include:

1. Sequential Access
2. Direct /Random Access
3. Indexed Access

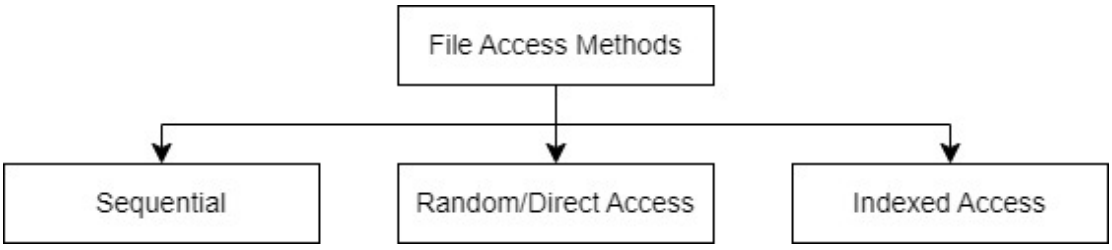


Figure 9.6: File Access Methods

9.3.1 Sequential Access

In sequential access, information in the file is processed sequentially, one record at a time. This approach is the most popular approach. Many editors and compilers typically use this approach. Reads and writes make up the bulk of the operations on a file. In sequential access, the operating system reads the file one word at a time. The pointer keeps track of the file's base address (starting address). If you want to read the first word in the file, the pointer gives you that word and increments its value by one word until the file is finished. Sequential file access is shown in figure 9.7

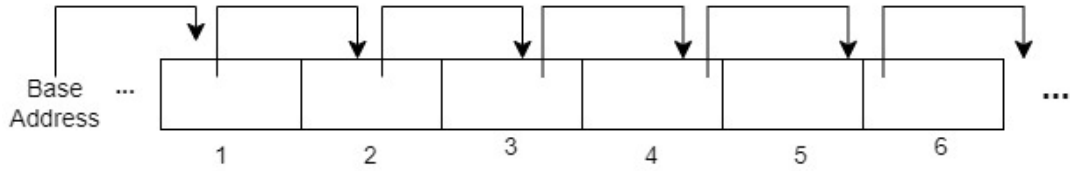


Figure 9.7: Sequential File Access

Advantages

1. It is very easy to implement.
2. It uses lexicographic (like dictionary) order to enable quick access to the next entry.
3. Sequential access typically incurs lower overhead compared to random access methods, such as indexed or direct access.
4. Sequential access is highly efficient for applications that process data sequentially, such as batch processing or data streaming.

Disadvantages

1. It works slow in case if the next file record to be accessed is not stored next to the currently pointed record.
2. To add a new record, you may need to move a large number of records from the file.
3. Sequential access imposes limitations on file modification operations, such as insertion or deletion of data at arbitrary positions within the file.

9.3.2 Direct Access

Another method is direct access (or relative access). A file is a collection of logical records that are of a fixed length. In this scheme, all records can be read and written quickly in any order. Direct-access is based on a file's disk model because a disk allows random access to any block of a file. Direct file access is shown in figure 9.8. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 3, then read block 8. There are no restrictions on the order of reading or writing for a direct-access file. Direct-access files provide immediate access to large volumes of data. Databases are a common example of this type of file. When a query comes in about a specific subject, it figures out which block has the answer and then directly read that block to get the information. For the direct-access method, the file operations must specify the block number.

Figure 9.8: Direct File Access

Advantages

1. Direct file access allows for efficient random access to specific portions of a file. Applications can directly access any block within the file without needing to read through the entire file sequentially, resulting in faster data retrieval times.

2. Direct file access provides flexibility for various file operations, including insertion, deletion, and modification of data at arbitrary positions within the file.

3. Direct file access is well-suited for interactive applications where users need to access specific data quickly.

4. Direct file access is optimized for indexed access methods, where the location of data within the file is determined by an index or key.

Disadvantages

1. Direct file access requires more complex implementation compared to sequential access methods. Managing file pointers, indexes, and ensuring data integrity can be challenging, especially in multi-user or multi-threaded environments.

2. Direct file access can incur higher overhead compared to sequential access methods, particularly for maintaining data structures such as indexes or pointers.

3. Direct file access can make files more susceptible to corruption or data loss if access operations are not properly managed.

9.3.3 Indexed File Access

Indexed file access is a method used in file systems to efficiently access and manage files by utilizing an index structure. In this approach, each file is associated with an index that contains pointers to the locations of the file's data blocks on the disk.

The index serves as a mapping between logical file addresses and physical disk addresses, allowing for quick and direct access to specific data within a file. When a file needs to be accessed, the index is consulted to locate the appropriate data blocks, eliminating the need to search through the entire file sequentially.

Advantages

1. Indexed access allows for fast random access to any part of a file, making it ideal for applications that require frequent and unpredictable access patterns.

2. The use of indexes provides flexibility for file manipulation operations, such as insertion, deletion, and modification of data, without the need for large-scale data movement.

3. Indexed access can help reduce fragmentation since files can be stored non-contiguously on the disk while still allowing for efficient access to data blocks.

Disadvantages

1. Maintaining and updating the index structure can incur overhead, particularly for large files or files with a high degree of fragmentation.

2. The size of the index may impose limits on the maximum size of files that can be efficiently managed using indexed access methods.

9.4 Directory

Directory is a collection of files and directories. A directory is a type of file system that stores information about other computer files and directories(figure 9.9). Files are organized by holding related files in one directory. In a tree-like file system (where files and directories are arranged in a tree-like structure), a subdirectory is a directory inside another directory.

A directory is a special type of file that serves as a container for other files and directories. It provides a hierarchical structure for organizing and managing files on a storage device such as a hard drive.

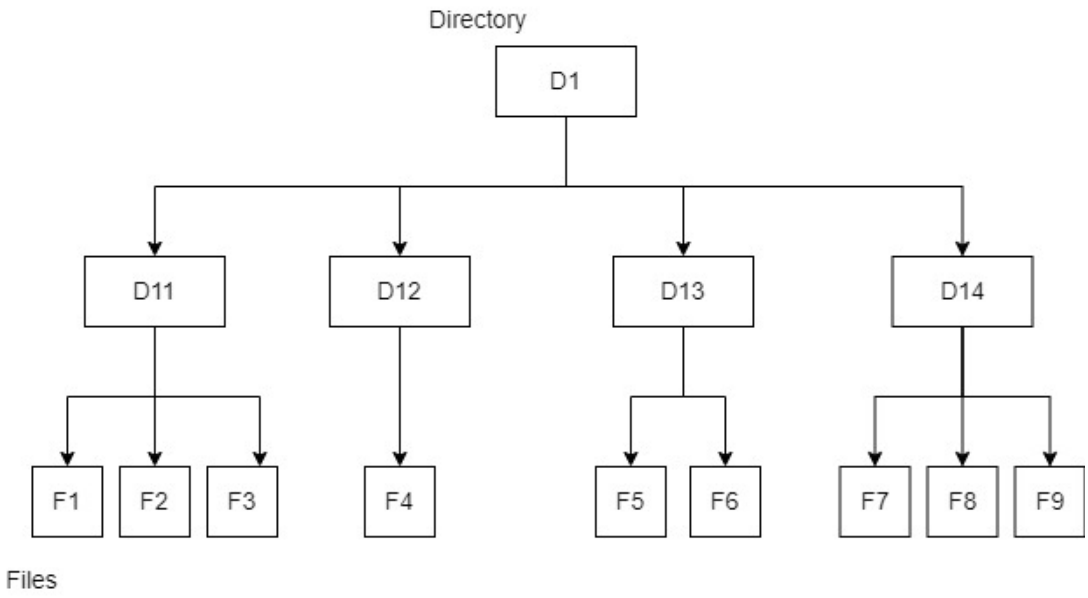


Figure 9.9: Directories

9.5 Directory Structure

9.5.1 Single Level Directory

Single-level directory is the simplest directory structure. All the files are in one directory(figure 9.10). It is simple to maintain and easy to understand. However, a single level directory has some limitations. For example, if you have a large number of files or if you have a system with multiple users. It will be difficult to ensure unique names for all the files in one directory.

Advantages

1. Easy implementation.

2. Efficiency: file searching is fast.

Disadvantages

1. Naming: name collision because two files can have the same name
2. Grouping: cannot group the same type of files together.

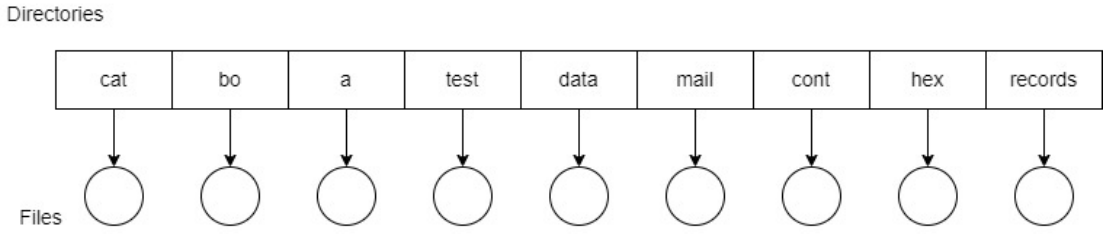


Figure 9.10: Single Level Directory

9.5.2 Two Level Directory

The problem with a single-level directory is that it often causes different users to have different file names. The standard way to handle this is to create separate directories for each user. In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user(figure 9.11). When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.

Advantages

1. Efficient searching
2. Naming - using same file name is possible in a different directory.

Disadvantages

1. Naming - collisions
2. Grouping capability – not possible except by user

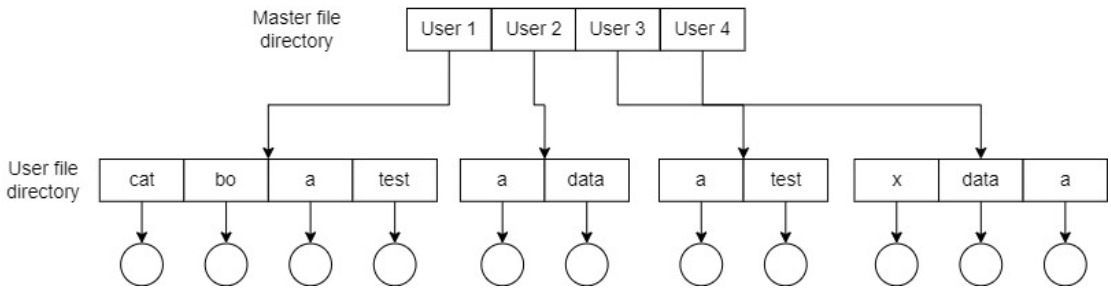


Figure 9.11: Two Level Directory

9.5.3 Tree Structured Directory

A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A real-life example of a tree-structured directory is the file system hierarchy in a computer operating system, such as Windows(Panek, 2019), macOS, or Linux. Figure 9.12 shows organization of files and directories in tree structured directory.

Tree-structured directories are like a family tree for organizing files and folders on a computer. It starts with a main folder (like "root") and branches into subfolders, which can further branch into more subfolders.

To manage files and directories in Tree structure directories, One bit in each directory entry defines the entry:

- as a file (0),
- as a subdirectory (1).

Path names can be of two types:

- An *absolute path* name begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A *relative path* name defines a path from the current directory.

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.

- For example, user can access a file of other user by specifying its path names.
- User can specify either an absolute or a relative path name.
- Alternatively, user can change her current directory to be other user's directory and access the file by its file names.

Advantages

1. Efficient searching – current working directory
2. Naming - same file name in a different directory
3. Grouping capability

Disadvantages

1. Structural complexity

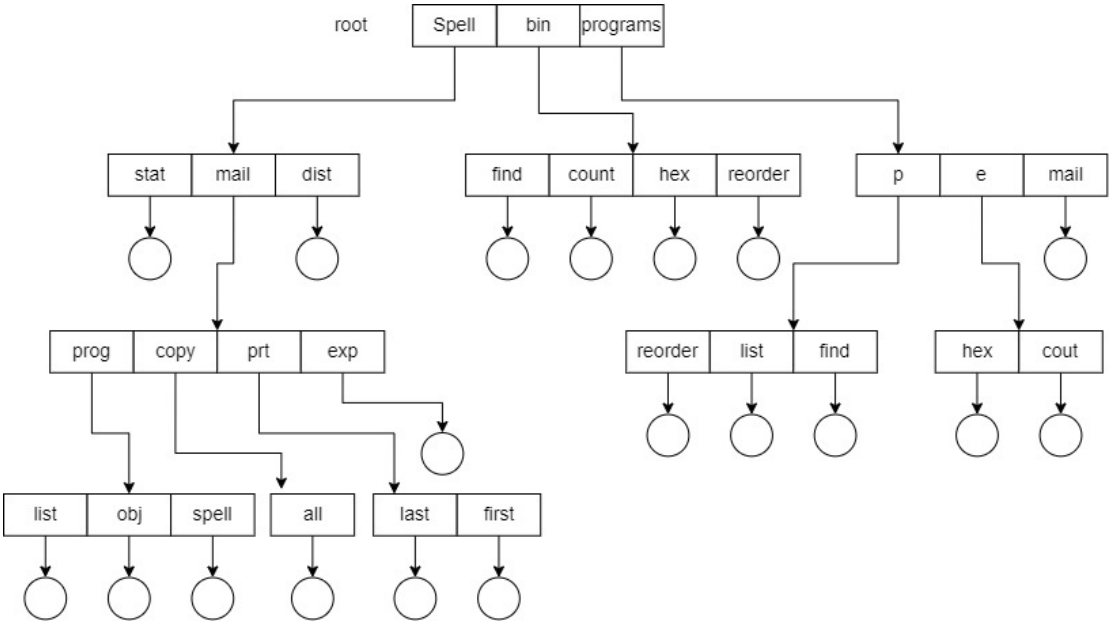


Figure 9.12: Tree Structured Directory

9.5.4 Acyclic Graph Directories

A tree structure prohibits the sharing of files or directories. An acyclic graph is a graph with no cycles. It allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme. It is important to note that a shared file (or directory) is not the same as two copies of the file.

Acyclic Graph Directories are structures where files or folders are organized in a way that no loops or cycles are allowed.

9.5.5 Advantages

1. Acyclic graph directories allow for more flexible organization than strictly tree-structured directories.
2. They can be more efficient for certain types of data organization.
3. Since files or folders can be linked in multiple ways without forming cycles, it's possible to avoid redundancy by creating symbolic links instead of duplicating data.

9.5.6 Disadvantages

1. Acyclic graph directories can quickly become complex, especially as the number of links between files and folders increases.
2. It can be more difficult to visualize the directory structure and understand how files and folders are related to each other, especially for users accustomed to traditional tree structures.s

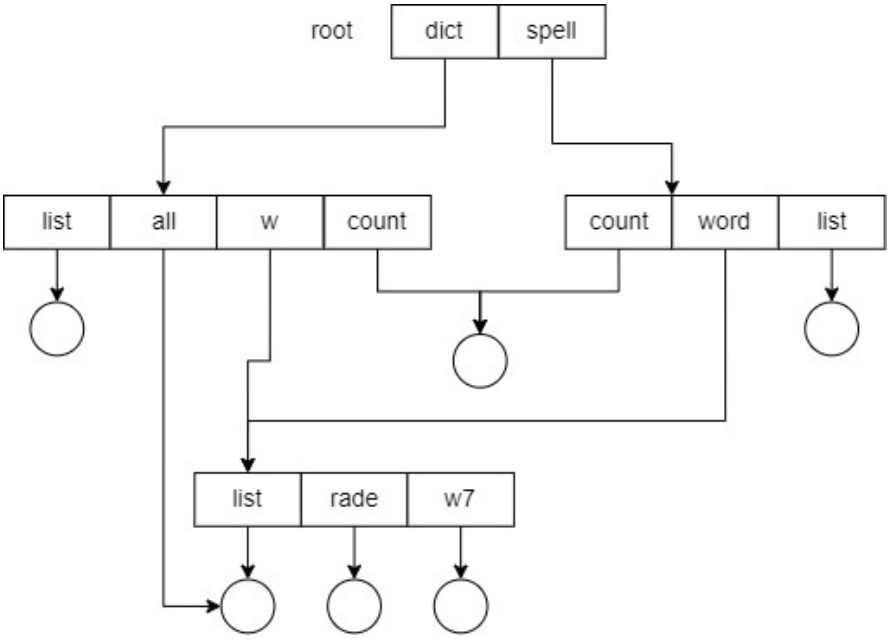


Figure 9.13: Acyclic Graph Directories

3. Performance Overhead.

Conclusion: In conclusion, the choice of file allocation technique in operating systems involves a careful consideration of various trade-offs. Contiguous allocation is advantageous for its simplicity and performance benefits but is hampered by fragmentation and inflexibility in file size management. Linked allocation, while offering dynamic file growth and better disk space utilization, suffers from performance penalties due to the overhead of managing pointers. Indexed allocation provides a balanced approach with efficient access and reduced fragmentation, though it introduces extra overhead for smaller files. Ultimately, the optimal allocation method depends on the specific requirements and constraints of the operating system, including performance goals, disk usage patterns, and file system complexity. Understanding these trade-offs is crucial for designing efficient and effective file systems.

9.6 Exercise

1. Discuss the significance of each common file attribute (e.g., file name, file size, file permissions) in ensuring data integrity and access control within a file system.

2. Compare and contrast contiguous allocation, linked allocation, and indexed allocation techniques in terms of their advantages, disadvantages, and suitability for different types of file systems.

3. Explain the concept of sequential access, direct/random access, and indexed access methods for accessing files, highlighting their characteristics and use cases.

4. Compare and contrast the efficiency of sequential access and direct/random access methods for reading and writing data, considering factors such as access time and storage utilization.

5. Describe the role of index blocks in indexed access methods and discuss how they improve file access performance and manageability.

9.7 Multiple Choice Questions

1. What does the "Seek" operation enable users to do?
   (a) Create new files and directories
   (b) Retrieve the content of a file
   (c) Change the content of a file
   (d) Shift the file pointer to a specific location within the file
2. What does the "Truncate" operation do in a file system?
   (a) Reduces the size of a file
   (b) Increases the size of a file
   (c) Deletes the file from the file system
   (d) Renames the file to a new name
3. Who is the file owner according to file attributes?
   (a) The user account or group that created the file
   (b) The administrator of the operating system
   (c) The owner of the storage device
   (d) The operating system itself
4. What is a drawback of contiguous allocation?
   (a) Reduced file access speed
   (b) Increased file fragmentation
   (c) Requires storage of pointers for each block
   (d) Allows random access to disk blocks
5. What advantage does indexed allocation offer over linked allocation?
   (a) Requires fewer pointers
   (b) Reduced file fragmentation
   (c) Improved access time using index block
   (d) Simple access with only starting address
6. What is a potential difficulty in file recovery associated with linked allocation?
   (a) Requires extra overhead
   (b) File fragmentation
   (c) Complications due to disk errors or corruption
   (d) Slow access times
7. Which file access method requires the use of an index table to locate the data within the file?

(a) Sequential Access

(b) Direct/Random Access

(c) Indexed Access

(d) Fragmented Access

8. What is a characteristic of indexed access?

(a) Requires reading data in a linear order

(b) Uses a pointer-based mechanism for data retrieval

(c) Access time depends on the position of data within the file

(d) Divides the file into fixed-size blocks

9. What advantage does an acyclic graph directory structure offer?

(a) It allows for faster directory traversal

(b) It provides more efficient storage utilization

(c) It allows for multiple parent directories for a directory

(d) It reduces the complexity of directory management

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 10

Input and Output Devices

Abstract: This chapter delves into the intricate aspects of input and output (I/O) management within operating systems, focusing on various methodologies and architectural considerations essential for optimizing system performance and reliability. Key topics include the architecture and functioning of Direct Memory Access (DMA) controllers, a comparative analysis of programmed I/O versus interrupt-driven I/O, and an exploration of memory-mapped I/O. The chapter further investigates spooling mechanisms, principles of I/O synchronization, and the pivotal role of device drivers in the operating system architecture. By examining these core elements, the chapter aims to provide a comprehensive understanding of the challenges and solutions associated with efficient I/O management in modern operating systems.

