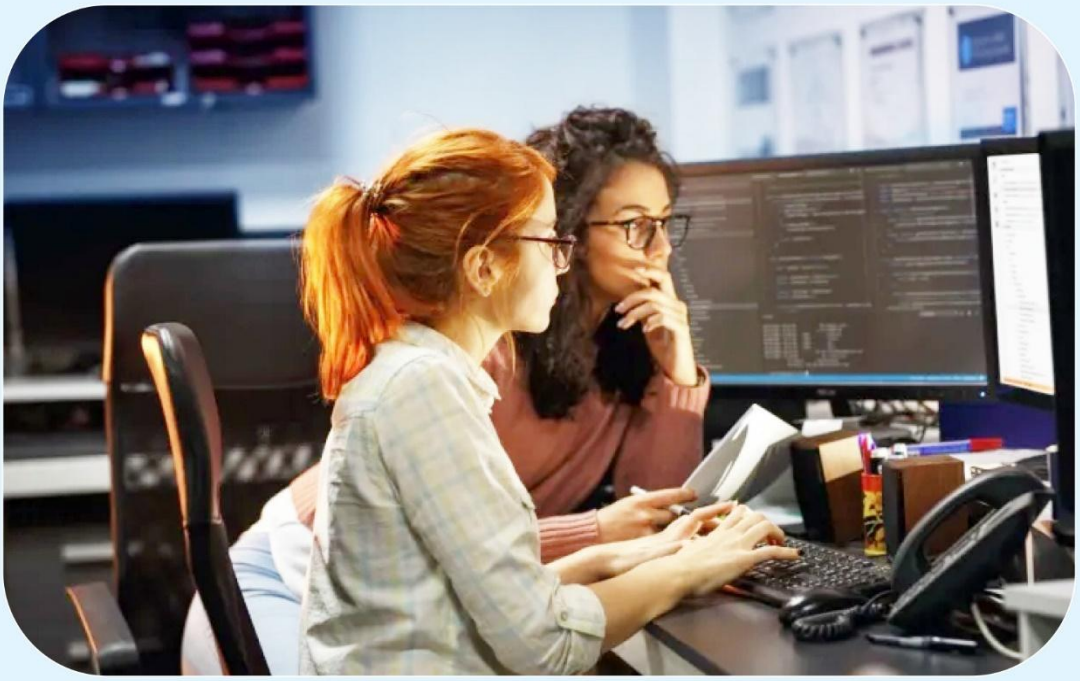
MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR DISTANCE & ONLINE EDUCATION

Operating System Concepts & Shell Programming

Master of Computer Applications (MCA)
Semester - 2



SELF LEARNING MATERIAL

Master of Computer Applications

MCA 203

Operating System Concepts and Shell Programming

COURSE DEVELOPMENT EXPERT COMMITTEE

Prof. (Dr.) K. P. Yadav, Vice Chancellor, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Sanjay Kumar, Professor and Dean, Pt. Ravishankar Shukla University, Raipur, Chhattisgarh

Prof. (Dr.) Jatinder Kumar R. Saini, Professor and Director, Symbiosis Institute of Computer Studies and Research, Pune

Dr. Ronak Panchal, Senior Data Scientist, Cognizant, Mumbai

Mr. Saurabh Chandrakar, Senior Software Engineer, Oracle Corporation, Hyderabad

COURSE COORDINATOR

Dr. Sunita Kushwaha, Associate Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

COURSE PREPARATION

Dr. Sunita Kushwaha, Associate Professor and Mr. Digvijay Singh Thakur, Assistant Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

March, 2025

ISBN: 978-93-49916-81-4

@MATS Centre for Distance and Online Education, MATS University, Village- Gullu, Aarang, Raipur-(Chhattisgarh)

Printed & Published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. Meghanadhudu Katabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

Operating systems (OS) are essential for managing computer hardware and software resources, ensuring efficient execution of applications. This course provides a comprehensive understanding of operating system fundamentals, including process and memory management, file systems, I/O handling, and shell programming. Students will gain both theoretical knowledge and practical skills necessary for OS administration and system-level programming.

Module 1: Operating System Basic Concepts – Overview

An operating system serves as a bridge between users and computer hardware, providing essential functionalities such as resource management, multitasking, and security. This Unit introduces the fundamental concepts, architecture, and types of operating systems, highlighting their role in modern computing environments.