Keywords: Input/Output (I/O) Management, Direct Memory Access (DMA), Programmed I/O, Interrupt-Driven I/O, Memory-Mapped I/O, Spooling, I/O Synchronization, Device Drivers.

10.1 Input and Output(I/O) Management

Input devices and output devices are important part of computer systems. They allow users to interact with the computer, providing input data and receiving output results. Common input devices are keyboard, scanner, mouse etc.and common output devices are printer,monitor, etc.
I/O devices provide following important functions:

1. Facilitate user interaction: I/O devices, such as keyboards, mice, touchscreens, and microphones, enable users to interact with the computer and provide input data.
2. Provide output: I/O devices, such as monitors, printers, and speakers, allow users to receive output results from the computer.
3. Enhance productivity: I/O devices can increase productivity by making it easier and faster to perform tasks. For example, a scanner can quickly digitize a document, and a printer can produce hard copies of important documents.
4. Enable multimedia capabilities: I/O devices, such as cameras, webcams, and microphones, allow users to create and share multimedia content, such as photos, videos, and audio recordings.
5. Accessibility: I/O devices, such as speech recognition software and braille displays, make it possible for people with disabilities to interact with the computer.

10.1.1 I/O Related Tasks

An operating system (OS) controls how the computer's resources—its CPU, main storage, and input devices and output devices are used by one or more users. Operating system acts as a coordinator for input/output devices and the rest of the computer system. It provides a standardized interface for communicating with devices, manages I/O operations to optimize performance, handles interrupts and errors, and ensures that multiple devices can operate simultaneously. Operating system carried out following task in I/O management:

- Device drivers: The OS uses device drivers to communicate with I/O devices. These drivers provide a standardized interface for the OS to interact with the devices.
- I/O scheduling: The OS manages I/O operations by scheduling them to optimize performance. The OS may prioritize certain I/O requests over others to ensure that critical operations are completed quickly.
- Buffering: The OS uses buffers to manage I/O operations. A buffer is a temporary storage area that holds data while it is being transferred between the device and the system. Buffers help to prevent delays by allowing the device and the system to operate asynchronously.
- Interrupt handling: I/O devices often generate interrupts to signal that a transfer is complete or that an error has occurred. The OS handles these interrupts and takes appropriate action, such as notifying the application that initiated the I/O operation.
- Plug and Play: The OS provides support for Plug and Play devices, which are designed to be automatically recognized and configured by the system without requiring user intervention.
- Device management: The OS manages device resources, such as memory and bandwidth, to ensure that multiple devices can operate simultaneously without interfering with each other.

10.2 Types of I/O devices

I/O devices are roughly classified into two classes:

1. Block devices: In Block I/O devices, driver communicate in terms of fixed-sized blocks. They are used to store data that can be accessed randomly, meaning that the device can access any block of data as quickly as any other block stored. Hard drives, USB cameras, Disk-On-Key, etc. are a few examples of block storage devices.
2. Character devices: These devices are communicating in stream of characters. Example of character devices: – Mouse, keyboard, printers.

10.3 Device Drivers

An operating system (OS) interacts with input devices through device drivers, which are software components that allow the OS to communicate with the input devices. Device drivers provide a standardized interface for the OS to interact with input devices, enabling the OS to read data from the devices and respond to user input. Figure 10.1 shows how device drivers interact with device controller to access I/O devices. The interaction between an OS and input devices typically works as below:

1. Device detection: When an input device is connected to the computer, the OS detects the device and loads the appropriate device driver.

2. Device initialization: The device driver initializes the input device, configuring it for use with the computer.

3. Data transfer: When the user interacts with the input device, the device generates data that is sent to the device driver. The device driver then sends the data to the OS.

4. Data processing: The OS processes the data, interpreting it as user input. For example, if the user types a key on the keyboard, the OS interprets the key press and sends it to the appropriate application.

5. Interrupt handling: The input device may generate interrupts to signal that new data is available. The device driver handles these interrupts and ensures that the OS receives the data in a timely manner.

6. Device management: The OS manages input devices as system resources, allocating and releasing them as needed to ensure that multiple devices can be used simultaneously without interfering with each other.

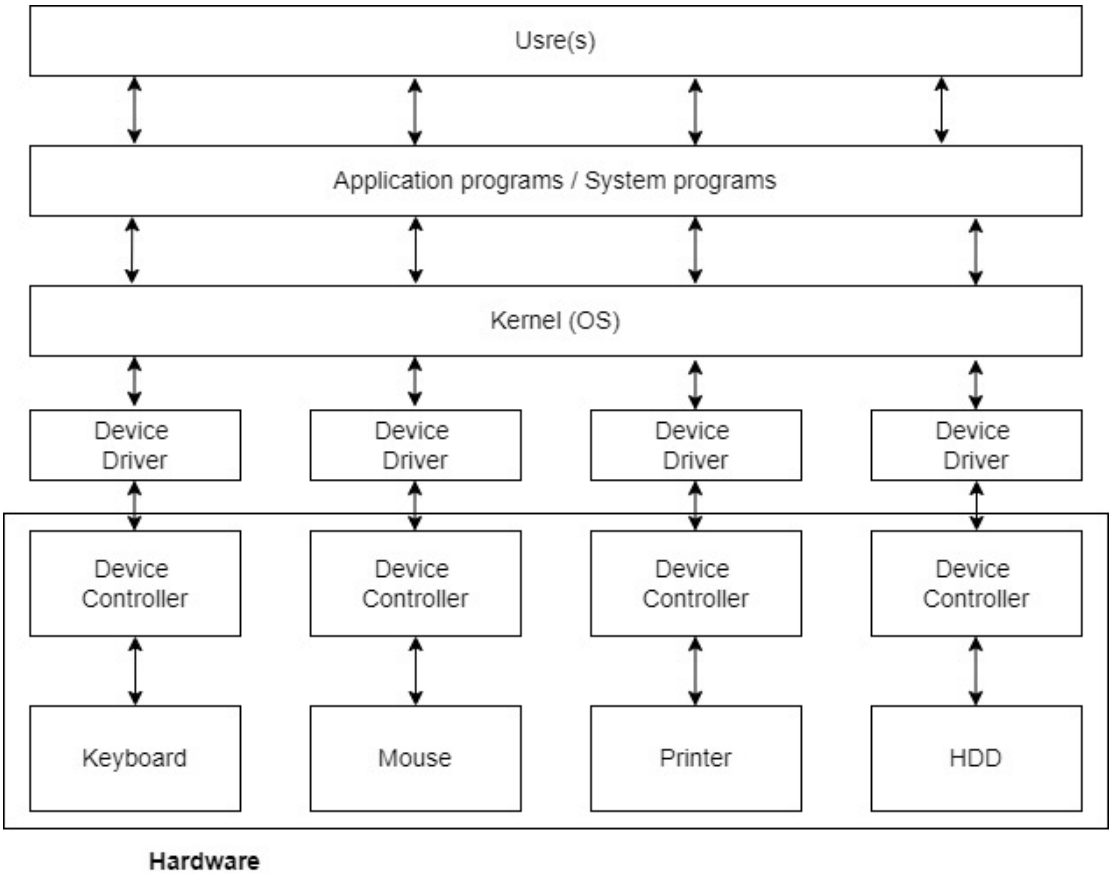


Figure 10.1: Device drivers and controllers

10.4 Device Controllers

A device controller manages the CPU's incoming and outgoing signals by serving as bridge between the CPU and the I/O devices. A plug and socket are used to attach a device to the computer, and a device controller is attached to the socket. Digital and binary codes are used by device controllers. I/O devices contain electrical and mechanical components (figure 10.2). The electrical component of an I/O device is a device controller. It may be able to handle multiple devices. It is also called adapter – Often takes the form of a printed circuit card.

Controller's tasks: It manages the device's operations. Data is sent to the device controller from a connected device. It converts a serial bit stream to a block of bytes and does any necessary error correction. It temporarily saves that information in the controller's local buffer (a special-purpose register).

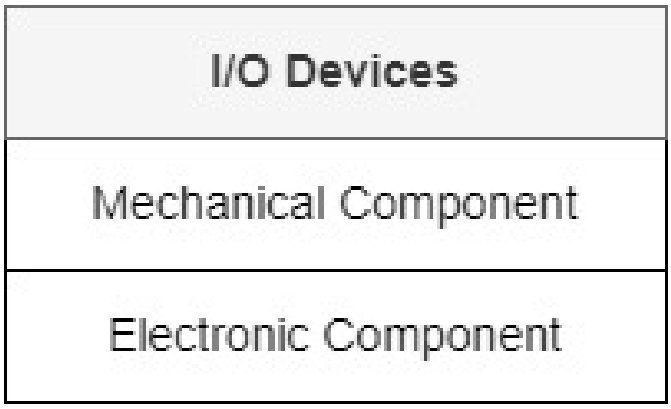


Figure 10.2: Device controllers

A corresponding device driver exists for each device controller. The memory controller is connected to the memory. The keyboard is connected to the keyboard controller, the monitor is connected to the video controller. Every I/O device is connected to the appropriate controllers. The common bus connects these controllers to the CPU. Device controller exchanges data with CPU via registers:

1. By writing into these registers
    - OS can command it to deliver or accept data
    - Or switch the device on or off
2. By reading from the registers
    - OS can learn the status of the device

Table 10.1 shows difference between device drivers and controllers.

Table 10.1: Difference between Device Drivers and Device Controllers

Features	Device Drivers	Device Controllers
Implemented in	Software	Hardware
Definition	It is a software that is mostly used in computers to run and control programs that communicate with different parts of a device.	It is an interface between the computer system and I/O devices. The device controller communicates with the system using the system bus.
Purpose	A device driver is a software that enables a computer to communicate and interact with a particular hardware device, such as a printer, sound card, graphics card, etc., so that the computer can understand its setup and specifications. The associated device won't operate properly without the proper device driver.	It is a hardware that is connected to the computer's I/O bus and offers a middle layer through which the OS can send orders like read, write, or more complicated commands.
Classification	User and kernel device drivers.	Serial port controller for a serial port, or complex like a SCSI controller

10.5 I/O Concepts

10.5.1 Uniform Driver Interface

The UDI (Uniform Driver Interface) is a project developed by a number of companies to provide a portable interface for the device drivers. The objective of UDI was designed to make the device drivers portable. The device driver works across the H/W and OS without any changes in the driver source. Several OS, platform, and device hardware vendors participated in the UDI project. UDI provided an encapsulated environment for the drivers with well-defined interfaces that isolated the drivers from OS policies, as well as from platform and I /O bus dependencies.

10.5.2 Spooling

Spooling stands for "Simultaneous Peripheral Operations On-Line" or sometimes "Spooler." It's a computer system technique used to improve the efficiency and smooth operation of input and output (I/O) devices, especially in situations where multiple processes or users are trying to use these devices simultaneously. Spooling works by temporarily holding data in a queue or buffer, allowing the computer to manage the I/O operations more effectively(figure 10.3). Here's how it works:

1. A buffer called SPOOL is created, which will have jobs and data till device is ready to make use and execute that job or operate on the data.

2. The data in the buffer (SPOOL) is loaded onto the main memory for the required operations when device is ready.

3. Spooling treats secondary memory as a huge buffer that can store any number of jobs and data. The jobs from SPOOL buffer are taken in the FIFO order.

4. The selected jobs are transferred into the main memory. After which they are executed one by one by the CPU.

5. After execution output generated will be transferred into the main memory and then it will be copied in secondary memory. Then output can be transferred into output.

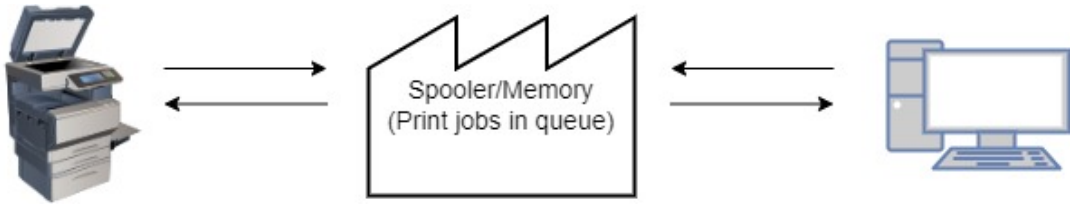


Figure 10.3: Spooling

Benefits of Spooling

1. Eliminate CPU Waiting: The primary benefit of Spooling is that it eliminates the need for direct interaction between I/O devices and the CPU. Consequently, the CPU does not have to pause and wait for I/O operations to complete, given that I/O devices typically have slower processing speeds compared to the CPU.

2. Improve CPU Utilization: With spooling in place, the CPU remains consistently active. It completes the execution of all tasks in the spool, ensuring that the CPU remains utilized.

3. Support for Concurrent I/O Operations: Spooling facilitates the concurrent operation of multiple I/O devices without any interference. It ensures that various I/O operations can be processed simultaneously without conflicts.

4. Maintaining CPU Speed: Spooling allows applications to execute at the full speed of the CPU while I/O devices may operate at their speeds (slow speed).

10.6 I/O Buffering

The I/O buffering is required in both case whether device is a block device or a character device. Let take an example, A user process wants to access data from slow device. The user process calls the read system call to the device. An approach to dealing with the incoming characters is for the user process to execute a read system call and to block for one character at a time. Each character that arrives causes an interrupt, and the interrupt-service procedure passes the character onto the user process before unblocking it. This approach is very slow because a user process has to be started up for every incoming character. A solution is to use a buffer in user space and which do not need involvement of user process in reading every character. A buffer is an area of the main memory that is used to temporarily store or hold data. To put it another way, a data buffer is a place where data is temporarily stored after being transmitted from one device or application to another. The process of temporarily storing data in the buffer is known as buffering. Buffer

can be implemented in the hardware as a fixed memory location or in software as a virtual data buffer pointing to a physical memory location. In both cases, the data stored in a data buffer is stored on the physical storage medium.

10.7 Types of Buffering

There are mainly three types of I/O buffering in the operating system as shown in figure10.4.

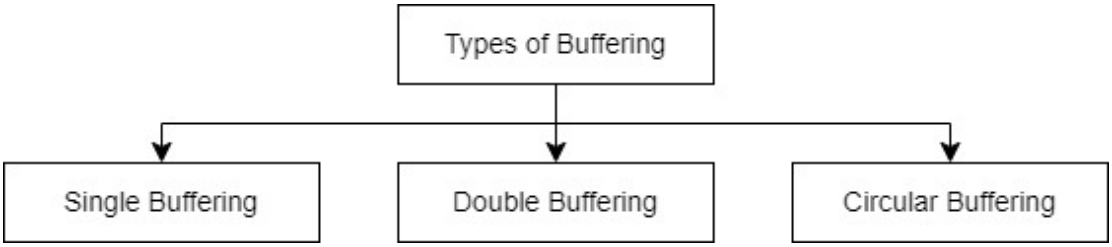


Figure 10.4: Types of Buffering

10.7.1 Single Buffer

In Single Buffering, only one buffer is used to transfer the data between two devices(figure 10.5). The producer process produces one block of data into the buffer. After that, the consumer process can consume the buffer. Only when the buffer is empty, the processor again produces the data.

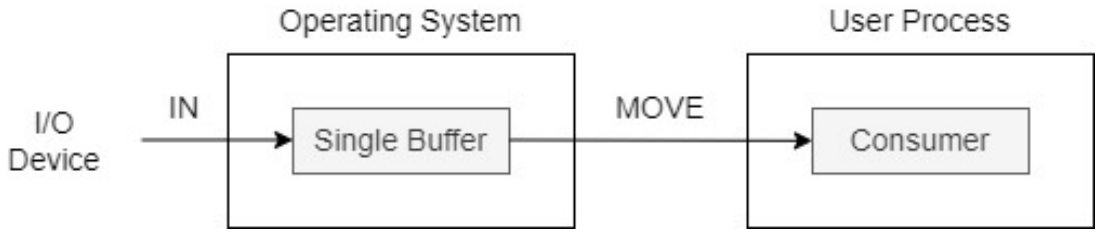


Figure 10.5: Single Buffer

10.7.2 Double Buffer

Double Buffering uses 2 buffers instead of 1. In double buffering, one buffer is used by the producer for keeping items while another buffer is consumed by the consumer at the same time(figure 10.6). Therefore, the producer does not have to wait for the buffer to fill. It is also called buffer swapping.

10.7.3 Circular Buffer

Circular buffer uses circular queue data structure. Circular buffer has a memory area and two pointers(figure 10.7). The first pointer indicates the next free word in the buffer where new data can be added. The second pointer indicates the first word in the buffer that has not yet been removed. The useful property of a circular buffer is that it does not need to

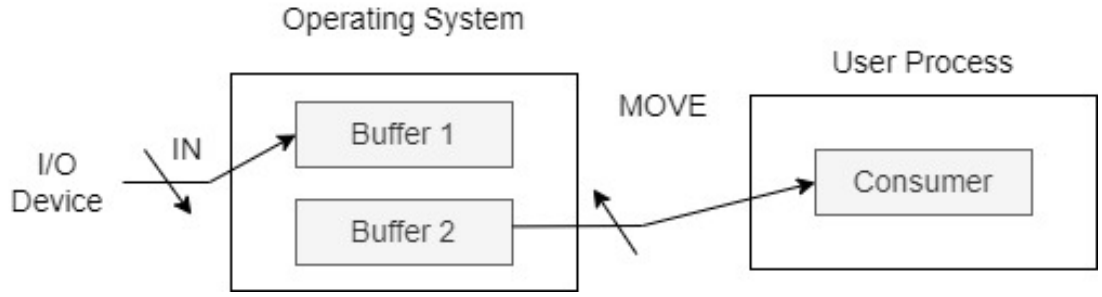


Figure 10.6: Double Buffer

have its elements shuffled around when one is consumed. (If a non-circular buffer were used then it would be necessary to shift all elements when one is consumed.)

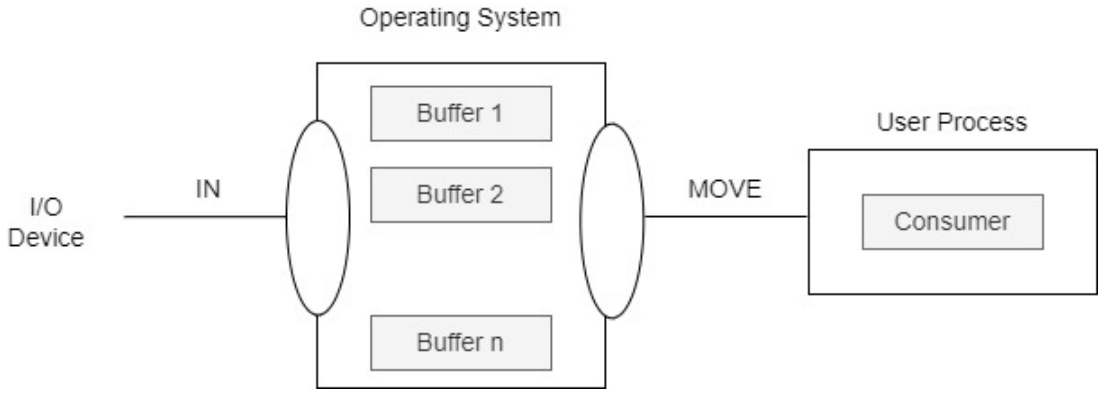


Figure 10.7: Circular Buffer

10.8 Blocking vs Nonblocking I/O

Blocking I/O and non-blocking I/O are two different approaches for handling input and output operations in computer systems. Blocking I/O makes the program wait until the I/O completes, while non-blocking I/O allows the program to continue executing other tasks and check back later for the results, making it suitable for scenarios where responsiveness and resource efficiency are important.

10.8.1 Blocking I/O

- Synchronous: Blocking I/O is a synchronous approach. When a program initiates an I/O operation (e.g., reading from a file or receiving data from a network socket), the program typically stops and waits for the operation to complete.
- Program Waiting: During a blocking I/O operation, the program is blocked or paused until the I/O operation finishes. This means that if the operation takes a long time (e.g., reading a large file or waiting for network data), the program can become unresponsive.
- Simplicity: Blocking I/O is straightforward to use and understand because it follows a linear and predictable flow. However, it can lead to inefficient resource utilization if

the program is frequently waiting for I/O.

- Resource Efficiency: It may not be the most resource-efficient approach because it can result in under utilization of CPU resources while waiting for I/O to complete.

10.8.2 Non-blocking I/O

- Asynchronous: Non-blocking I/O is an asynchronous approach. When a program initiates an I/O operation, it continues executing other tasks without waiting for the operation to finish.
- Program Continuity: In non-blocking I/O, the program does not get stuck waiting for the I/O operation to complete. Instead, it can perform other tasks or check back later to see if the I/O is finished.
- Complexity: Non-blocking I/O is generally more complex to implement compared to blocking I/O because it requires mechanisms like callbacks, polling, or event-driven programming to handle asynchronous I/O.
- Resource Efficiency: Non-blocking I/O can lead to better resource utilization, especially in situations where a program needs to handle many concurrent I/O operations or when responsiveness is critical.

10.9 I/O Techniques

There are number of ways to handle data transfers between the CPU and I/O devices. Some methods transport data to and from the memory unit directly, while others use the CPU as an intermediary step. There are mainly three ways of Input Output communication:

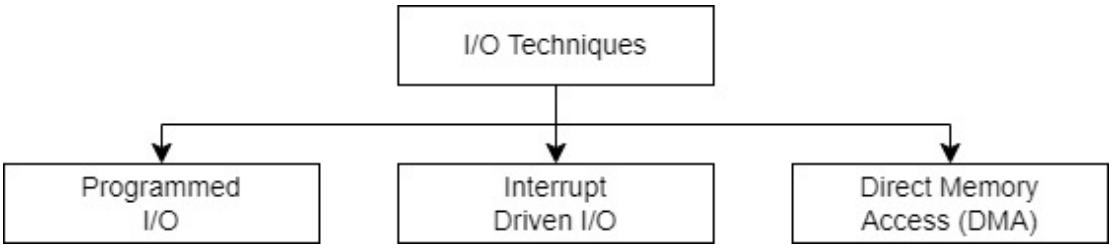


Figure 10.8: I/O Techniques

10.9.1 Programmed I/O

The I/O device doesn't have direct memory access in the programmed I/O. The CPU must run a program or a specific set of instructions for data to flow from an I/O device to memory. This approach use program for I/O communication that is the reason, it is called Programmed I/O.

In programmed I/O, I/O device does have direct access to the memory unit. The CPU initiate specific set of instructions in order to transport data from an I/O device to memory, including input instructions that move data from the device to the CPU and store instructions that move data from the CPU to memory. When I/O is programmed, the CPU remains in the programming loop until the I/O unit is ready to transfer data. This time in which CPU is waiting for I/O is overhead. This is the reason this method is slow and

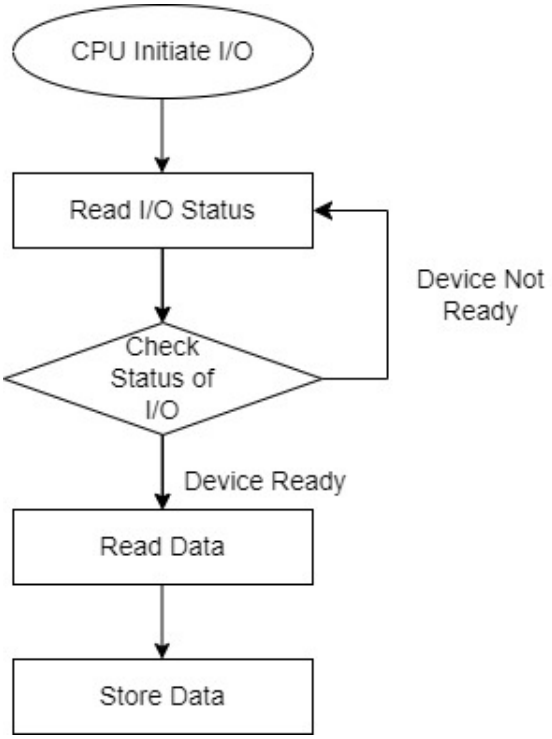


Figure 10.9: Programmed I/O

inefficient. The waiting time wasted in loop can be avoided by using an interrupt facility. In small, slow computers, the programmed I/O method is particularly helpful. The CPU issues the I/O device the instruction "Read," then waits in the program loop for the I/O device to respond. programmed I/O method is easy to understand and easy to program. For example, in communication with mouse CPU is waiting within a loop to get updated mouse pointer and click button states.

10.9.2 Interrupt Driven I/O

In this method, processor does not wait until the I/O operation is completed. The processor performs other tasks while the I/O operation is being performed. When the I/O operation is completed, the I/O module interrupts the processor to inform operation is completed. The interrupt driven I/O is faster than the programmed I/O module. Typical steps in Interrupt driven I/O are:

1. The I/O device produce and send an interrupt signal after completion of I/O operation by setting an interrupt enabled bit in the status register.

2. When the processor receives an interrupt signal, it stores the return address of the program counter into the memory stack. Then, the control branches to the interrupt service routine.

3. The interrupt service routine processes the I/O Transfer that is required.

4. Once the interrupt routine is completed, the CPU returns to the previous program and resumes the previous process.

The drawback of interrupt-driven I/O is that while reading or writing character interrupt will be generated. This method wastes some CPU time because handling of interrupts take time.

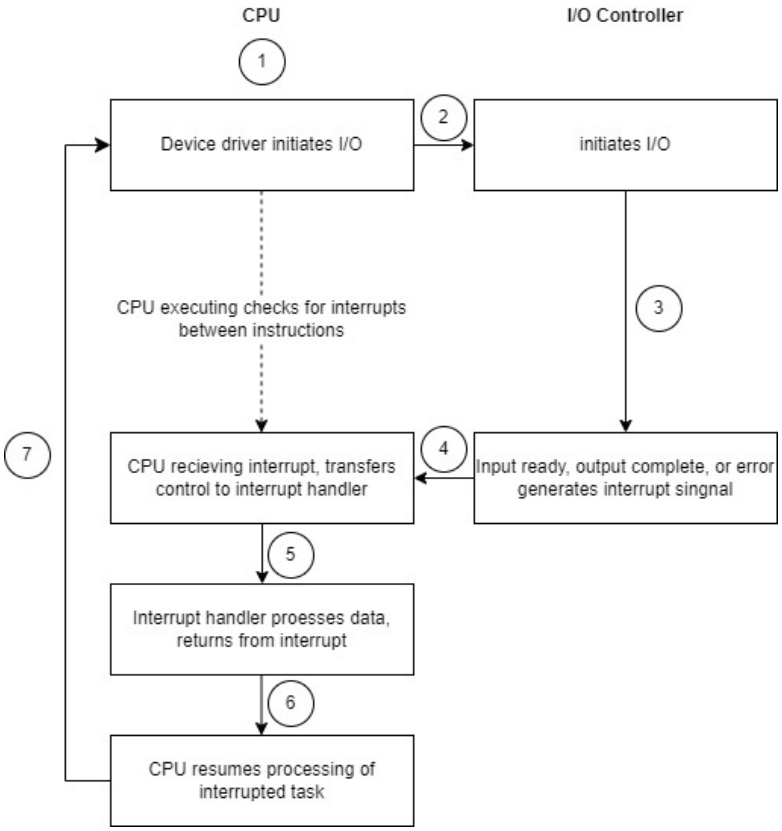


Figure 10.10: Interrupt Driven I/O

10.9.3 Direct Memory Access (DMA)

Programmed I/O use an expensive general-purpose processor to check status bits and to provide data into a controller register one byte at a time. The Direct memory access, or DMA based I/O system (Arun et al., 2020) can allow peripherals to communicate with one another directly over memory buses without the involvement of CPU. The CPU is not in use during DMA and has no influence over the memory buses. To manage the transfer between the memory unit and the I/O devices directly, the DMA controller takes control of the buses.

The typical sequence of DMA operation is as following:

1. DMA Request : Input-output device wants to transfer some data to the memory or from memory, it sends a DMA Request (DRQ) to the DMA controller.

2. Hold request (HLD): The DMA controller accepts this (DMA Request) DRQ and asks the CPU to capture or hold a few clock cycles by requesting a Hold request (HLD).

3. Hold acknowledgment (HLDA): The CPU receives a hold request (HLD) from the DMA controller, releases the bus, and sends the hold acknowledgment (HLDA) to the DMA controller.

4. DMA Acknowledgement (DACK): The DMA controller approves the input-output device (DACK) to perform data transfer after getting a hold acknowledgment (HLDA).

5. DMA Transfer: The DMA controller takes power from the system bus and transfers data to or from memory.

6. Notify completion of Transfer: Once the data transfer completes, the DMA raises an interrupt to notify the processor about completing the data transfer function.

DMA can transfer data in three modes:

1. Burst Mode: In this mode DMA controller once gains the charge of the system bus, then it releases the system bus only after completion of data transfer. CPU has to wait for the system buses till DMA controller completes transfer.
   Advantages:

   - Fastest mode: This mode is fastest mode of DMA Transfer.

   Disadvantages:

   - Less user friendly: In this mode during the DMA transfer, CPU will be blocked.

2. Cycle Stealing Mode: In this mode, the DMA controller forces the CPU to stop its operation. The DMA gets the control over the bus for a short term from CPU. The DMA controller releases the bus after each byte has been transferred before making another request for the system bus. The DMA controller steals the clock cycle needed to transmit every byte.
   Advantages:

   - CPU will not be blocked at all.

   Disadvantages:

   - Low data transfer rate: The rate of DMA data transfer will be less than burst mode of DMA.

3. Transparent Mode: If the CPU does not require system bus, then DMA controller takes the charge of system bus. CPU will not be blocked due to DMA. This is the slowest mode of DMA Transfer since DMA controller has to wait to get the access of system buses from the CPU. That is the reason in this mode less amount of data will be transferred. Advantages:

   - CPU won't be blocked entire time.

   Disadvantages:

   - Slowest DMA transfer rate.

10.9.4 Benefits of DMA

1. Faster read write: It transfers data without the involvement of the processor, which makes read write task faster.

2. DMA reduces the clock cycle requires to read or write a block of data.

3. Implementing DMA also reduces the overhead of the processor.

10.9.5 Disadvantages of DMA

1. Cost: DMA is a hardware unit; it will require additional cost to implement a DMA controller in the system.

2. Cache coherence problem: DMA transfer could change the contents of main memory that has been previously cached by the processor.

10.10 Design Issues for Input Output Management

Operating systems play a crucial role in managing the input and output operations of a computer system. The design and implementation of an operating system's input and output management subsystem must address several key issues to ensure efficient and effective utilization of hardware resources. Some of these design issues include:

1. Device Independence: One important design issue for input and output management in operating systems is device independence.This means that the operating system should provide a uniform interface for different types of devices, allowing applications to perform input and output operations without having to worry about the specific details of each device. This design issue is crucial for ensuring compatibility across different hardware configurations and allowing applications to be easily ported and run on various systems.

2. Scheduling and Control: Another important design issue for input and output management is scheduling and control.The operating system needs to handle multiple input and output requests concurrently, efficiently allocating resources and ensuring fairness in accessing the devices.To address this issue, operating systems use various scheduling algorithms, such as round-robin or priority-based scheduling, to determine the order in which input and output operations are performed. This helps to optimize resource utilization and ensure that critical tasks are given priority.

3. Buffering: Buffering is another design issue for input and output management. It involves the use of memory buffers to temporarily store incoming or outgoing data, allowing for efficient data transfer between devices and applications. Buffering helps to improve performance by reducing the overhead of frequent data transfers and providing a smoother flow of data.

4. Error Handling and Recovery: The design of an operating system's input and output management subsystem must also consider error handling and recovery mechanisms.These mechanisms are necessary to handle potential errors during input and output operations, such as device failures or data corruption.The operating system should have robust error handling mechanisms in place to detect and recover from these errors, ensuring the reliability and integrity of data.

5. Access Control: The operating system must have mechanisms in place to control access to input and output devices.This includes determining which applications or users have permission to access specific devices, preventing unauthorized access and ensuring data security.In addition, the operating system should provide mechanisms for managing and enforcing access control policies, such as user authentication and authorization.

Conclusion: Effective input and output management is critical to the performance and reliability of operating systems. This chapter has highlighted the importance of various I/O techniques and architectures, including DMA controllers, programmed I/O, interrupt-driven I/O, and memory-mapped I/O. It also emphasized the role of spooling in optimizing I/O operations and the significance of I/O synchronization in preventing data corruption and ensuring smooth data flow. The discussion on device drivers elucidated their crucial role in providing a standardized interface for hardware-software communication. In summary, understanding and implementing efficient I/O management strategies is essential for enhancing system performance, ensuring data integrity, and achieving seamless device integration in operating systems.

10.11 Exercise

1. What are I/O devices in the context of an operating system, and why are they important?
2. Explain the concept of device controllers and their role in I/O management.
3. What is DMA?
4. What is the Life Cycle of an I/O request?
5. What do you understand by Synchronous and Asynchronous I/O System?
6. What are the Blocking and Non-Blocking input output?
7. What are the characteristics of I/O Devices?
8. Write about I/O buffering.
9. What are the different modes of Interrupt? How is polling achieved?
10. Explain the difference between an internal interrupt and software interrupt.
11. How is an interrupt enabled and detected?
12. Explain the concepts of spooling.
13. Explain the architecture of DMA controller in detail.
14. Discuss the advantages and disadvantages of programmed I/O versus interrupt-driven I/O.
15. What is memory-mapped I/O, and how does it differ from traditional I/O methods?
16. Operating System Design Issues in I/O Management:
17. Explain the principles of I/O synchronization and its importance in operating system design.
18. What is the purpose of device drivers, and how do they fit into the OS architecture?

10.12 Multiple Choice Questions

1. What is the primary function of input devices in a computer system?

   (A) To provide output results to users
   (B) To facilitate user interaction and provide input data
   (C) To enhance productivity by performing tasks faster
   (D) To enable multimedia capabilities such as photo editing

2. Which of the following is NOT an example of an input device?

   (A) Keyboard
   (B) Scanner
   (C) Printer
   (D) Mouse

3. What role does the operating system play in I/O management?

   (A) It controls how computer resources are used by users
   (B) It enhances the multimedia capabilities of input devices
   (C) It provides a standardized interface for communicating with I/O devices
   (D) It ensures that multiple devices can operate simultaneously

4. What are device drivers used for in I/O management?

   (A) To enhance productivity by optimizing performance
   (B) To provide a standardized interface for the OS to interact with I/O devices
   (C) To handle interrupts and errors in I/O operations
   (D) To manage the communication between input and output devices

5. What is the role of device drivers in interacting with input devices?

   (a) They manage the CPU's incoming and outgoing signals
   (b) They initialize the input devices and configure them for use
   (c) They interpret user input and send it to the appropriate application
   (d) They provide a standardized interface for the OS to communicate with input devices

6. When does the device driver handle interrupts in the interaction between the OS and input devices?

   (a) During device detection
   (b) During data transfer
   (c) During device initialization
   (d) When the input device signals that new data is available

7. What does the device controller do in relation to input devices?

   (a) It manages the device's operations and processes user input

(b) It provides a standardized interface for the OS to interact with input devices

(c) It converts a serial bit stream to a block of bytes and handles error correction

(d) It interprets user input and sends it to the OS for processing

8. How does Spooling improve the efficiency of input and output (I/O) devices?

   (a) By synchronizing I/O operations across multiple devices

   (b) By storing data temporarily in a buffer, allowing effective management of I/O operations

   (c) By directly executing jobs from secondary memory without involving the main memory

   (d) By increasing the processing speed of I/O devices through parallel operations

9. In Spooling, what happens to the data stored in the buffer when the device is ready for operation?

   (a) It is immediately executed by the CPU

   (b) It is transferred to the main memory for execution

   (c) It is stored permanently in secondary memory

   (d) It is deleted from the buffer to make space for new data

10. Which term refers to the approach where a program pauses and waits for an I/O operation to complete?

    (a) Asynchronous I/O

    (b) Spooling

    (c) Blocking I/O

    (d) Non-blocking I/O

11. What is the primary advantage of non-blocking I/O over blocking I/O?

    (a) Simplicity of implementation

    (b) Synchronous execution of tasks

    (c) Improved resource utilization

    (d) Predictable flow of execution

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 11

Security

Abstract: This chapter delves into the critical aspects of operating system security, emphasizing the importance of safeguarding computer systems and their resources from unauthorized access and damage. It explores the primary goals of operating system security, namely confidentiality, integrity, and availability, and discusses various security threats categorized into program threats and system and network threats. Additionally, the chapter covers security mechanisms including authentication, authorization, encryption, firewalls, and intrusion detection systems. The content further examines the concept of protection domains and access control lists, essential for defining and enforcing access policies.

Keywords: Operating system security, confidentiality, integrity, availability, program threats, system threats, authentication, authorization, encryption, firewalls, intrusion detection, protection domain, access control list.

Security means protecting your computer system and resources such as CPU, memory, disk, software programs and most importantly data/information in it from unauthorized access or damage. If a computer program is run by an unauthorized user then he/she may cause severe damage to the computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc. The primary goals (figure 11.1) of operating system security are confidentiality, integrity, and availability(Jaeger, 2008):

1. Confidentiality: It ensures that sensitive information and data are kept private, so that only authorized users or processes can access them. To achieve this, various measures such as access controls, encryption, and user authentication are implemented to maintain confidentiality and prevent unauthorized access.

2. Integrity: Data integrity means making sure that information stays accurate and unchanged from start to finish. It's important to prevent any unauthorized changes, deletions, or damage to the data. To maintain data integrity, techniques like checksums, digital signatures, and file permissions are used to ensure that the information remains reliable and free from any tampering or corruption.

3. Availability: This goal is about ensuring that the system and its resources are available and functional when needed. Protection against denial-of-service (DoS) attacks and hardware failures falls under this category.

In brief, operating system should prevent malicious access of the system by users or programs. It should ensure each resource shared is only used according to system rules.

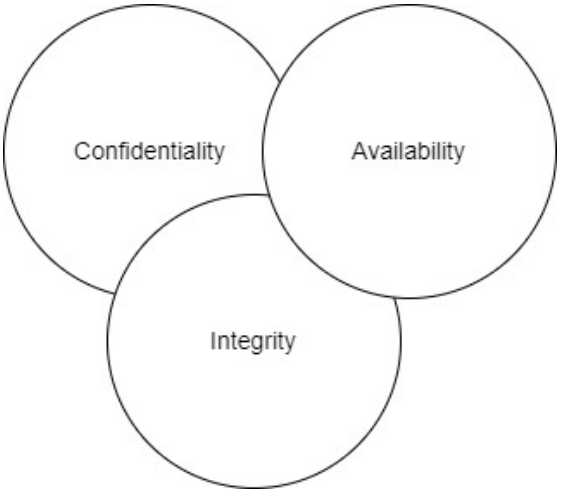


Figure 11.1: Security goals

These rules can be set up by system developers or system managers. Any unwanted programs will not damage the entire system. In case if damage cannot be avoided it must be minimal.