Module2: Process Management and Process Synchronization

Processes are the basic units of execution in an OS. This Unit covers process creation, scheduling algorithms, inter-process communication (IPC), and synchronization techniques. Students will explore concurrency control, deadlock handling, and techniques for efficient process execution in multi-tasking systems.

Module 3: Memory Management

Efficient memory management is crucial for system performance and resource optimization. This Unit explores memory allocation techniques, paging, segmentation, virtual memory, and memory swapping. Students will learn how operating systems manage RAM efficiently to ensure smooth application execution.

Module 4: File Systems and I/O Management

File systems organize and store data systematically in an OS. This Unit covers file system structures, file access methods, disk scheduling algorithms, and I/O management techniques. Students will gain an understanding of how OS handles file storage, retrieval, and peripheral device management.

Module 5: Basics of Shell Programming

Shell programming allows users to automate tasks and interact with the OS using command-line scripts. This Unit introduces shell scripting fundamentals, basic commands, control structures, and script execution. Students will learn how to write shell scripts for system automation and administration.

MODULE 1
INTRODUCTION TO OPERATING SYSTEM

LEARNING OUTCOMES

- To understand the basic concepts of an operating system (OS).
- To explore the need and functions of an OS.
- To analyze different types of operating systems.
- To study OS services and system calls.
- To examine OS structure and design goals.

Unit 1.1: Introduction to Operating Systems

1.1.1 Introduction to Operating Systems

Operating systems are one of the most essential classes of software in computing technology. An operating system (commonly referred to as an OS), on the other hand, is a critical bridge between the computer's physical machinery and the software applications you use on a day-to-day basis. Because modern computing devices possess complex hardware elements—from extremely powerful central processing units to sophisticated memory hierarchies and numerous I/O devices—without an operating system, they would remain haphazard, uncoordinated components that cannot perform useful work.

For example: The operating system is the critical interface that turns hardware into a cohesive, working computing machine, coordinating the myriad interactions between physical resources and software requirements. Operating system is fundamental, it manages computer hardware, provides common services for computer programs, and provide user with an user interface to computer system. They have grown from simple program loaders and memory managers on early mainframe computers into complex working environments, supporting multitasking, multi-user operations and distributed computing over networks.

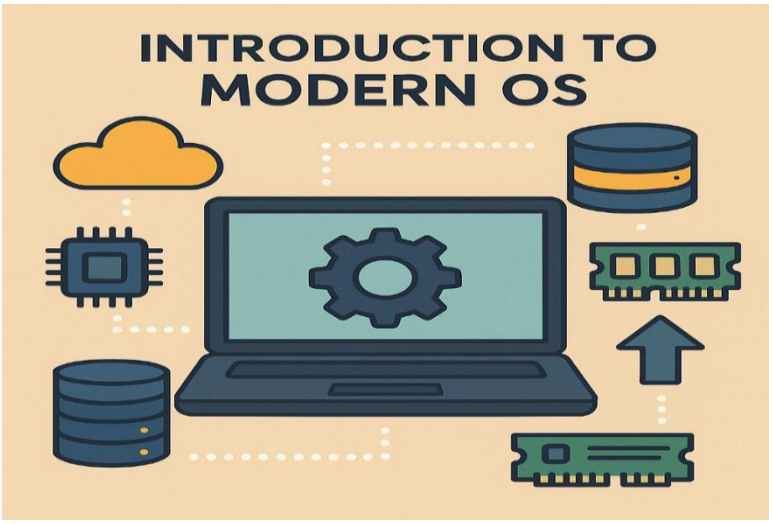


Figure 1.1..1 : Modern operating systems

Modern operating systems, from supercomputers to personal desktop machines to mobile devices and even embedded systems in everyday objects, all share core design and implementation utility principles,

while also tailoring their designs to meet the needs of the hardware environments and use cases they were chosen to serve. The detailed exploration of operating systems provides us not just with the practical knowledge of how computers work at a fundamental level, but also the philosophical considerations concerning resource allocation, security paradigms, and user interface design that have influenced the evolution of computers and continue to shape its path forward.

This course will introduce you to the core concepts, components, and design principles that require the powerful software systems we call operating systems by providing a foundation upon which later study of the specifics of implementations of operating systems and the theoretical underpinnings of those implementations will be built. Operating systems have evolved in much the same way as the computers they serve, progressing in phases that respond to new hardware and new applications. However, the earliest electronic computers of the 1940s and 1950s had no components we'd recognize as an operating system today; these machines had to be run directly by their users, who physically input programs and data with the help of switches, punch cards or paper tape.

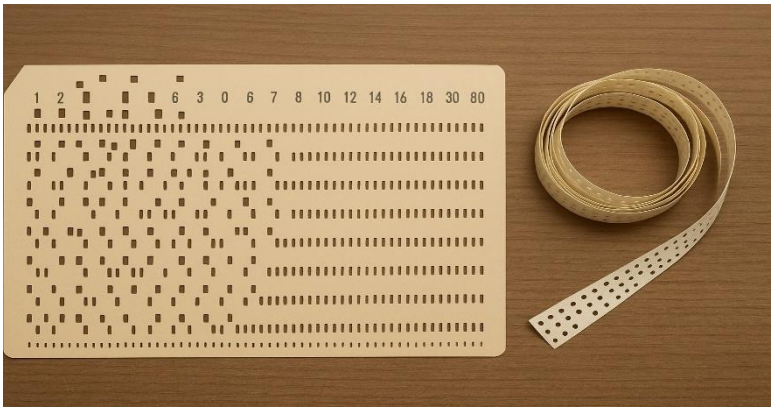


Figure 1.1.2: Punch Cards and Paper Tapes

Programs were fully in charge of the machine while executing, and writing programs required intimate knowledge of the hardware architecture. In the late 1950s, the emergence of batch processing systems marked the beginning of the actual operating system, which automated the loading and execution of series of programs in the background, making use of costly and scarce computing resources by reducing idle time between jobs.

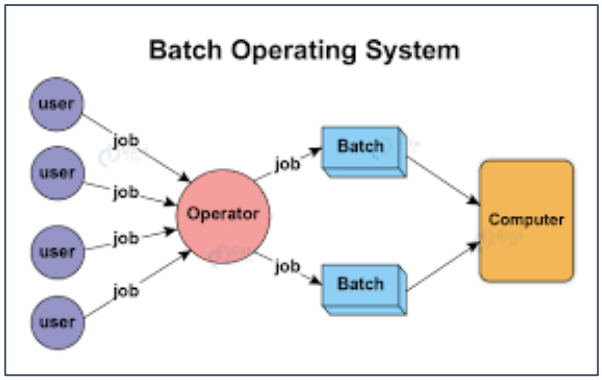


Figure 1.1.3: Batch Operating System

Operating systems It was too complex to trust a single program directly to the hardware, and it generally controlled the execution of one or more workloads, and executed with the allocation of hardware resources in memory and CPUs and allowed multiple users to communicate interactively with the computing environment (time sharing).

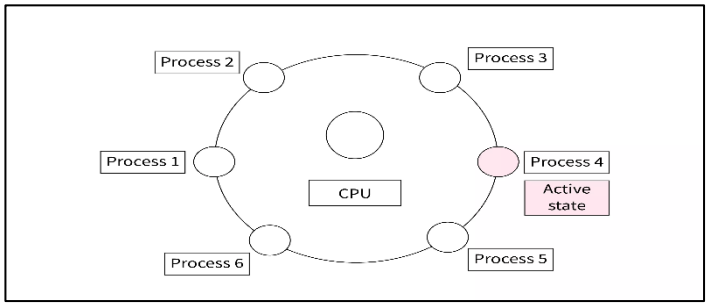


Figure 1.1.4: Time Sharing Operating System

Personal computing rose in the 1980s, still relying on command-line interfaces but making operating systems like MS-DOS, the Macintosh System Software, and various implementations of UNIX available for individual computers, and user interfaces matured into graphical user interfaces as the standard for human/machine interaction in the 1990s, including Microsoft Windows, the Macintosh operating system, and various Linux distributions with desktop environments. The 21st century saw the rise of the networked operating system with focus on internal security and multimedia, as then the mobile explosion of the 2010s saw the birth of new models altogether around touch, battery and connection optimized operating systems like Android and iOS. Cloud computing, virtualization, and containerization take this even further — they extend the operating system to a distributed computing environment where many devices serve as part of a dynamic resource allocation and management pool across a vast web of connected server

infrastructure. Across this evolution, operating systems have always been dealing with basic problems: effectively managing hardware resources, providing developers with layers of abstraction to simplify application development, ensuring the security and stability of the system, and building a user experience that is more convenient points that are still valid no matter the specific implementation or hardware platform used.