11.1 Security Threats

Threats can be classified in Program Threats and System & Network Threats.

11.1.1 Program Threats

A program threat is a user program made to do malicious tasks. For example, a user program is transmitting user credentials via a network to some hacker. Few common program threats are following:

1. Virus: A virus is generally a small code embedded in an application. Virus can delete or modify user data. They can replicate themselves on computer systems and are also designed to infect other computers. They can also be spread through emails, files and infected hardware (figure 11.2).

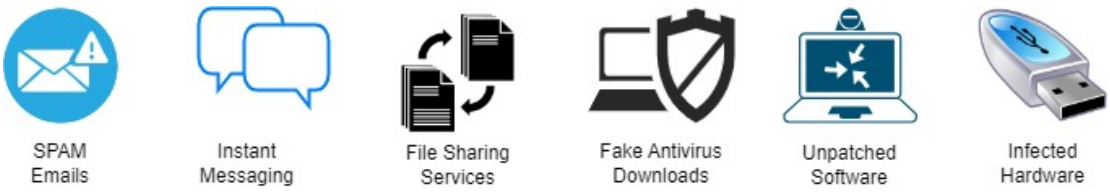


Figure 11.2: Common ways by which Virus spreads

2. Trojan horse - a Trojan horse is a threat, in which a standard program hides malicious code inside. It captures user login information and stores it so it can be sent to a malicious user later so they can access system resources and log into the machine. Trojan horse is not self replicating like other computer viruses.

3. Trap Door – Trap door are security hole in a program by which someone can access any resource without following normal security procedure. They have been used by programmers to test and debug code for many years.

4. Logic Bomb - A logic bomb is a program that can attack an operating system, a program, or a network by carrying malicious code in its set of instructions. It is activated in response to a certain event, such as a specified date or time, the deletion of a specific record from a system, or the start of the infected software program.

11.1.2 System & Network Threats

System threats can use network connections and system services improperly to harm users. They can also activate viruses on a complete network. A system threat creates an environment in which operating system resources for example user files are accessed without permission. Few common system threats are:

1. Denial of Service (DoS) - Denial of service attacks keeps genuine users from getting to system resources. In it attackers normally flood requests to user processes, web servers or operating system services. The overloaded process will not be able to complete its usual work normally. For example, users may not be able to use the internet if denial of service attacks a web browser process. More advanced form of DoS is Distributed Denial of Service (DDoS) attack (McHoes and Ballew, 2012), in which an attacker sends request from multiple sites at the same time (figure 11.3).

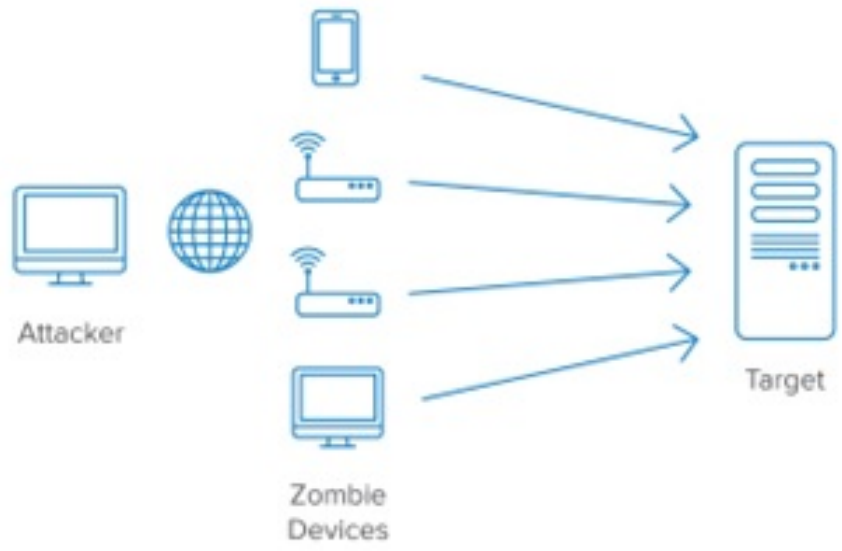


Figure 11.3: DDoS Attack

2. Worm - Worm is a process that can drastically reduce a system's performance by demanding a lot of system resources. A worm process creates numerous clones of itself, each of which consumes system resources and prevents all other processes from obtaining the resources they need. Even a entire network can be halted by worms (figure 11.4).

3. Port Scanning - Port scanning is a mechanism by which a hacker can detects system vulnerabilities finding network services running on a host. They can also be used by security analysts to confirm network security policies (figure 11.5).

4. Buffer Overrun: It's also known as buffer overflow. It is a common security problem with the operating system. Basically, it's when more data is put into a buffer or data

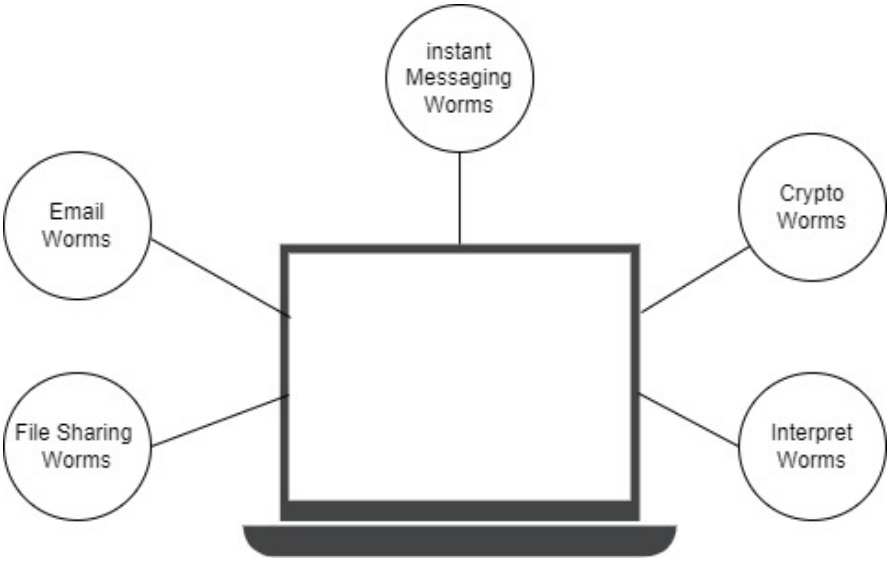


Figure 11.4: Types of worms

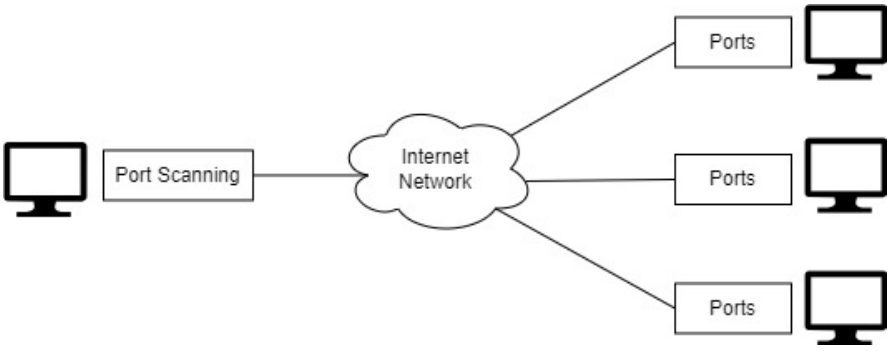


Figure 11.5: Port Scanning

holding area than it has room for, and it can overwrite other data. Hackers use this to crash the system or use specially designed malware to get into the system (figure 11.6).

11.2 Security Mechanisms

11.2.1 Authentication

It is process of verifying the identity of users or processes trying to access the system. It is done by using passwords, biometrics, or smart cards. The purpose of authentication(figure 11.7) is to prevent unauthorized access to sensitive data and resources in the operating system. Operating Systems generally identifies/authenticates users using following three ways:

1. Username / Password - User need to enter a registered username and password with Operating system to login into the system.

2. User card/key - User need to punch card in card slot, or enter key generated by key generator in option provided by operating system to login into the system.

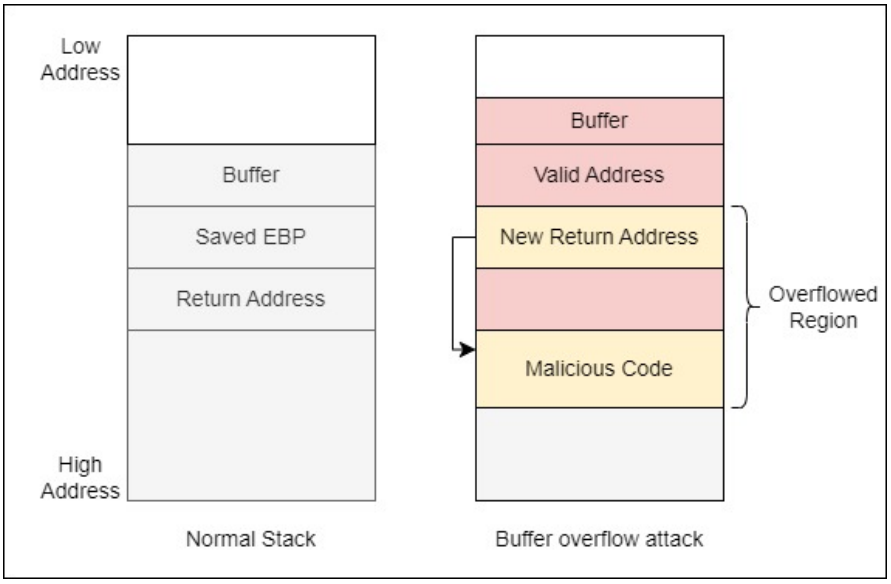


Figure 11.6: Buffer Overflow attack

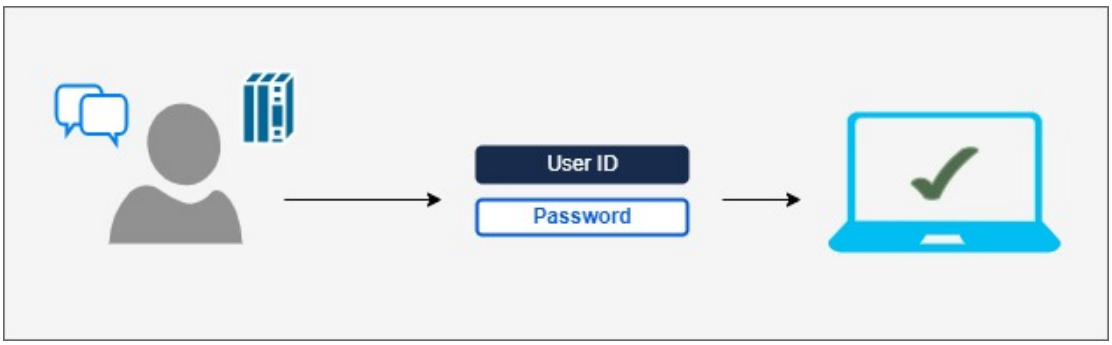


Figure 11.7: Authentication

3. User attribute - fingerprint/ eye retina pattern/ signature - User need to pass his/her attribute via designated input device used by operating system to login into the system.

11.2.2 Authorization

Authorization in an operating system is like a permission slip. It decides what a user is allowed to do and access. Just like how you might need permission to enter certain places, the operating system decides what actions a user can perform and what resources they can access within the operating system. For example read authorization gives read permission to certain user.

11.2.3 Encryption

Encryption (Berghel et al., 2008) is the process of converting data into a secret code that can only be understood by authorized parties. Only the people with the special key (decryption key) can understand secret code(message) . It keeps your information safe whether it's sitting somewhere or being sent over the internet (figure 11.8).

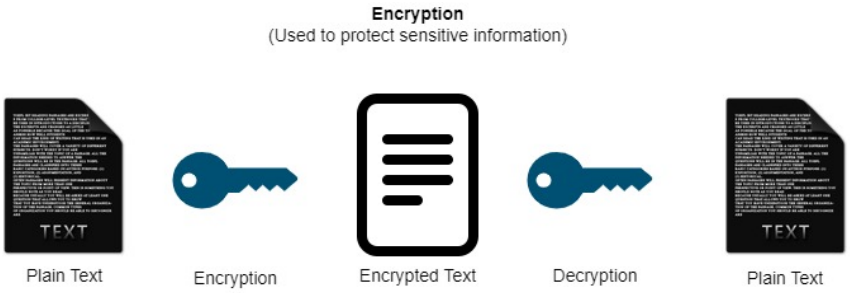


Figure 11.8: Encryption

11.2.4 Firewalls

Firewalls are used to filter network traffic, allowing only authorized communication and blocking potentially malicious traffic (Andress, 2014). They help prevent unauthorized access and attacks from external networks (figure 11.9).

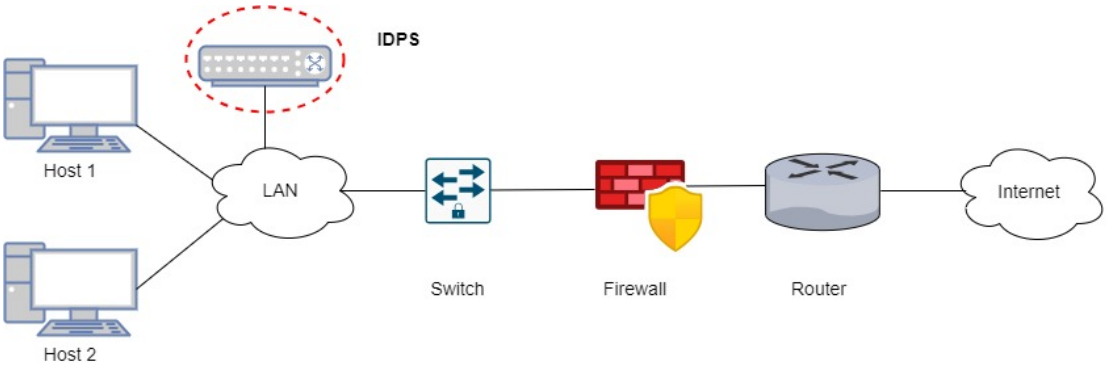


Figure 11.9: Firewalls

11.2.5 Intrusion Detection and Prevention Systems (IDPS)

An intrusion detection system monitors and analyzes network traffic. It looks for any suspicious activity, like an exploit attempt or an incident that could be a threat to your network. The prevention module prevents any suspicious activity or attack. An IDPS prevents attacks by dropping malicious packets, blocking suspicious IP addresses and alerting security in case of any threats. It generally uses a database for signature recognition and can be programmed to recognize attacks based on traffic and behavioural anomalies. While firewall performs actions such as blocking and filtering of traffic, the IDPS detects and alert a system administrator or prevent the attack as per configuration (figure 11.7).

11.2.6 Secure Boot

Secure Boot is a way to make sure your computer boots up using only software that's trusted by the manufacturer. It works on Windows, many Linux distributions, and some BSD variants. When the PC boots up, the firmware looks at the signatures of all the boot software, including the firmware drivers and operating system. If they're all okay, the PC boots up and the firmware gives the operating system control.

11.2.7 Patch Management

Operating systems and software should be updated with security patches on a regular basis to address known vulnerabilities and protect against threats.

11.2.8 Audit Trails and Logging

An audit log is a security record that stores who has accessed a computer system and what actions were taken over a specific time. These logs are necessary for detecting security problems. Audit trials are used to perform in-depth analysis of how data has changed on the system.

11.2.9 Sandboxes and Virtualization

Sandboxes separate processes from each another and also from system. It will reduce possibility of security compromise. Virtualization enables the running of multiple OS on a single physical machine.

11.3 Protection Domain

A protection domain is a set of resources that a process can access. In it all resources both hardware & software are modeled as objects. It ensures that the process should only have access to those objects it needs to complete its task. The access mode for object cannot be changed and object can be accessed for particular time slot. A process or object is an abstract data type within a computer system. Each object has its own set of operations.

In a domain component, an object can be defined as: Object: (ID, a set of operations on an object).

The process-domain relationship can be a static relationship or a dynamic relationship. Dynamic relationship offers domain switching, in which objects can change domains. Domains can be implemented as users, processes, or procedures. For example, if every user is a domain, then the domain defines the user's access, and changing the domain means changing the user ID.

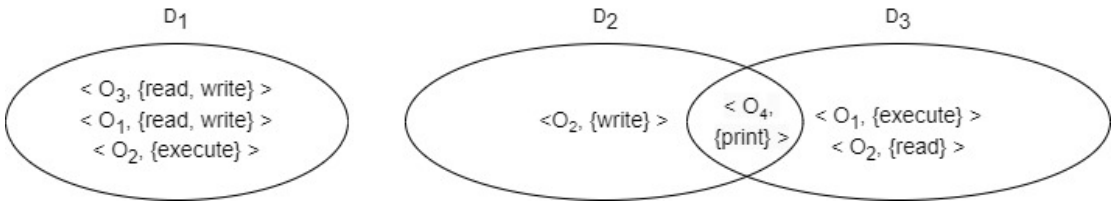


Figure 11.10: Protection Domain

11.4 Access Control List

An access control list (ACL), with respect to a computer file system, is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects(figure 11.11). Each entry

in a typical ACL specifies a subject and an operation. For instance, if a file object has an ACL that contains (Alice: read,write; Bob: read), this would give Alice permission to read and write the file and Bob to only read it. ACL provides better file security by enabling you to define file permissions on "per-user/per group" basis. Simplifying they can be viewed as several user-defined layers of classic Unix permission overlaid on each other. ACLs provide inheritance of attributes from directories to files via so called default permissions (which exists only for directories) and generalize umask concept by applying it to read and write operations, in addition to file creation like in classic Unix.

Access Control: This involves defining and enforcing rules that determine who can access specific resources (such as files, directories, and devices) and what actions they are allowed to perform. Commonly used methods are Access control lists (ACLs), permissions, and role-based access control (RBAC) .

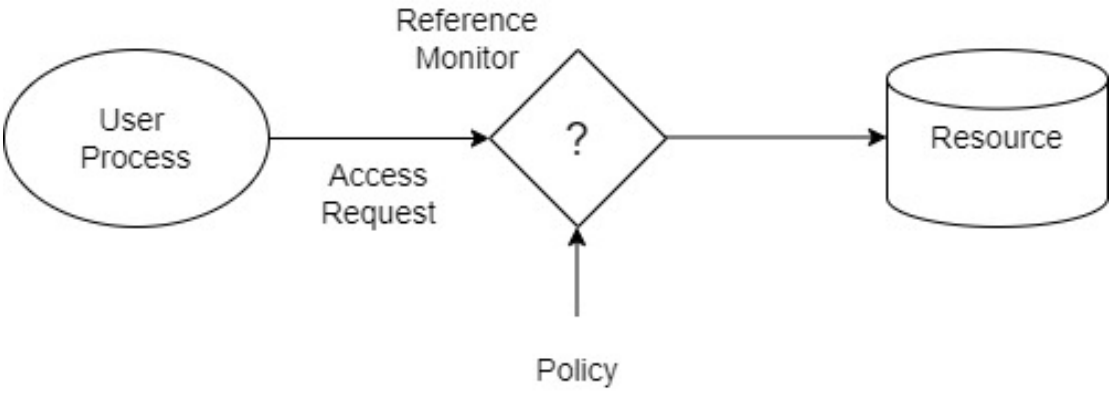


Figure 11.11: Access Control List

Conclusion: In conclusion, the chapter provides a comprehensive overview of operating system security, highlighting its fundamental goals and the various threats that can compromise system integrity. By implementing robust security mechanisms such as authentication, authorization, and encryption, along with effective use of firewalls and intrusion detection systems, operating systems can mitigate risks and protect critical resources. Understanding and applying these security principles are essential for maintaining the reliability and trustworthiness of modern computing environments.

11.5 Exercise

1. How Access Control List can be useful for managing file access?
2. Discuss various security threats in File System of OS.
3. Write a short note on authentication?
4. Write a note on Generic Security Attacks.
5. Access metrics mechanism.

6. Write short notes on following: (i) Design Principles of Security (ii) Firewall.
7. Explain Protection Mechanism illustrating use of Protection Domain.

11.6 Multiple Choice Questions

1. What is the primary goal of operating system security?
   (a) Efficiency
   (b) Confidentiality, integrity, and availability
   (c) System performance
   (d) Compatibility
2. Which security goal ensures that sensitive information is kept private and only accessible to authorized users?
   (a) Integrity
   (b) Availability
   (c) Confidentiality
   (d) Authentication
3. What is the purpose of data integrity in operating system security?
   (a) To ensure that system resources are available and functional when needed
   (b) To prevent unauthorized access to sensitive information
   (c) To ensure that information remains accurate and unchanged
   (d) To authenticate users trying to access the system
4. Which of the following is an example of a program threat?
   (a) Denial of Service (DoS) attack
   (b) Port scanning
   (c) Buffer overrun
   (d) Digital signature
5. What is the role of authentication in operating system security?
   (a) Filtering network traffic
   (b) Preventing unauthorized access to system resources
   (c) Encrypting data for secure transmission
   (d) Monitoring network traffic for suspicious activity
6. What is a protection domain?
   (a) A set of users with similar access permissions
   (b) A set of resources that a process can access
   (c) A type of dynamic relationship between objects
   (d) A list of permissions attached to a computer file system

7. How are objects represented in a protection domain?

   (a) As users and processes
   (b) As abstract data types with their own set of operations
   (c) As hardware and software components
   (d) As dynamic relationships between domains

8. What is the purpose of domain switching in a protection domain?

   (a) To change the user ID of a process
   (b) To change the access mode for an object
   (c) To enable objects to change domains
   (d) To define user access permissions

9. What does an access control list (ACL) specify?

   (a) Which users or system processes are granted access to objects
   (b) The set of operations allowed on given objects
   (c) Both A and B
   (d) None of the above

10. How does an ACL enhance file security?

    (a) By defining file permissions on a per-user/per-group basis
    (b) By providing inheritance of attributes from directories to files
    (c) By generalizing the umask concept for read and write operations
    (d) All of the above

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

DeepScience
Open Access Books

Chapter 12

Virtualization

Abstract: This chapter delves into the historical evolution and current advancements of virtualization technologies. It explores the fundamental concepts of hypervisors, differentiating between Type 1 and Type 2 hypervisors, and examines the role of paravirtualization and hardware-assisted virtualization. The chapter also covers containerization with Docker, highlighting its impact on application deployment and development. Furthermore, it discusses the integration and optimization of VMware ESXi in virtualized environments, as well as security measures and troubleshooting techniques. The chapter concludes by addressing the latest trends in cloud computing, serverless computing, and edge computing, emphasizing how virtualization continues to drive innovation in IT infrastructure.

Keywords: Virtualization, Hypervisors, Type 1 Hypervisor, Type 2 Hypervisor, Paravirtualization, Hardware-Assisted Virtualization, Containerization, Docker, VMware ESXi, Cloud Computing, Serverless Computing, Edge Computing.

Virtualization (Mishra and Kulkarni, 2018) is a technology that allows you to create multiple simulated environments or dedicated resources from a single, physical hardware system. This enables you to run multiple operating systems and applications on a single machine, providing various benefits such as increased efficiency, flexibility, and cost savings.

The concept of virtualization can be interpreted in a variety of ways, and its components are available in every aspect of computing. This current development can be seen by virtual machines. It provides significant advantages in terms of resource utilization, cost savings, and flexibility by enabling many operating systems or applications to run simultaneously on a single physical server or computer. For instance, a server can be divided into several virtual machines (VMs), each of which can run its own operating system and set of applications. The system with and without virtual machines presented in figure 12.1.

Organizations may benefit from this as it eliminates the need for numerous physical servers, reducing expenses on hardware, power use, and maintenance. Along with greater scalability and flexibility, virtualization also makes application deployment and management smoother. Different environments can be isolated from one another, preventing problems in one VM from affecting the others. In general, operating system virtualization offers a more seamless and effective method of computing.

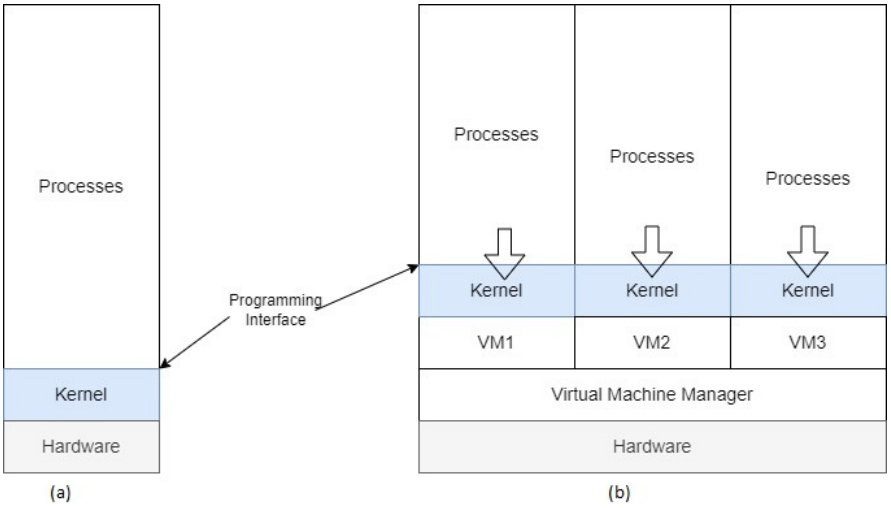


Figure 12.1: System models (a)Nonvirtual Machine (b)Virtual Machine

12.1 Historical Background and Development of Virtualization Technologies

Virtualization has a rich history that can be traced back to the early days of mainframe computing. It was during this time that IBM's CP-40 and CP-67 systems first introduced the concept of virtual machines (VMs). These VMs allowed for the partitioning of mainframe hardware, enabling multiple users and applications to run simultaneously. This concept, also known as early mainframes, revolutionized the way that hardware was utilized. By maximizing the use of resources, organizations were able to improve efficiency and productivity.

The development of the first hypervisors was a game-changer in the world of technology. IBM mainframes now allow simultaneous running of multiple virtual machines, a significant breakthrough for developers and IT professionals. The CP/CMS, also known as the Control Program/Cambridge Monitor System, was one of those groundbreaking hypervisors that paved the way for modern virtualization. With this incredible technology, it became possible to maximize the utilization of hardware resources and achieve unprecedented levels of efficiency. The limitations of physical servers were lifted, allowing for the creation and management of multiple virtual machines with ease. The CP/CMS opened up a whole new world of opportunities for businesses and organizations, allowing them to streamline their operations and reduce costs. It was truly a revolution in the field of computing. Today, virtualization has become an integral part of our lives, powering cloud computing, virtual desktop infrastructure, and much more. But do not forget where it all began - with the development of hypervisors like CP/CMS. Virtualization has changed the way we think about computing, and it all started with that first hypervisor.

During the 1980s, virtualization technologies started to gain traction and become more accessible to a wider range of organizations. Companies like IBM and DEC played a crucial role in commercializing these technologies and making them available to the market. IBM's VM/370 and DEC's VAX VMM are two notable examples of virtualization platforms that emerged during this time. IBM's VM/370 was a virtual machine operating system that allowed multiple instances of operating systems to run concurrently on a single mainframe computer. This technology enabled organizations to consolidate their computing resources and optimize their hardware utilization. By partitioning the mainframe into multiple virtual machines, IBM's VM/370 offered improved flexibility, scalability, and cost-efficiency for

businesses. Similarly, DEC's VAX VMM, or Virtual Memory Manager, provided virtualization capabilities for the popular VAX architecture. This technology allowed multiple virtual machines to run on a single VAX system, enabling organizations to maximize their computing resources and improve overall system performance. With the VAX VMM, businesses could consolidate their workloads, reduce hardware costs, and simplify system management. The commercialization of virtualization technologies by companies like IBM and DEC in the 1980s marked a significant milestone in the evolution of computing.

In the early 2000s, x86 virtualization started to gain popularity with the introduction of hardware-assisted virtualization features. This marked a significant advancement in virtualization technology on the x86 architecture. Intel VT-x and AMD-V were two such technologies that revolutionized the field. These technologies greatly enhanced performance and security, making it possible to develop more reliable and robust virtualization solutions. With hardware support for virtualization, the overhead and limitations of software-based virtualization were significantly reduced, leading to improved efficiency and better utilization of resources.

During the 2000s to the 2010s, the field of virtualization witnessed a significant increase in the number of hypervisor platforms available in the market. This period marked the emergence of both open-source and commercial hypervisors, catering to different needs and requirements. Open-source hypervisors like Xen and KVM gained popularity for their flexibility and cost-effectiveness, allowing organizations to build their virtualization solutions. On the other hand, commercial options like VMware ESXi and Microsoft Hyper-V provided robust features and support, making them preferred choices for businesses seeking enterprise-level virtualization solutions. Each hypervisor platform brought its own set of unique features and capabilities, enabling users to select the one that best suited their specific use cases and requirements. This proliferation of hypervisor platforms during this period offered organizations a wide range of options to choose from, allowing them to tailor their virtualization infrastructure to their specific needs.

Containerization technologies, led by Docker, have gained immense popularity in the 2010s and have revolutionized the field of virtualization. These technologies have introduced a new dimension to virtualization by offering lightweight and efficient application isolation and deployment. Containers are specifically designed to package applications and their dependencies, making it easier for developers to create, deploy, and manage applications across different environments. By providing a standardized way to package and distribute software, containerization has greatly simplified the process of application deployment and has enabled faster and more efficient software development cycles. The rise of containerization has also led to the development of container orchestration platforms like Kubernetes, further enhancing the capabilities of containerized applications in terms of scalability and resilience.

Cloud computing has become increasingly popular in the 2010s, with cloud service providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud leading the way. These providers heavily rely on virtualization technologies to deliver scalable and flexible cloud services to their customers.

In recent years, there has been a further evolution of virtualization technologies in the cloud with the emergence of serverless computing platforms and Function as a Service (FaaS). These platforms abstract the underlying infrastructure entirely, allowing developers to run their code without the need to manage virtual machines or containers directly. Serverless computing represents a paradigm shift in cloud computing as it eliminates the need for developers to provision and manage infrastructure resources. FaaS is a key component of serverless computing, allowing developers to write small, event-driven functions that can

be executed in response to specific events or triggers. These functions are stateless and can be scaled independently, making them highly flexible and efficient for handling dynamic workloads. Overall, the combination of virtualization technologies, such as virtual machines and containers, with serverless computing platforms and FaaS has revolutionized the way cloud services are delivered and managed.

In the 2020s and beyond, edge computing has emerged as a key technology for handling the vast amounts of data generated by the Internet of Things (IoT) devices. With edge computing, resources are distributed across edge devices, bringing computation and storage closer to where the data is being generated. This proximity allows for faster processing and reduced latency. However, managing resources in such a distributed environment can be challenging. Virtualization aids in efficient resource management by virtualizing edge resources, ensuring effective allocation of computing power and storage across the network of edge devices. Edge virtualization allows for better utilization of resources, improved scalability, and enhanced flexibility in edge computing environments.

Virtualization technologies have made significant advancements in recent years, particularly in the areas of security and isolation. These advancements have been driven by the growing need to protect sensitive data and applications from various threats. One such feature that has emerged is nested virtualization, which allows for the creation of virtual machines within virtual machines. This provides an additional layer of security and isolation, as each nested virtual machine operates independently from its parent virtual machine. Additionally, hardware-based security extensions, such as Intel SGX, offer enhanced protection by providing secure enclaves within the processor that can be used to isolate sensitive data and code. Furthermore, there have been improvements in the isolation between virtual machines and containers, ensuring that each instance is securely separated from others, preventing any potential breaches or attacks. These advancements in virtualization technologies contribute to a more secure and isolated computing environment, giving organizations peace of mind when it comes to protecting their critical assets.

The historical context and development of virtualization technologies reflect their adaptability and relevance in the ever-changing landscape of computing. From their origins in mainframes to their current role in cloud, edge, and security, virtualization technologies continue to shape modern IT infrastructure and services.

12.2 Hypervisors

A hypervisor, also known as a virtual machine monitor (VMM), is a software or hardware-based technology that allows multiple virtual machines (VMs) to run on a single physical host or server.

Its primary role is to create and manage these virtualized instances, abstracting and efficiently allocating the underlying physical hardware resources. Hypervisors enable the isolation of VMs from each other, making it possible for different operating systems and applications to coexist and run independently on the same physical server.

Key functions and characteristics of a hypervisor include:

- Resource Abstraction: The hypervisor abstracts physical resources like CPU, memory, storage, and network interfaces, presenting them as virtualized resources to VMs.
- Isolation: VMs operate in isolated environments, preventing them from interfering with or accessing each other's data or processes.

- Resource Allocation: Hypervisors manage the allocation of physical resources to VMs, ensuring fair and efficient resource utilization.
- Performance Optimization: They employ techniques to minimize overhead and maximize the performance of virtualized workloads.
- Hardware Emulation: In the case of Type 2 hypervisors, they often rely on hardware emulation to interact with the physical hardware, allowing VMs to run different operating systems from the host OS.
- Management Interfaces: Hypervisors provide tools or interfaces for administrators to create, configure, monitor, and manage VMs.

Table 12.1: Comparative Analysis of Type 1 and Type 2 Hypervisor

Parameters	**Type 1 Hypervisor**	**Type 2 Hypervisor**
Architecture	Runs directly on the physical hardware, without the need for an underlying host operating system.	Runs on top of an existing host operating system.
Performance	Typically offers better performance and efficiency since it has direct access to hardware resources.	Tends to have slightly more overhead compared to Type 1 hypervisors due to the host OS layer.
Use Cases	Commonly used in enterprise environments for server virtualization, data centers, and cloud computing.	Often used for development, testing, and desktop virtualization on personal computers or workstations.
Examples	VMware vSphere/ESXi, Microsoft Hyper-V, Xen, KVM.	VMware Workstation, Oracle VirtualBox, Parallels Desktop
Management	Often managed remotely through a separate management console.	Managed through a user interface on the host OS.
Security	Generally considered more secure due to its minimal attack surface and isolation from the host OS.	May be considered less secure due to its reliance on the security of the host OS.

12.2.1 Comparison of Type 1 and Type 2 hypervisors

Type 1 hypervisor is illustrated in figure 12.3(a). It is technically equivalent to an operating system because it is the independent program running in the most privileged mode. Its role is to provide support for numerous copies of the physical hardware. A type 2 hypervisor, illustrated in figure 12.3(b), is a separate kind. t is a program that uses a regular process to allocate and schedule resources, such as Windows or Linux. The comparative analysis based on various parameters presented in the table 12.1.

Type 1 hypervisors are typically chosen for high-performance server virtualization scenarios where efficiency and security are paramount. They are suited for large-scale virtualization

deployments. Type 2 hypervisors, on the other hand, are more commonly used for desktop virtualization, testing, and development on personal computers where performance and security considerations may be less critical.

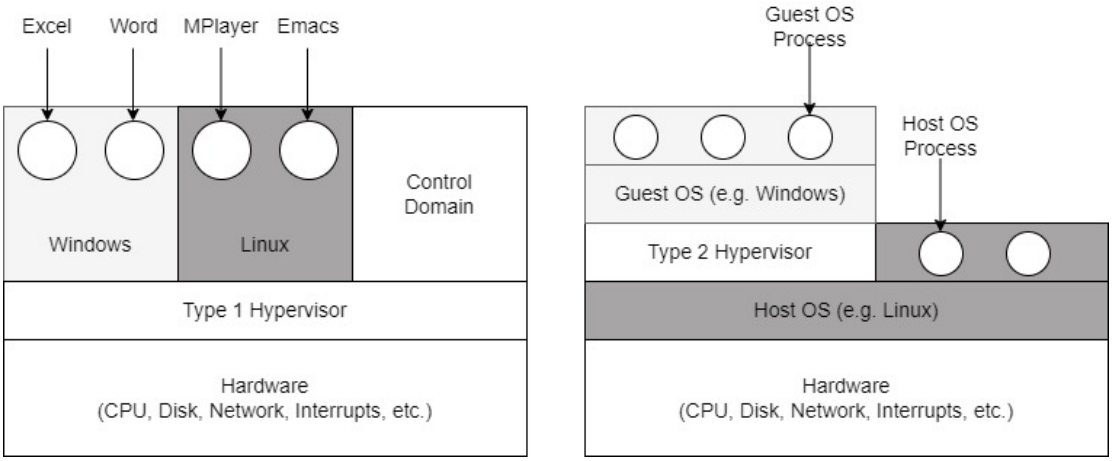


Figure 12.2: Location of type 1 and type 2 hypervisors

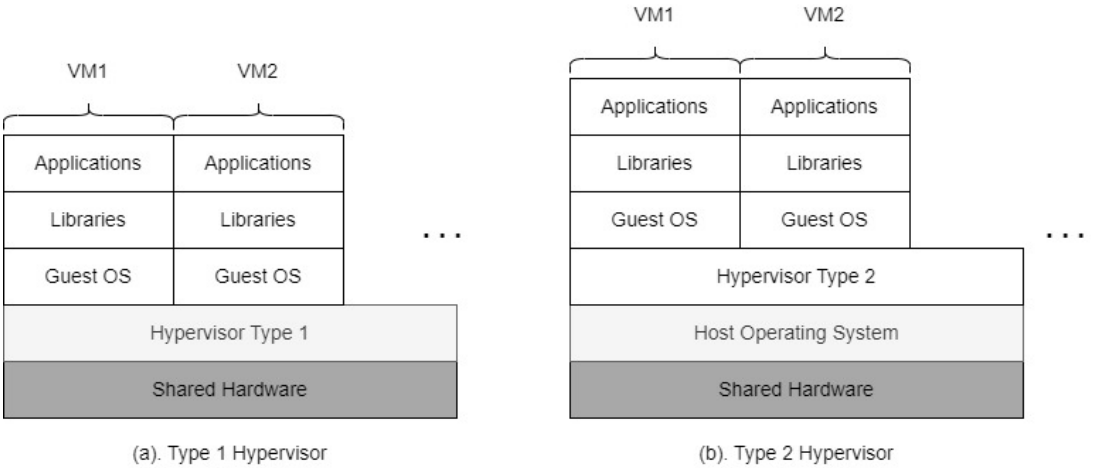


Figure 12.3: Type 1 and type 2 hypervisors

12.2.2 Paravirtualization

Paravirtualization in virtual machines is a technology that offers numerous benefits, including improved performance, efficiency, and security. By enabling direct communication between the guest operating system and the hypervisor, paravirtualization eliminates the need for hardware emulation, resulting in faster execution of instructions and reduced overhead as presented in figure 12.4. This means that virtual machines can achieve near-native performance levels, making them an excellent choice for resource-intensive workloads and high-performance computing environments.

One of the key advantages of paravirtualization is its ability to enhance security in virtual machines. By allowing the hypervisor to monitor and control the actions of the guest operating system paravirtualization provides a higher level of visibility and control. This helps prevent malicious activities and improves overall system integrity, ensuring that virtual

machines remain secure and protected. Paravirtualization also enables better resource management. Virtual machines can dynamically allocate and adjust their resource usage based on demand, allowing for optimal performance and efficient utilization of hardware resources. This flexibility ensures that virtual machines can effectively adapt to changing workloads and allocate resources where they are most needed. It also contributes to cost savings in virtual machine environments. By eliminating the need for hardware emulation, organizations can reduce their hardware costs and achieve better utilization of existing resources. This makes paravirtualization an attractive option for businesses looking to optimize their infrastructure and minimize expenses.