You learn from a variety of sources and specializations; however, the definition and understanding of the architecture of an operating system is often vague and can come across confusing to the common reader. At the lowest level, the kernel is the heart of the operating system,

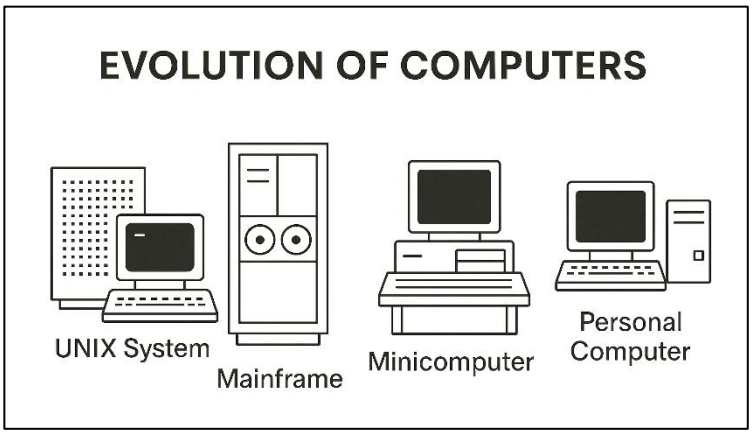


Figure 1.1.5: Evolution of Computer (from CUI to GUI interface)

running in privileged mode with hardware access and handling essential functions such as process and memory management, file systems, device drivers, and inter-process communication. So, amongst these processes, managing their access to the CPU is process management (including process scheduling to give the illusion of concurrency, process creation and termination, process synchronization and communication and context switching on single core systems). Memory management involves mapping virtual addresses to physical memory addresses, allocating and deal locating chunks of memory, maintaining a page or segment table, and providing memory protection against unwanted access. Device Management Device management is a process of controlling hardware peripheral through device drivers that abstract device-specific details and present standardized interfaces, allocate device to competing processes, service interrupts from hardware components. The networking stack is at the core of modern operating systems: it implements communication protocols, manages

local network interfaces, provides socket abstractions for network programming, and handles routing and packet filtering. Security elements are woven throughout the operating system, providing user authentication and user authorization, enforcing access to resources, giving encryption services, and protecting against malware and other security threats. On top of these basic services, modern operating systems provide application programming interfaces (APIs) i.e., methods for applications to request standard services from an OS as well as graphical subsystems to allocate display resources and act as a windowing system, and user interface frameworks to abstract away some of the complexities of building interactive applications.

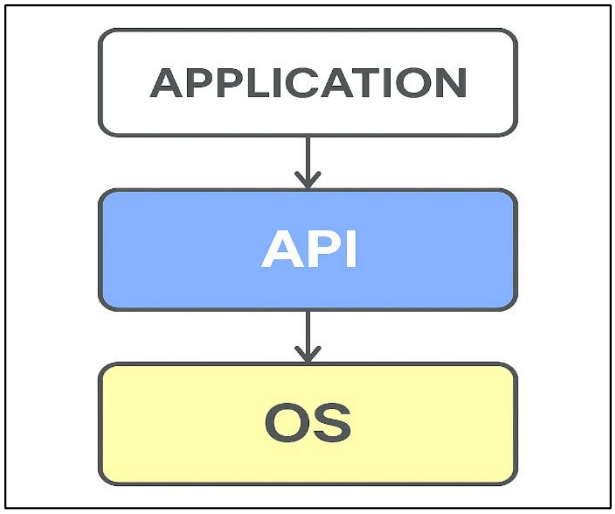


Figure 1.1.6: Application Programming Interface (API)

The operating systems have intricately layered architecture demonstrating several design principles: modularity (capability of constructing components independently and modifying them without having an effect on other components), abstraction (the ability of an operation to hide details of implementation behind the sorts of interfaces that are less complicated) protection (which prevents unauthorized access to resources) and extensibility (refers to the capability of the system to adapt to the ever-changing hardware and software capabilities). This is a crucial aspect of operating system functionality and is used heavily in system performance and resource utilization, as well as the overall user experience. A process, from the operating system's perspective, represents a single task of running a program and includes information not just about the program code, but also the state of the work in progress, including the program counter, register values, values of the program variables, files currently open,

and the program's memory allocations. When a new process is being created—from a user request, an existing process request, or at boot time by the system itself—the operating system allocates all the necessary resources, sets up data structures to keep track of the state of the process, and loads the program code into memory.