Moreover, paravirtualization offers seamless integration with existing virtualization technologies, making it easy to implement in various environments. This compatibility allows organizations to leverage their current virtualization infrastructure and seamlessly incorporate paravirtualization for even greater performance and efficiency gains. With its ability to work alongside other virtualization technologies, paravirtualization provides a flexible and scalable solution that can meet the diverse needs of different organizations. In addition, paravirtualization also simplifies the migration process in virtual machine environments. With its compatibility and seamless integration, organizations can easily transfer virtual machines between different hosts or platforms without the need for extensive reconfiguration or downtime. This flexibility allows for efficient workload management and enables organizations to adapt to changing infrastructure needs.

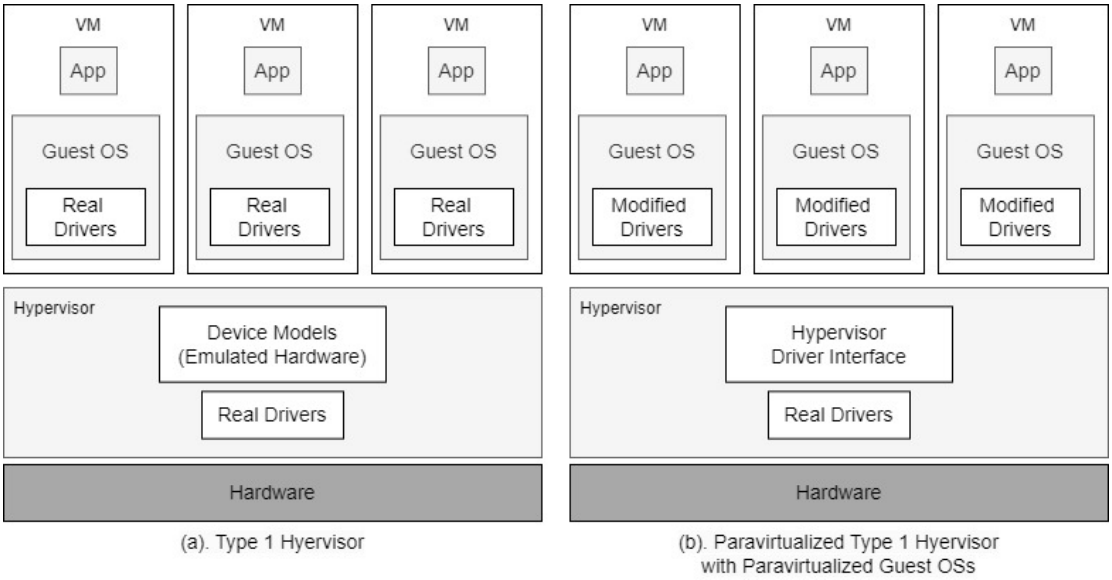


Figure 12.4: Paravirtualization

12.2.3 Hardware-Assisted Virtualization

Hardware-assisted virtualization, also known as hardware virtualization, is a technology that enables the efficient and secure execution of multiple virtual machines (VMs) on a single physical server.

It utilizes special hardware features, such as Intel VT-x or AMD-V, to offload virtualization tasks from the software layer to the hardware layer. This improves performance and reduces

overhead, allowing for better isolation and management of VMs. Additionally, hardware-assisted virtualization provides support for features like nested virtualization, which enables running VMs within VMs, and allows for the use of hypervisors like VMware ESXi or Microsoft Hyper-V. By leveraging specialized hardware features, such as Intel VT-x or AMD-V, virtualization tasks are offloaded from the software layer to the hardware layer, resulting in improved performance and reduced overhead. This technology not only enables the efficient execution of multiple virtual machines on a single physical server but also provides support for advanced features like nested virtualization and compatibility with popular hypervisors like VMware ESXi or Microsoft Hyper-V.

12.3 Container and Docker

Container virtualization is a modern method where a virtualization container operates atop the host OS kernel, creating isolated environments for applications. Unlike hypervisor-based virtual machines (VMs), containers don't mimic physical servers; instead, they share a single OS kernel among all applications on a host. This shared kernel eliminates the need for separate OS instances per application, leading to significantly reduced resource consumption and overhead.

In comparing container and hypervisor software stacks (figure 12.5), containers necessitate only a minimal container engine for support. This engine establishes each container as an independent entity by procuring dedicated resources from the OS. Subsequently, each container application directly utilizes the host OS resources. While specifics vary among container products, a container engine typically executes these tasks:

- Maintaining a streamlined runtime environment and toolset for container, image, and build management.
- Initiating a process for the container.
- Overseeing file system mount points.
- Requesting resources like memory, I/O devices, and IP addresses from the kernel.

Docker offers a more streamlined and standardized method for running containers than earlier versions. These Docker containers, stored as cloud-based images, are easily invoked by users for execution whenever required. It comprises various key components that work together to enable containerization and management of applications. Here's a breakdown of these fundamental components:

1. Docker Engine: This is the core of the Docker platform. It includes the Docker daemon (also known as the Docker Engine) responsible for managing Docker objects like images, containers, networks, and volumes. It operates on the Docker host.
2. Docker Host: The Docker host is the machine (physical or virtual) where the Docker daemon runs. It's the environment where containers are created and managed. The Docker Engine operates on this host, providing the runtime for containers.
3. Docker Client: The Docker Client is a command-line tool that allows users to interact with the Docker daemon through a CLI (Command Line Interface). Users issue commands to the Docker daemon using the Docker Client.

4. Docker Images: Docker Images are the building blocks of containers. They are templates that contain the application code, runtime, libraries, dependencies, and configurations needed to run a specific application. Images are created either from scratch or based on existing images.

5. Docker Containers: Containers are instances of Docker images. They are isolated, lightweight, and portable execution environments where applications and their dependencies run. Each container runs as a separate process on the Docker host.

6. Docker Registry: Docker Registry is a storage and distribution system for Docker images. It is used to store and retrieve Docker images. Docker Hub is the official public registry provided by Docker, while organizations often use private registries to store proprietary or sensitive images.

These components collectively form the Docker ecosystem, allowing developers and system administrators to create, manage, and distribute containerized applications efficiently. The Docker Engine (daemon), Docker Client, Docker Host, Images, and Containers are the core elements, while the Registry serves as the repository for Docker images.

Docker has revolutionized software development and deployment by providing a standardized way to package applications and their dependencies into containers. It offers benefits such as portability, scalability, efficiency, and consistency across different environments, making it a popular choice for modern software development workflows.

12.4 Processor Issue

In a virtual environment, processor resources are managed through two primary methods. Firstly, software emulation (e.g., QEMU, Android Emulator) emulates a chip as software, offering easily transportable resources but with performance inefficiencies due to resource-intensive emulation processes. The second approach allocates processing time segments from the physical processors of the virtualization host to the virtual processors of hosted VMs. This method, employed by most hypervisors, efficiently schedules time on physical processors, intercepts VM OS instructions, executes them on the host's processors, and returns results, ensuring optimal usage of physical server processor resources.

When migrating applications to virtual environments, determining the appropriate number of virtual processors for each VM is crucial. Often, there's a tendency to replicate the processor count from the original physical server, overlooking actual usage and advancements in newer virtualization servers. Tools are available to monitor resource usage and recommend optimal VM sizing, while best practices include starting with one vCPU for a VM and incrementally adding as needed. Overallocation of vCPUs in a VM can negatively impact performance on busy virtualization hosts.

Operating systems manage hardware by mediating between application requests and hardware via device drivers and controllers. They employ protection rings to control access and privilege levels, with the most trusted layer (Ring 0) handling interactions with hardware. Hypervisors, operating in Ring 0, control hardware access for hosted VMs. When a system command is initiated within a VM, the hypervisor intercepts and manages the action, preventing disruptions to the physical server and other hosted VMs.

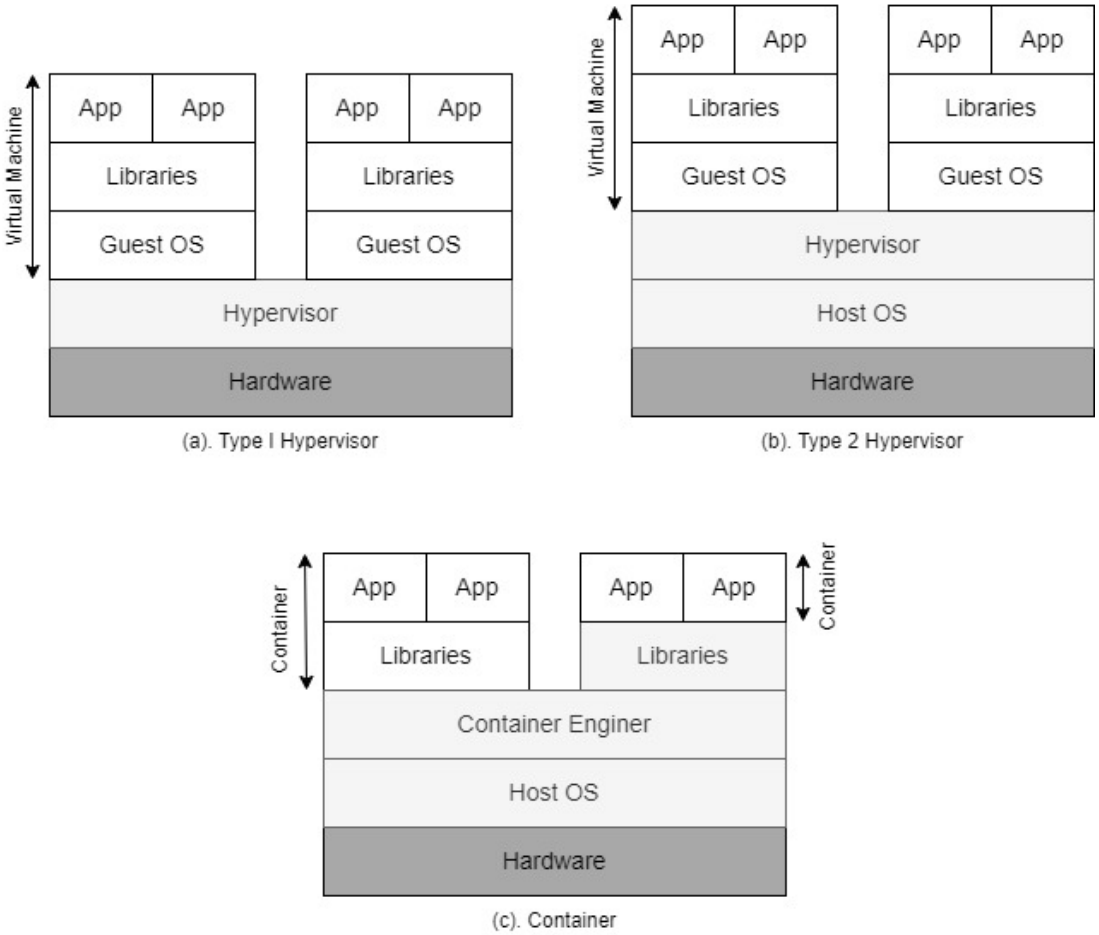


Figure 12.5: Comparison of virtual machines and containers

12.5 Memory Management

Similar to the allocation of vCPUs, the memory assigned to a virtual machine is a critical configuration decision. Memory resources often become the primary bottleneck as virtual infrastructures expand. Memory usage in virtual environments emphasizes managing physical resources over just creating a virtual entity. Adequate memory provisioning for a virtual machine is vital to support the operating system and applications efficiently. Typically, a virtual machine operates with fewer resources than its physical host. For instance, imagine an 16GB RAM physical server hosting a virtual machine provisioned with 2GB of memory. Despite the physical server having more memory, the virtual machine only sees and utilizes 2GB. The hypervisor manages memory requests using translation tables, enabling the guest operating system to access memory addresses as expected. However, issues persist as application owners often request memory allocations mirroring their previous physical infrastructures, leading to overprovisioned virtual machines and wasted memory resources. In the case of the 16GB server, only seven 2GB VMs could be hosted, reserving 2GB for the hypervisor itself. Efficiently sizing virtual machines based on their actual performance characteristics can help alleviate this problem. Hypervisors offer features to optimize memory usage, such as page sharing, akin to data deduplication in storage. This technique identifies and shares duplicate memory blocks among VMs. For example, if multiple VMs load the same OS version or run identical applications, many memory blocks become duplicates. The

hypervisor, managing virtual-to-physical memory transfers, identifies and links shared pages in the VM's translation table.

Given that the hypervisor handles page sharing, the operating systems within virtual machines remain unaware of the activities occurring in the physical system. Another approach, reminiscent of thin provisioning in storage management, involves allocating more storage to a user than the system actually possesses. This strategy aims to set a high mark that is often never reached.

Similarly, in managing virtual machine memory, we allocate 2GB of memory, which is what the VM operating system perceives. However, the hypervisor can repurpose some of the allocated memory for another VM by reclaiming older, unused pages. The hypervisor virtually expanding and pressuring the guest OS to move pages to disk. Once these pages are cleared, the driver contracts, allowing the hypervisor to utilize the physical memory for other VMs. This process occurs during periods of memory contention.

12.6 I/O Management

The performance of an application often depends on the server's allocated bandwidth. When storage access or network traffic is limited, it can make an application seem slower. In virtualized environments, managing input/output (I/O) is crucial. In this setup, the operating system in a virtual machine interacts with a device driver, which then connects to an emulated device managed by the hypervisor (figure 12.6). These emulated devices mimic actual hardware like network interface cards or controllers. The hypervisor's I/O stack helps translate the virtual machine's I/O requests to the physical host's devices, ensuring proper communication between virtual and physical elements. While different vendors might have architectural nuances, the fundamental model remains similar across virtualization setups.

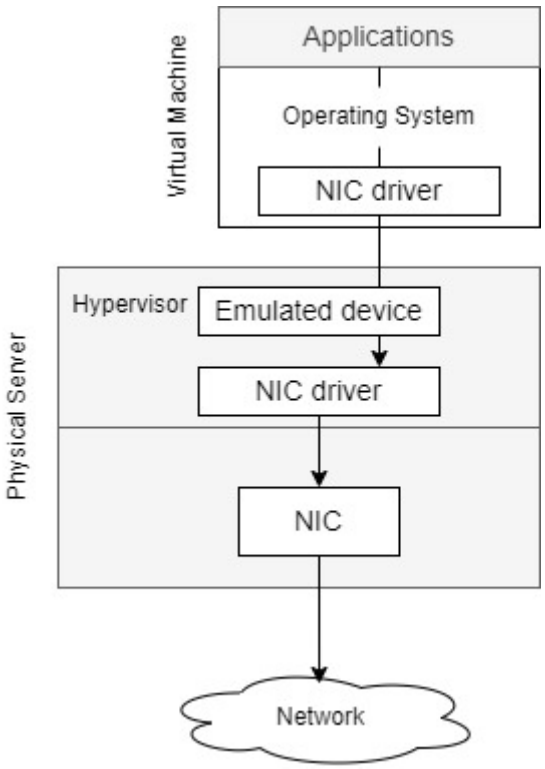


Figure 12.6: I/O in a virtual environment

12.7 VMware ESXi

ESXi, developed by VMware, is a commercially accessible Type 1 hypervisor offering users a bare-metal solution to host virtual machines directly on their servers.

VMware ESXi offers numerous benefits for virtualization, making it a popular choice among businesses and organizations. One of the key advantages is its ability to efficiently utilize resources, allowing for optimal performance and cost savings. By consolidating multiple virtual machines onto a single physical server, VMware ESXi maximizes the utilization of hardware resources, reducing the need for additional servers and minimizing power consumption. This not only saves money but also helps to create a more environmentally friendly IT infrastructure.

In addition to resource utilization, VMware ESXi also simplifies management tasks. With its intuitive interface and comprehensive management tools, administrators can easily deploy and manage virtual machines, reducing the complexity and time required for traditional server management. This streamlined approach allows IT teams to focus on more strategic initiatives, improving overall efficiency and productivity.

Furthermore, VMware ESXi offers increased flexibility in deploying and managing virtual machines. Its robust virtualization platform supports a wide range of operating systems and applications, enabling organizations to run diverse workloads on a single infrastructure. This flexibility allows for better resource allocation and scalability, ensuring that businesses can adapt to changing demands and optimize their IT infrastructure.

Key features of VMware ESXi: VMware ESXi is packed with powerful features that enhance the virtualization experience. One of the standout features is high availability, which ensures that virtual machines remain accessible even in the event of hardware failures. By automatically detecting and recovering from failures, VMware ESXi minimizes downtime and improves the overall reliability of virtualized environments.

Live migration is another key feature of VMware ESXi. With live migration, virtual machines can be moved between physical servers without any disruption to users or applications. This enables organizations to perform maintenance tasks, balance workloads, and optimize resource utilization without impacting the end-user experience.

Fault tolerance is yet another important feature of VMware ESXi. By providing continuous availability for mission-critical applications, fault tolerance eliminates single points of failure and ensures uninterrupted operation. This is achieved by creating a duplicate virtual machine that runs in parallel with the primary virtual machine, constantly mirroring its state. In the event of a failure, the duplicate virtual machine seamlessly takes over, preventing any downtime or data loss.

Lastly, vMotion is a feature that allows for the live migration of virtual machines across different physical servers, even if they have different underlying hardware. This flexibility enables organizations to dynamically allocate resources, optimize performance, and improve overall workload management.

12.7.1 Use cases for VMware ESXi

VMware ESXi is widely used across various industries and scenarios due to its versatility and reliability. In data centers, VMware ESXi is the go-to virtualization platform for consolidating servers, reducing hardware costs, and simplifying management. By virtualizing servers, organizations can achieve higher levels of efficiency, scalability, and agility, enabling them to respond quickly to changing business needs.

Cloud computing is another area where VMware ESXi shines. With its robust virtualization capabilities, VMware ESXi forms the foundation for many cloud service providers,

enabling them to offer scalable and flexible infrastructure as a service (IaaS) solutions. By leveraging VMware ESXi, organizations can easily provision and manage virtual machines in the cloud, allowing for rapid deployment of applications and services.

Software development and testing is yet another common use case for VMware ESXi. By creating virtualized test environments, developers can quickly provision and configure multiple virtual machines, enabling them to test software in different operating systems and configurations. This helps to identify and resolve issues early in the development cycle, improving the quality and reliability of software releases.

Disaster recovery is also a critical use case for VMware ESXi. By replicating virtual machines to a secondary site, organizations can ensure business continuity in the event of a disaster. VMware ESXi's robust features, such as high availability and fault tolerance, play a crucial role in minimizing downtime and data loss, allowing businesses to quickly recover and resume operations.

Performance optimization techniques for VMware ESXi: To maximize the performance of virtual machines running on VMware ESXi, it is important to implement various optimization techniques. Resource allocation is a key aspect of performance optimization. By properly allocating CPU, memory, and storage resources to virtual machines, organizations can ensure that each workload has the necessary resources to operate efficiently. This involves monitoring resource usage, identifying bottlenecks, and adjusting resource allocations accordingly.

Storage configuration is another critical factor in performance optimization. By leveraging features such as storage vMotion and storage I/O control, organizations can optimize storage performance and ensure that virtual machines have fast and reliable access to data. Properly configuring storage arrays, using technologies like RAID and SSDs, can also significantly improve performance.

Network optimization is yet another area to focus on when optimizing performance in VMware ESXi. By properly configuring network settings, such as network adapters, virtual switches, and network load balancing, organizations can ensure that virtual machines have fast and reliable network connectivity. This is particularly important for applications that rely heavily on network communication, such as web servers and database servers.