A process goes through several states during its lifecycle: running (actively executing on some CPU), ready (waiting to be allocated to a CPU), blocked (waiting for some event, such as I/O completion), and terminated (execution has finished or has been aborted).

For Example: Imagine you send a document to a shared office printer. At first, when you click the print button, the operating system creates a new print job, which represents the **New** state of the process. The job is

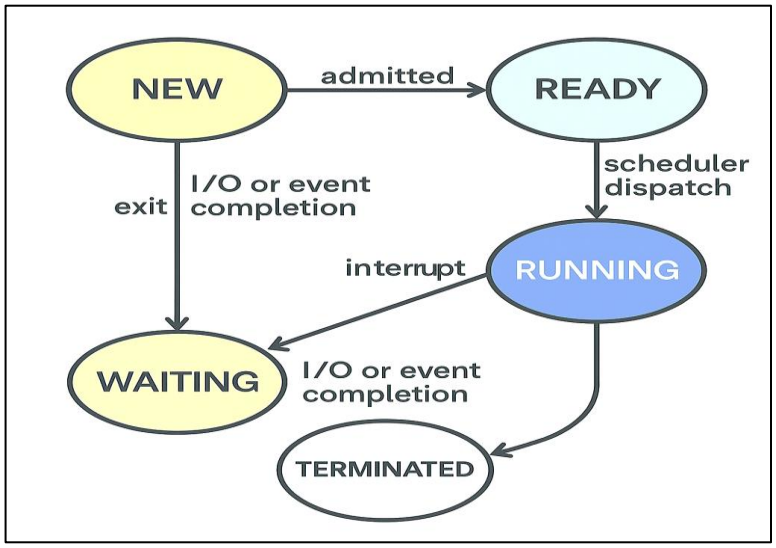


Figure1.1.7: Process Life Cycle in OS

then moved into the **Ready** state, waiting in the print queue because the printer might be busy with other tasks. When the scheduler selects your job and the printer begins to process it, the job enters the **Running** state. Partway through, the printer might run out of paper, which forces the process into a **Waiting** state while it pauses and waits for someone to refill the paper tray. Once the paper is added, the process goes back to the **Ready** state, waiting again for the printer to become available. When the printer resumes and finishes printing your document, the process transitions through **Running** one last time and then ends in the **Terminated** state, as the job is removed from the queue. In this way, the print job's journey mirrors the process life cycle in an operating system. The operating system's scheduler must decide which ready process to run next according to algorithms that balance competing

objectives such as fairness, priority enforcement, response time, throughput, and resource utilization.

Threads are the basic units of execution that share the address space within a process, making it a lightweight alternative to concurrent programming that avoids the cost of a full process creation. Similarly in thread management, but with the added challenge of developing concurrency control and synchronization to avoid race conditions and provide safety in data access to shared resources.

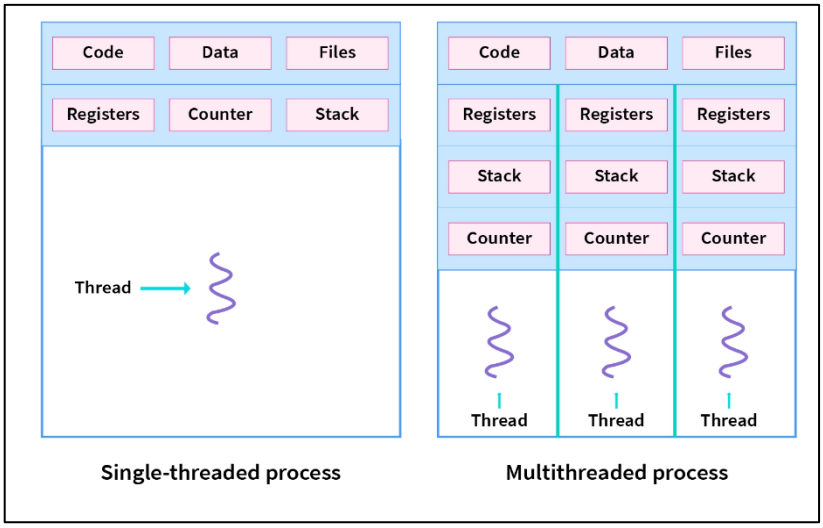


Figure 1.1.8: Thread in OS

To enable cooperating processes to coordinate their action and exchange data, modern operating systems provide several interposes communication and synchronization mechanisms, such as pipes, message queues, shared memory, semaphores, and mutexes.

For Examples: Here are Few real-world examples of Thread system as follows:

Web Browser:

A single browser process can have multiple threads—one for rendering the page, one for handling user input, and others for downloading files simultaneously.

Word Processor:

While typing in Microsoft Word, one thread handles text input, another thread checks spelling and grammar in the background, and another handles autosaving without interrupting your work.

Multiprocessors and multicore systems further complicate process management, as they must also consider processor affinity (keeping certain work units on certain processors in order to make the best use of local caches), balancing the load between multiple processing units, and parallel execution models that take into account the multiple sources of hardware parallelism. These have been developed over years of research, leading to advanced operating system features such as migrating processes between computational nodes in distributed systems, check pointing processes to allow recovery from faults, and dynamic scheduling algorithms that optimize resource allocation according to varying uses of workload and environmental conditions. Process and thread management is critical for the overall performance, responsiveness, scalability and optimal utilization of hardware resources while preserving system stability under different loads.

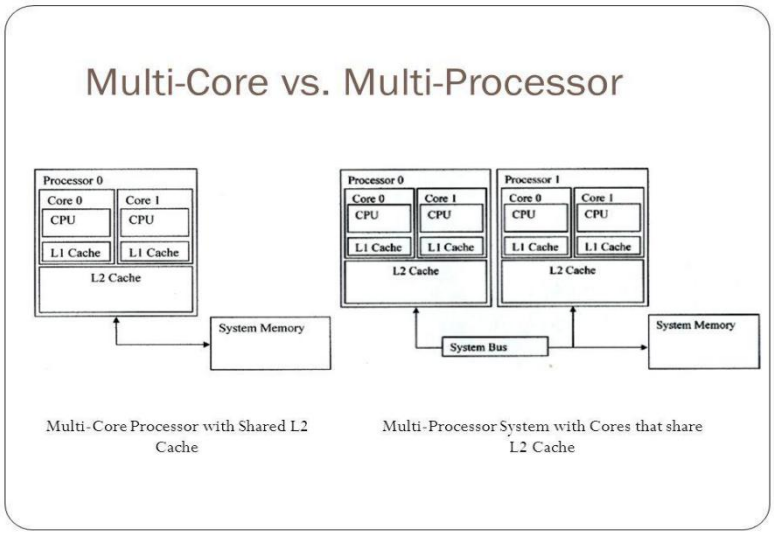


Figure1.1.9: Multi-Processor and Multi-Core Systems

An underlying OS feature that has wide-reaching effects for the performance of the system and the programs running on it, as well as the hardware being used, is Memory management. Perhaps the main issue of memory management is to allocate the available physical memory resources among the various competing processes in a way that users know that their sensitive data is protected and are running in their own "large" address space. Virtual memory: Modern operating systems implement virtual memory systems, which provide an address space for a program that is separate from the physical memory the program runs in. Programs can use this space directly instead of the physical memory they will actually occupy, allowing each program to

think it has more memory than what is available and that it has access to the complete memory space.

This conversion from virtual to physical memory is generally performed by dedicated hardware (the Memory Management Unit, MMU) under the direction of operating system data structures such as page tables.