In addition to these techniques, regularly monitoring and analyzing performance metrics, such as CPU usage, memory utilization, and network throughput, can help identify performance bottlenecks and areas for improvement. By proactively addressing these issues, organizations can optimize the performance of their virtualized environments and ensure a smooth and efficient operation.

12.7.2 Integration with VMware ecosystem

VMware ESXi seamlessly integrates with other VMware products and technologies, forming a comprehensive ecosystem for advanced management, monitoring, and automation capabilities. One of the key components of this ecosystem is vCenter Server, which provides a centralized management platform for VMware ESXi hosts and virtual machines. With vCenter Server, administrators can easily deploy, configure, and manage virtualized environments, simplifying day-to-day operations and improving overall efficiency.

vSphere is another important component of the VMware ecosystem. It extends the capabilities of VMware ESXi by providing additional features and functionalities, such as distributed resource scheduling, high availability, and data protection. By leveraging vSphere, organizations can further enhance the performance, availability, and security of their virtualized environments.

vRealize Suite is yet another integral part of the VMware ecosystem. It offers a comprehensive set of cloud management tools that enable organizations to automate and optimize their virtualized infrastructure. With vRealize Suite, administrators can gain deep insights into resource utilization, automate routine tasks, and ensure compliance with industry standards and regulations.

By integrating with these and other VMware products and technologies, VMware ESXi provides organizations with a holistic solution for managing and optimizing their virtualized environments. This seamless integration enables businesses to leverage the full potential of VMware ESXi and achieve higher levels of efficiency, scalability, and agility.

12.7.3 Security for VMware ESXi

VMware ESXi incorporates several built-in security features that help protect the virtual infrastructure from threats and vulnerabilities. One of the key security features is Secure Boot, which ensures that only digitally signed and trusted components are loaded during the boot process. This prevents the execution of unauthorized or malicious code, reducing the risk of compromise.

Virtual machine encryption is another important security feature of VMware ESXi. By encrypting virtual machine disks, organizations can ensure that sensitive data remains protected, even if the virtual machine is compromised or stolen. This helps to maintain data confidentiality and compliance with industry regulations.

Network segmentation is yet another security consideration for VMware ESXi. By properly configuring virtual networks and implementing firewall rules, organizations can isolate virtual machines and control network traffic, reducing the risk of unauthorized access and data breaches. This is particularly important in multi-tenant environments, where multiple organizations share the same physical infrastructure.

In addition to these built-in security features, organizations should also follow best practices for securing their virtual infrastructure. This includes regularly applying security patches and updates, implementing strong access controls and authentication mechanisms, and regularly monitoring and auditing the virtual environment for any suspicious activities.

By taking these security considerations into account, organizations can ensure that their virtual infrastructure remains secure and protected from potential threats and vulnerabilities.

12.7.4 Migration to VMware ESXi

Migrating from other virtualization platforms to VMware ESXi requires careful planning and execution. The first step in the migration process is to assess the compatibility of existing virtual machines and applications with VMware ESXi. This involves identifying any potential compatibility issues, such as unsupported hardware or software, and finding suitable alternatives or workarounds.

Once compatibility has been established, the next step is to plan the migration strategy. This includes determining the optimal timing for the migration, considering factors such as workload requirements and business priorities. It is also important to define the migration scope, identifying which virtual machines and applications will be migrated and in what order.

During the migration process, organizations can leverage various tools and technologies provided by VMware to simplify and automate the migration tasks. These tools help streamline the migration process, minimize downtime, and ensure a smooth transition to VMware ESXi.

After the migration is complete, it is important to perform thorough testing and validation to ensure that the migrated virtual machines and applications are functioning as expected. This includes verifying performance, functionality, and compatibility with other systems and applications.

By following a well-defined migration plan and leveraging the tools and resources provided by VMware, organizations can successfully migrate to VMware ESXi and take advantage of its powerful virtualization capabilities.

12.7.5 Troubleshooting common issues in VMware ESXi

While VMware ESXi is a robust virtualization platform, users may encounter common issues that require troubleshooting. One common challenge is performance bottlenecks, which can impact the overall performance and responsiveness of virtual machines. To address this issue, administrators can analyze performance metrics, such as CPU usage, memory utilization, and disk I/O, to identify the root cause of the bottleneck. This may involve adjusting resource allocations, optimizing storage configurations, or fine-tuning network settings.

Another common issue is network connectivity problems. This can manifest as slow network performance, intermittent connectivity, or complete network failure. To troubleshoot network connectivity issues, administrators can check network configurations, verify network adapter settings, and test network connectivity using tools like ping and traceroute. They can also examine firewall rules and network security settings to ensure that they are not causing any disruptions.

In addition, administrators may encounter issues with virtual machine stability and reliability. This can include unexpected crashes, freezes, or unresponsiveness. To troubleshoot these issues, administrators can review system logs, monitor resource usage, and check for any conflicting software or driver issues. They can also consider updating VMware tools and virtual machine hardware versions to ensure compatibility and stability.

By addressing these common issues and utilizing the troubleshooting techniques mentioned, administrators can effectively resolve problems in VMware ESXi and maintain a stable and efficient virtualization environment.

12.8 XEN

In the early 2000s, Cambridge University pioneered the Xen open-source hypervisor, which has since spawned various hypervisor variants due to increased virtualization demands. Alongside the open-source version, commercial offerings like Citrix and Oracle have developed their own Xen-based hypervisors. Unlike VMware, Xen operates with a dedicated initial domain, known as Dom0, which manages the hypervisor and has direct hardware access. Multiple Linux versions embed Xen capabilities, allowing the creation of virtual environments. Companies opt for Xen-based solutions due to cost-effectiveness or in-house Linux proficiency. Guest domains on Xen, called DomU, rely on Dom0 for network and storage access via BackEnd drivers (figure 12.7). However, potential issues or maintenance affecting Dom0 can impact all supported virtual machines, as it functions as a Linux instance. Despite lacking some advanced VMware ESXi capabilities, Xen continually evolves with each release, introducing new features and refining existing ones in the open-source realm.

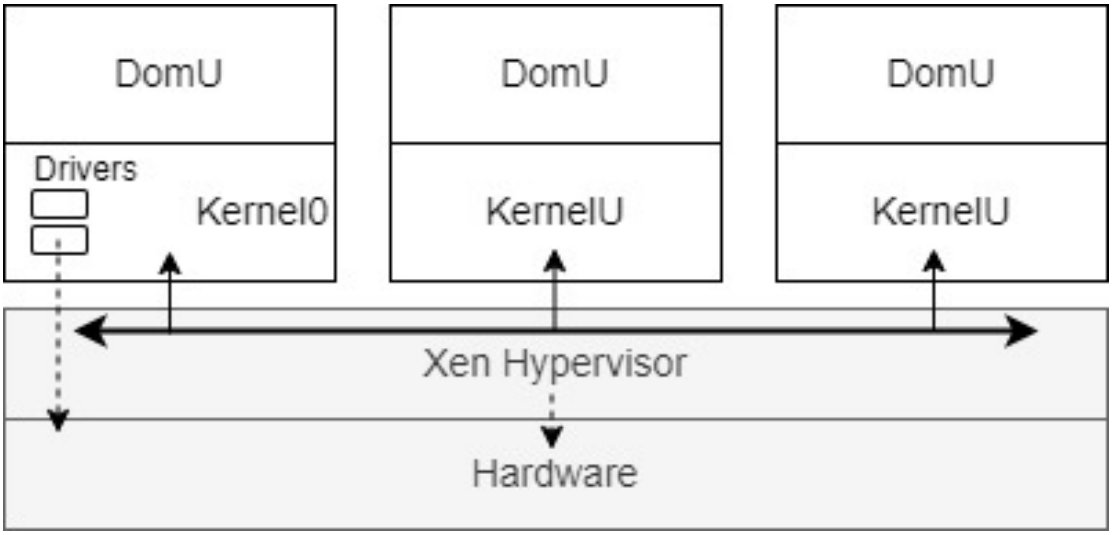


Figure 12.7: Xen

Conclusion: The chapter provides a comprehensive overview of virtualization technologies, tracing their development from mainframe origins to their pivotal role in modern IT infrastructure. The comparison of hypervisors, including their use cases and performance characteristics, offers valuable insights for selecting appropriate virtualization solutions. The discussion on containerization and Docker underscores the shift towards lightweight and efficient application management. Moreover, the integration of VMware ESXi with other VMware products demonstrates the benefits of a cohesive virtualized ecosystem. As the industry progresses, emerging trends such as serverless and edge computing highlight the ongoing evolution and relevance of virtualization technologies in enhancing scalability, efficiency, and resource management across diverse computing environments.

12.9 Exercise

1. Explain the differences between Type 1 and Type 2 hypervisors, highlighting their respective use cases and advantages in virtualized environments.
2. What are the primary advantages of container-based virtualization over traditional hypervisor-based virtualization, and in what scenarios might one be preferred over the other?
3. What is the difference between full virtualization and paravirtualization?
4. What is the role of hypervisors in virtualization?

12.10 Multiple Choice Questions

1. Which of the following best describes virtualization?
   (a) The process of creating a virtual version of a device or resource
   (b) The act of simulating physical hardware to run multiple operating systems or applications

(c) The development of cloud computing technologies

(d) The process of decentralizing computing resources for faster processing

2. What significant advantage does virtualization offer in terms of resource utilization?

   (a) It allows for the creation of hardware-independent applications

   (b) It enables the simultaneous running of multiple operating systems or applications on a single physical server

   (c) It enhances the security features of computing systems

   (d) It facilitates real-time data processing for IoT devices

3. Which historical event marked a significant milestone in the development of virtualization?

   (a) The introduction of Intel VT-x and AMD-V technologies

   (b) The emergence of serverless computing platforms

   (c) The commercialization of hypervisor platforms like CP/CMS

   (d) The rise of containerization technologies led by Docker

4. What is the primary role of a hypervisor?

   (a) To create and manage virtual machines

   (b) To abstract and allocate physical hardware resources

   (c) To provide support for numerous copies of the operating system

   (d) To interact with the physical hardware directly

5. Which of the following is NOT a key function or characteristic of a hypervisor?

   (a) Resource Abstraction

   (b) Performance Optimization

   (c) Interference between VMs

   (d) Hardware Emulation

6. In the comparison between Type 1 and Type 2 hypervisors, which is true?

   (a) Type 1 hypervisors are commonly used for desktop virtualization.

   (b) Type 2 hypervisors are suitable for large-scale virtualization deployments.

   (c) Type 1 hypervisors run as independent programs in the most privileged mode.

   (d) Type 2 hypervisors provide support for numerous copies of physical hardware.

7. What advantage does paravirtualization offer in virtual machines?

   (a) Reduced performance and efficiency

   (b) Increased overhead in executing instructions

   (c) Direct communication between guest OS and hypervisor

   (d) Dependence on hardware emulation for faster execution

8. How does container virtualization differ from hypervisor-based virtualization?

(a) Containers mimic physical servers, while hypervisor-based VMs share a single OS kernel.

(b) Containers utilize a minimal container engine, while hypervisor-based VMs require a separate OS instance per application.

(c) Containers share a single OS kernel among applications, while hypervisor-based VMs each have their own OS instance.

(d) Containers require more resources and overhead compared to hypervisor-based VMs.

9. Which of the following tasks is typically performed by a container engine?

(a) Maintaining multiple OS instances for each application

(b) Initiating a separate process for each hypervisor-based VM

(c) Overseeing file system mount points for container applications

(d) Allocating dedicated resources from the OS for each container

10. What advantage do containers offer in terms of resource consumption?

(a) They require separate OS instances per application, reducing resource consumption.

(b) They utilize a streamlined runtime environment for container management.

(c) They share a single OS kernel among applications, leading to reduced resource consumption.

(d) They allocate dedicated resources from the host OS for each container application.

Deep Science Publishing
https://doi.org/10.70593/978-93-49307-65-0

Chapter 13

CASE STUDY: Linux Operating System

Abstract: This chapter explores the multifaceted world of the Linux operating system, emphasizing its robust security features, efficient process management, and flexible file system architecture. It delves into the Linux kernel's interaction with hardware and software components, ensuring smooth system operations and fair resource allocation. The chapter further contrasts the Linux kernel with the shell, elucidates the hierarchical structure of the Linux file system, and discusses the integral security measures in place to protect against unauthorized access and data loss. Practical exercises and multiple-choice questions are provided to reinforce understanding of key concepts and operational aspects of Linux.

Keywords: Linux Operating System, Linux Kernel, Process Scheduling, File System Hierarchy, User Authentication, Security Updates, Firewall Configuration, Data Integrity, Resource Management.

Linux is a Unix-like operating system kernel developed by Linus Torvalds and released in 1991 under the GNU General Public License. It is open-source and free to use, making it popular for both personal and enterprise use.

Linux is an open-source and community developed operating system. Unlike proprietary systems like Windows or macOS, Linux offers users a level of freedom, flexibility, and customization that is unparalleled. It was originally developed as a hobby in 1991 by the computer scientist Linus T. Torvalds. During his time at the university, Linus wanted to develop an alternative to the Unix operating system. He wanted to create an open-source, free, and community-driven operating system based on the Unix principles and design. As a result, Linux became the most popular operating system on the publicly available internet servers, and the only operating system used on the fastest 500 supercomputers. Linux source code can be modified and made available commercially or non-compromise to anyone under GNU General Public License (GPL). One of the most important features of Linux is its robust security. Linux provides facility of user permissions and access control, this operating system provides a secure environment for your digital activities. Additionally, because Linux is open-source, it benefits from constant community-driven development. It is constantly updating and bugs are quickly identified and patched. Another key advantage of Linux is its stability. Whether it is running on personal computer or at servers in data centers, Linux gives exceptional performance. It can handle high workloads without crashing or slowing

down. Linux supports a wide variety of hardware architectures, including desktop computers, smartphones, embedded systems, and more. This flexibility makes Linux extremely adaptable across a wide range of devices. In addition, Linux users have access to a wide range of software. Few popular software available on linux for different categories like office applications, excel sheets, file manager, audio player, PDF viewer, text editor, video player, and archiving utility and development environment for high-level programming languages like Python and C++. Linux offers wide range of application majority of them are freely available.

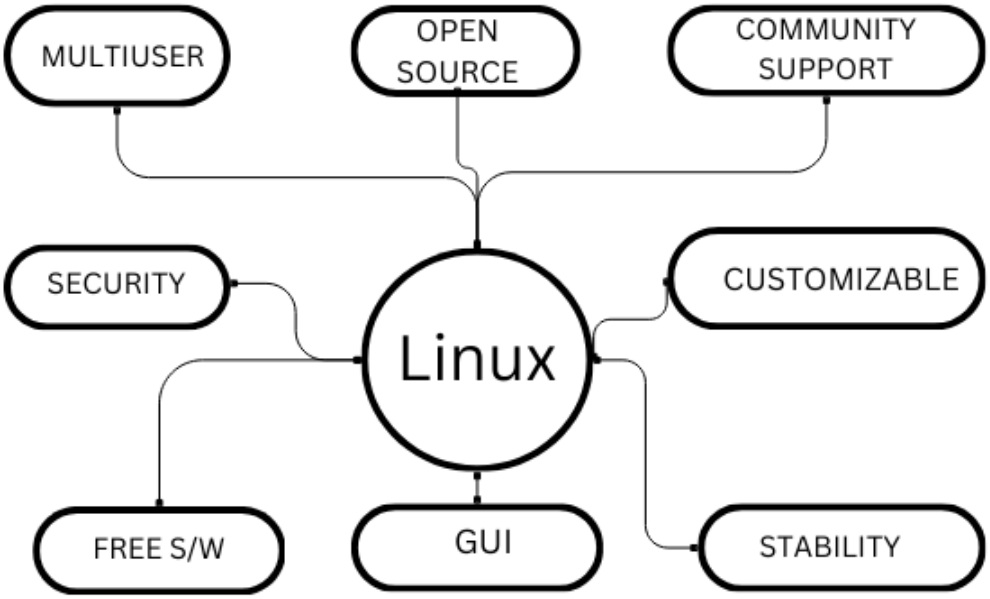


Figure 13.1: Features of Linux

13.1 Features of Linux

An open-source OS, Linux offers a wide range of features that have made it a favorite of developers and tech aficionados alike. Here are some key features of Linux:

1. Security: Linux is known for its robust security measures, with built-in firewalls and encryption tools to protect data from potential threats. Generally Linux does not need antivirus software to be installed on computer. Linux has a lesser chance of being affected by viruses.

2. Stability: One of the best things about Linux is that it is very stable. It rarely hangs or crash. This makes it perfect for operating on critical systems and servers.

3. Compatibility: Linux supports a broad range of hardware devices as well as software applications.

4. Multi-tasking: Linux is having one of the most powerful process management systems. It can run multiple programs at the same time efficiently.

5. User Interface and settings: Linux offers full control over the look and feel of system. From the desktop environment to the system settings, everything can be customized according to needs. Linux has made significant improvement in providing user-friendly interfaces, and distributions like Ubuntu have simplified installation processes, making it accessible even for novices.

6. Open-Source Community: The vast community of dedicated developers behind Linux ensures regular updates, bug fixes, and continuous improvement.

7. Cost-effective: As an open-source OS, Linux is free to use and download—no licensing fees required!

8. Scalability: Whether you're using it on a personal computer or enterprise-level server infrastructure, Linux offers scalability options tailored to meet diverse needs.

9. Reliability: Thanks to its strong foundation in UNIX principles, Linux boasts reliability even under high workloads and demands.

13.2 Linux variants

In addition to its robust features and open-source nature, Linux also offers a wide range of variants that cater to different needs and preferences. These variants often referred to as "distributions". They are customized versions of the Linux operating system developed by various organizations and communities. Some popular Linux distributions include Ubuntu, Fedora, CentOS, Debian, and Arch Linux. Each distribution has its own unique set of features, software packages and user interfaces (UI). Popular Linux Distribution are:

Figure 13.2: Linux variants

- Ubuntu: Ubuntu is one of the most popular Linux distributions in the world. It is known for its user-friendly and easy-to-use interface. It aims to provide a stable and dependable desktop and server environment.
- Fedora: Developed by the community-supported Fedora Project under the sponsorship of Red Hat Inc., Fedora emphasizes cutting-edge technologies and provides frequent updates. It is commonly used by developers due to its focus on innovation.
- CentOS: CentOS is a free, open-source, enterprise-level operating system that is based on RHEL (Red Hat Enterprise Linux). The goal of CentOS is to provide a stable, long-term operating system with enterprise-level support. Many companies choose CentOS because it provides stability without licensing fees.
- Debian: Debian is a Debian-based operating system with a strong focus on free software principles. It prefers stability over cutting-edge features and is the foundation of many other distributions because of its large package repositories.
- Arch Linux: Designed for advanced users who prefer customization options at every level, Arch Linux follows a minimalist approach allowing users complete control over their operating system configuration.

These are just a few examples among hundreds of available Linux distributions catering to specific needs such as gaming (SteamOS), privacy (Tails), lightweight systems (Puppy Linux), server management (OpenSUSE Leap), security-focused environments (Kali Linux), educational purposes (Edubuntu).

Linux is highly customizable and comes in various distributions (distros) tailored to different use cases and preferences.

13.3 Understanding the Linux Architecture

Linux is a Unix-like operating system that is built around a monolithic kernel. This monolithic kernel architecture forms the core foundation of the Linux operating system. The Linux operating system architecture is designed to provide a stable, secure, and flexible platform for running various applications and services. The architecture of the Linux operating system consists of several key components, which work together to ensure efficient and reliable performance. The Linux operating system architecture can be divided into three main layers: the application layer, the middle layer, and the kernel layer(Figure 13.3).

- The application layer is the topmost layer of the Linux operating system architecture. It consists of various user applications that run on top of the operating system. These applications can range from simple command-line tools to complex graphical user interface applications.
- The middle layer of the Linux operating system architecture is known as the application framework. It provides a set of libraries, APIs, and services that enable developers to build and run applications on the Linux platform.
- The kernel layer is the core component of the Linux operating system architecture. It is responsible for managing system resources, such as memory, processes, and device drivers.

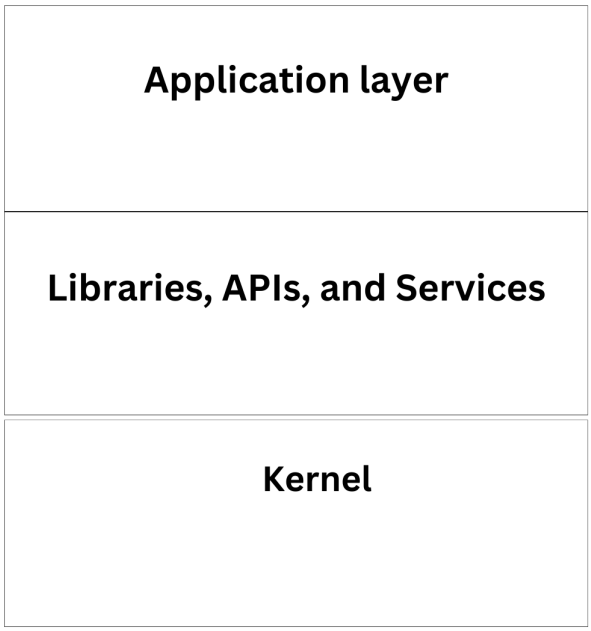


Figure 13.3: Linux Architecture

13.4 Key Components of Linux OS

Within the Linux operating system architecture, there are several key components that work together to ensure the smooth functioning of the system.

1. The Kernel: Heart of Linux System
   The Linux kernel is the core foundation of the Linux operating system architecture. It provides core operating system functionalities such as process management, memory management, device drivers, and handling of system calls. The kernel plays a crucial role in providing a secure and efficient environment for the other components of the operating system to function (Bovet and Cesati, 2002).

2. Role of Shell in Linux Architecture
   The shell is a command-line interface that allows users to interact with the operating system by executing commands and scripts. The shell acts as an intermediary between the user and the operating system, interpreting and executing the commands entered by the user. The shell is responsible for managing input and output, executing user commands, and performing file operations.

13.5 Linux Processes Management

Linux uses a multitasking model for process management. This means that the operating system can run multiple processes simultaneously, dividing the CPU's time among them. The Linux kernel manages processes by utilizing scheduling algorithms to determine which processes get CPU time and for how long. Broadly processes in Linux can be divided into two types:

1. Foreground processes are interactive processes that are executed by the system or by the user, and cannot be initialized by system services. These processes take input and

return output from the user. While foreground processes are running, a new process cannot be initiated directly from the same console.

2. Background processes are non-interactive processes that are run by the system or the user, and can even be managed by the user. Each background process has its own PID or process, and we can start other processes from the same console from which it is running. The background process will remain in a stop state until input is received from the keyboard (usually 'Enter' key), and then it will become a foreground process and be executed. Otherwise, it will be in a stop state.

Processes can also be classified in following 5 types:

1. The process that the user creates on the terminal is called the parent process. If it was created directly by the user, then the parent process is the kernel process.
2. The process that is created by another process is called the child process. For example, the process with PID 28500 (last line) is a child of the process with PID 26544.
3. Orphan processes are created when the parent process is executed before the child process, and the orphan process has "Init" as its parent process ID.
4. Zombie process: The processes which are already dead but shows up in process status is called Zombie process. Zombie processes have Zero CPU consumption.
5. Daemon process: These are system-related processes that run in the background.

Linux process management separates the creation of processes and the running of a new program by two different system calls shown in Figure13.4. A process contains all the

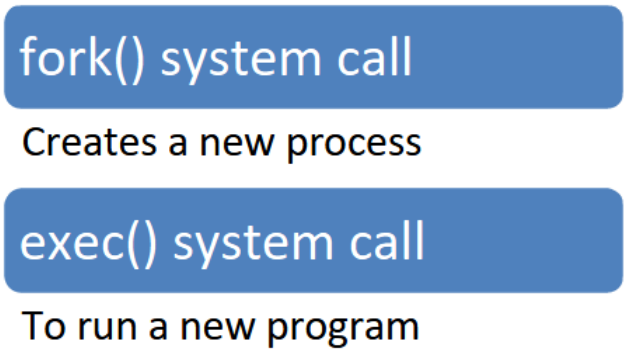


Figure 13.4: Linux system calls for process management

information that the operating system must maintain to track the execution of a single program. Under Linux, process properties fall into three groups: the process's identity, environment, and context.

13.5.1 Linux Scheduling

In scheduling LINUX operating system differentiate two types of processes : Real-time Process and Normal Process. Real-time processes are processes that need to be scheduled exactly based in real time or early. They cannot be delayed in any situation. The difference between a normal process and a real-time process is that a normal process will execute or stop based on the time that the process scheduler has defined. Therefore, a normal process

may experience a delay if the CPU has other high-priority process running. One of the most popular scheduling algorithms is priority-based scheduling. The scheduling algorithm schedule processes based on their priority and how much processing time they require. The Linux kernel implements two separate priority ranges for processes:

1. Real time priority: A range for real time priority is from 0 to 99. Higher real-time priority values correspond to higher priority. All real-time process are higher priority than normal processes. In contrast to nice values, real-time priority and nice values are in different value spaces.

2. Nice Value: A number from -20 to +19, with a default value of 0. The corresponding priority values are 100 (the highest priority value) to 139 (the lowest priority value), since the default base priority value is 120. Larger nice values indicate a lower priority, meaning that you are "being nice" to other processes on your system.

In the LINUX Kernel 2.6, Completely Fair Scheduler (CFS) was introduced. It can schedule the processes within a constant amount of time, regardless of the number of processes running on the system. Active processes are placed into a queue that holds the priority value for each process. The queue is called run queue, and the other queue is a list of expired processes known as expired queue. The expired process is placed in the expired queue when the allowed time of the process expires. Completely fair scheduler (CFS) allocates the CPU resources for executing the process. The main objective of the scheduler is to optimize the overall CPU usage and performance. The CFS scheduler assigns a priority to the interactive tasks and reduces the priority of the non interactive tasks. The recent development in Linux kernel scheduler and micro-kernel development is available in (Song et al., 2023), (Silva et al., 2006).

13.6 Understanding File System in Linux

All general-purpose computers need to store various types of data on hard disk drives (HDDs) or similar devices, such as USB memory sticks. A file system is a way to store and organize data on a computer system. In Linux, a file system is an essential part of the operating system. It is responsible for storing data on disk and on other storage devices. A Linux file system is a system that maintains a hierarchical structure of files and directories(Figure 13.4). This structure makes it easier to navigate and manage the data on the system.
Few important characteristics of LINUX File System are:

1. On a UNIX system, everything is a file, including devices, programs, and system information.

2. The Linux supports multiple file systems, for example: ext4 (Porter et al., 2009), XFS, Btrfs, JFS, and ZFS.

3. Linux uses tree like or hierarchical structure to store files. The hierarchy starts from the root directory (/), which contains all other directories and files.

4. There are many sub directories under root directory, each with its own purpose and set of files.

5. The file system is used to store data in the form of files and directories that can be accessed and changed by users and applications as per file access permissions.

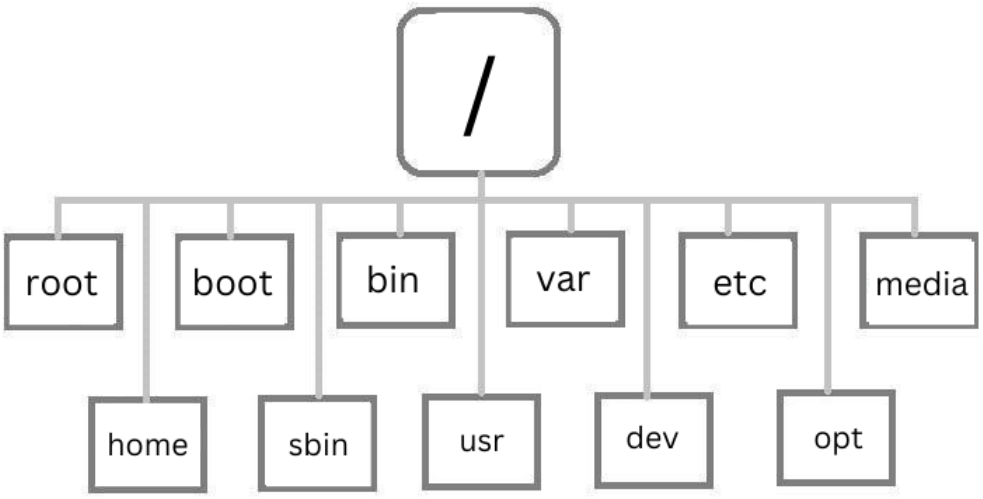


Figure 13.5: Linux file system

6. A partition or logical volume formatted with a specific type of file system that can be mounted on a specific directory called *mount point*.

The various sub directories in Linux file system and their usage is explained in Table 13.1.
Linux also having following files types:

- Directories: files that can store other files and directories.
- Special files: the mechanism used for input and output. Most special files are in /dev.
- Links: A link in UNIX is a pointer to a file. It is a kind of shortcuts to access a file. A soft link is similar to the file shortcut feature which is used in Windows Operating systems.
- (Domain) sockets: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- Named pipes: act more or less like sockets and it is a mechanism by which processes communicates with each other, without using network socket semantics.

13.7 Linux Commands

Linux offers rich graphical user interface (GUI) with powerful command line interface (CUI). The general purpose commands of Linux can be categorized in following categories:

13.7.1 Directory Commands

- Listing files and directories (ls)
  The "ls" command provides a clear overview of files and directories within a specified location. ls also provide following options to like "-l" or "-a" to display permissions, timestamps, hidden files, and more.

- Print working directory (pwd)
  The "pwd" stands for 'print working directory'; It shows your current location on the file system.

- Changing working directory (cd)
  The "cd" is used to change current working directory. With option ".." cd can change current working directory up one level. Current working directory can be moved directly into a specific folder using its absolute or relative path with cd.

- Creating new directory (mkdir)
  New directories can be created using "mkdir" command.

- Removing directory(rmdir)
  The "rmdir" command allows you to remove empty directories from system.

13.7.2 File Commands

- Copy File(cp)
  The "cp" command is used to make a copy of a file.
  SYNTAX: cp [source file name] [new file]

- Move File(mv)
  It is used to to move a file to another location or rename it.
  SYNTAX: mv [file name] [new location]

- Remove File (rm)
  It is used to delete a file.
  SYNTAX: rm [file name]

- Create File(touch)
  "touch" is used to create an empty file.
  SYNTAX:touch [file name]

13.7.3 Filter Commands

Linux offers variety of filter commands for manipulating and filtering data. These commands allow you to process text or data streams and perform specific actions on them. Few commonly used Linux filter commands:

1. grep: This command is a powerful tool for searching text patterns within files or output streams. It allows you to search for specific keywords or patterns, making it useful for extracting relevant information from large datasets.

2. sed: Short for stream editor, sed is a versatile utility that can perform various operations on text files or input streams. It enables you to find and replace text, delete lines based on patterns, insert new content at specific locations, and more.

3. awk: Awk is a scripting language that excels at processing structured data such as CSV files. With its robust pattern matching capabilities and built-in variables, awk allows you to extract columns, modify fields, calculate statistics, and manipulate data in numerous ways.

4. sort: As the name suggests, this command sorts lines of text alphabetically or numerically based on specified criteria such as field position or character position within each line.

5. uniq: Uniq filters out duplicate adjacent lines from sorted input by comparing them line by line.

6. tail/head : The tail command outputs the last part of a file (or standard input), while head displays the first part of a file (or standard input). They are particularly useful when dealing with large log files where you only need to view recent entries or top lines.

These are just a few examples of Linux commands available, for the comprehensive coverage specific books can be referred.

13.8 Linux Security

Security measures in Linux are crucial for safeguarding systems and data from unauthorized access and malicious activities. Here's a short note covering key aspects:

1. User Authentication: Linux employs various authentication mechanisms to control access to the system. This includes traditional methods like username/password authentication, as well as more advanced techniques like public key authentication using SSH (Secure Shell). Implementing strong passwords, enforcing password policies, and regularly auditing user accounts are essential practices for enhancing user authentication security.

2. File Permissions: Linux uses a robust permission system to regulate access to files and directories. Each file and directory has associated permissions for the owner, group, and others, determining who can read, write, or execute them. Administrators should carefully set permissions to ensure that sensitive files are only accessible to authorized users or processes. Regularly auditing and adjusting permissions can help mitigate security risks.

3. Firewalls (e.g., iptables): Firewalls are vital for controlling network traffic and protecting Linux systems from unauthorized access and network-based attacks. iptables is a powerful firewall utility in Linux that allows administrators to define rules for filtering incoming and outgoing traffic based on various criteria such as IP addresses, ports, and protocols. Configuring iptables to block unnecessary network traffic and only allow essential services can significantly enhance the security posture of Linux systems.

4. Security Updates: Keeping Linux systems up-to-date with the latest security patches is critical for addressing vulnerabilities and protecting against known exploits. Linux distributions provide package management tools (e.g., APT, YUM) that facilitate the installation of security updates and patches for installed software. Administrators should regularly check for available updates and apply them promptly to ensure the security and integrity of Linux systems.

By focusing on user authentication, file permissions, firewalls, and security updates, Linux administrators can establish a robust security framework to mitigate risks and safeguard systems against various security threats.

Linux has strong security features, including robust user authentication, granular file permissions, and powerful firewall capabilities.

Conclusion: The chapter provides a comprehensive overview of the Linux operating system, highlighting its significant features and operational intricacies. By focusing on critical areas such as user authentication, file permissions, firewalls, and security updates, it outlines the essential steps Linux administrators can take to establish a robust security framework. The discussion on the kernel and file system offers a clear understanding of their roles and structures, facilitating better management and organization of resources. Through practical exercises and evaluative questions, readers are encouraged to deepen their knowledge and practical skills, ensuring they are well-equipped to manage and secure Linux-based systems effectively.

13.9 Exercise

1. Explain the role of the Linux kernel in the operating system. How does it interact with hardware and software components to facilitate system operations?
2. Discuss the significance of process scheduling in a multitasking operating system like Linux. How does Linux ensure fair allocation of CPU resources among competing processes?
3. Compare and contrast the roles of the Linux kernel and the shell in the operating system.
4. Explain the hierarchical structure of a typical file system, such as the one used in Linux. Describe the purpose and contents of essential directories like /bin, /etc, /home, /var, and /tmp. How does this hierarchy facilitate organization and management of files and directories?
5. Discuss the measures employed by file systems to ensure data integrity and protect against unauthorized access and data loss.

13.10 Multiple Choice Questions

1. Which of the following is NOT a popular Linux distribution (distro)?
   (a) Ubuntu
   (b) Fedora
   (c) macOS
   (d) Arch Linux
2. What is a process in the context of Linux?
   (a) A physical device connected to the computer
   (b) A program in execution
   (c) A folder containing files

(d) A type of user account

3. Which tool is commonly used for firewall configuration in Linux?

   (a) Windows Defender
   (b) Norton Firewall
   (c) iptables
   (d) McAfee Firewall

4. What is the main purpose of the /root directory in Linux?

   (a) To store system configuration files
   (b) To house user data for regular users
   (c) To serve as the root directory of the file system
   (d) To act as the home directory for the root use

5. What is the purpose of security updates in Linux?

   (a) Enhancing system performance
   (b) Installing new features
   (c) Patching vulnerabilities
   (d) Optimizing disk space

6. Which component of Linux is responsible for managing hardware resources and providing essential services to other software?

   (a) Shell
   (b) Kernel
   (c) GUI
   (d) Compiler

Table 13.1: Sub directories in Linux File System

Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, vmlinuz.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting.
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a lost+found in its upper directory. Files that were saved during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command man proc in a terminal window.
/root	The administrative user's home directory. "/root" is the home directory of the root user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

References

Anderson, T. and Dahlin, M. (2014). *Operating Systems: Principles and Practice*. Recursive Books.

Andress, J. (2014). Chapter 11 - operating system security. In Andress, J., editor, *The Basics of Information Security (Second Edition)*, pages 171–187. Syngress, Boston, second edition edition.

Arpaci-Dusseau, R. and Arpaci-Dusseau, A. (2018). *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform.

Arun, C., Gopinath, A., Hanumanthaiah, A., and Murugan, R. (2020). Implementation of direct memory access for parallel processing. In *2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 390–394.

Bach, M. (1986). *The Design of the UNIX Operating System*. Prentice-Hall international editions. Prentice-Hall.

Berghel, H., Hoelzer, D., and Sthultz, M. (2008). Chapter 1 data hiding tactics for windows and unix file systems. In *Software Development*, volume 74 of *Advances in Computers*, pages 1–17. Elsevier.

Bovet, D. and Cesati, M. (2002). *Understanding the Linux Kernel*. O'Reilly Series. O'Reilly.

Buyya, R., Vecchiola, C., and Selvi, S. T. (2013). *Cloud Computing Architecture*, page 111–140. Elsevier.

Deitel (2004). *Operating System*. Pearson Education.

Dhamdhere, D. (2003). *Operating Systems: A Concept-based Approach*. McGraw-Hill.

ISRD (2006). *Basics Of Os Unix And Shell Programming*. McGraw-Hill Education (India) Pvt Limited.

Jaeger, T. (2008). *Operating System Security*. Synthesis lectures on information security, privacy and trust. Morgan & Claypool Publishers.

Liu, B., Wu, C., and Guo, H. (2021). A survey of operating system microkernel. In *2021 International Conference on Intelligent Computing, Automation and Applications (ICAA)*, pages 743–748.

McHoes, A. and Ballew, J. (2012). *Operating Systems DeMYSTiFieD*. Demystified. McGraw Hill LLC.

Mishra, D. and Kulkarni, P. (2018). A survey of memory management techniques in virtualized systems. *Computer Science Review*, 29:56–73.

Panek, C. (2019). *Windows Operating System Fundamentals*. Wiley.

Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. (2009). Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 161–176, New York, NY, USA. Association for Computing Machinery.

Rajagopal, R. (1999). Introduction to microsoft windows nt cluster server: Programming and administration.

Ramamritham, K. and Stankovic, J. (1994). Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2012). *Operating System Concepts*. Wiley Publishing, 9th edition.

Silva, D. D., Krieger, O., Wisniewski, R. W., Waterland, A., Tam, D., and Baumann, A. (2006). K42: An infrastructure for operating system research. *SIGOPS Oper. Syst. Rev.*, 40(2):34–42.

Singhal (2001). *Advanced Concepts In Operating Systems*. McGraw-Hill computer science series. McGraw-Hill Education (India) Pvt Limited.

SINHA, P. (1998). *DISTRIBUTED OPERATING SYSTEMS: CONCEPTS AND DESIGN*. PHI Learning.

Song, Y., Dai, H., Jiang, J., and Zhang, W. (2023). Multikernel: Operating system solution to generalized functional safety. *Security and Safety*, 2.

Stallings, W. (2009). *Operating Systems: Internals and Design Principles*. GOAL Series. Pearson/Prentice Hall.

Tanenbaum, A. and Tanenbaum, A. (1995). *Distributed Operating Systems*. Always learning. Pearson Education.

Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Pearson, Boston, MA, 4th edition.

Wolf, M. (2014). Chapter 4 - processes and operating systems. In Wolf, M., editor, *High-Performance Embedded Computing (Second Edition)*, pages 201–241. Morgan Kaufmann, Boston, second edition edition.

Index