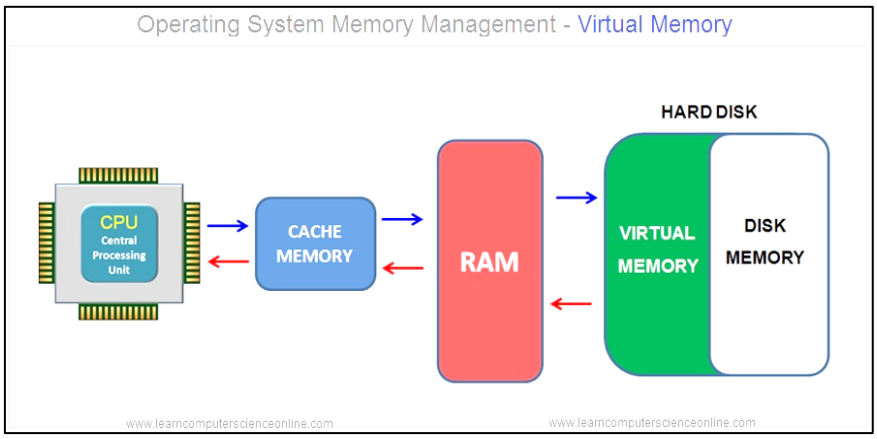


Figure 1.1.10: Virtual Memory System

Paging, which is the most commonly used virtual memory implementation technique, splits virtual memory into equally sized blocks of memory, or pages, and splits physical memory into frames, paving the way for fine-grained memory allocation and efficient allocation of infrequently (but potentially) used pages to secondary storage when physical memory rack is full.

Page replacement algorithms, such as Least Recently Used (LRU), First-In-First-Out (FIFO), and Clock algorithms, decide which pages to evict and when, balancing access frequency, regency, and page fault costs. More advanced memory management strategies include demand paging (paging in memory pages on access), copy-on-write (sharing read-only pages across processes until one of them writes to the page), memory-mapped files (which map file contents directly into a specified portion of the process address space), and large page support (using variable size memory pages so that at least some applications can reduce the translation of pages and fragmentation). Memory protection mechanisms enforce access restrictions to prevent processes from reading or writing to the memory allocated to other processes or (in most cases) the operating system kernel itself, implemented through protection bits in the page tables that are checked by the MMU during address translation. Address space layout randomization (ASLR) is also a technique employed by modern systems that adds another layer of

security of randomly reordering important locations where the program is occupying memory, making it harder for attackers to guess addresses. Advanced memory management features include working set models that attempt to keep a process's most actively used pages in physical memory, non-uniform memory access (NUMA) in multiprocessor systems, where access time to memory varies by distance to memory module, and transparent huge pages that use larger page sizes to reduce overhead for applications with contiguous memory access patterns.

Memory management is a critical part of an operating system design and implementation because it affects not only the speed of program execution, but also the responsiveness of the system, energy consumption, and concurrency in terms of the number of applications that can run without the overhead of pages being constantly made.

For Example: Imagine you are working on a laptop with multiple applications open at the same time — Suppose, a web browser with many tabs, a music player running in the background, and a graphics editing tool. The operating system's memory management is what ensures each application gets enough memory space to function properly without interfering with each other. It allocates specific memory blocks to the browser so it can load web pages, assigns a separate area of memory to the music player so it can buffer audio smoothly, and reserves another area for the graphics tool to handle large image files. If the browser closes a tab, memory management frees that portion of memory so another application can use it. When you switch from editing an image to browsing the web, the system may temporarily move less-used data from RAM to virtual memory on the disk to make room for what you're actively using. Just like a skilled organizer in a shared office, the memory manager keeps track of which areas are in use, which are free, and how to shuffle things around so every application runs efficiently without colliding with others.

The organization of I/O devices and decoupling between software and hardware capabilities and resources is provided by file systems and I/O management systems, which represent one of the most significant parts of an Operating System. I/O management handles the classic problem of presenting abstract, uniform, high-level interfaces to extremely heterogeneous hardware devices ranging from disk drives and network cards to keyboards, display units, and application-specific sensors each with its own timing behaviors, data formats, and control interfaces. Structured to separate and abstract various aspects of I/O, the operating system adopts a layered design approach: the lowest level contains

device drivers, which are hardware-dependent code responsible for interfacing with devices; above that there is a device-independent I/O layer that standardizes the common I/O operations of the same class of devices; and ultimately higher up are high-level interfaces that provide simple abstractions to applications. Depending on the specific implementation, input and output can be either synchronous, where the calling process gets suspended until the operation is complete, or asynchronous, where the process continues executing while the I/O operation completes in the background, and most new systems use an asynchronous model improving system responsiveness and throughput.

The OS uses a number of techniques to improve I/O performance, these include buffering (storing a subset of data in memory temporarily to help speed differences between devices and reducing batch operation timings), caching (storing a copy of recently requested data in memory to quickly access again, reducing access time), scheduling (rescheduling I/O requests in their order to minimize mechanical movements between devices), direct memory access (or DMA, which

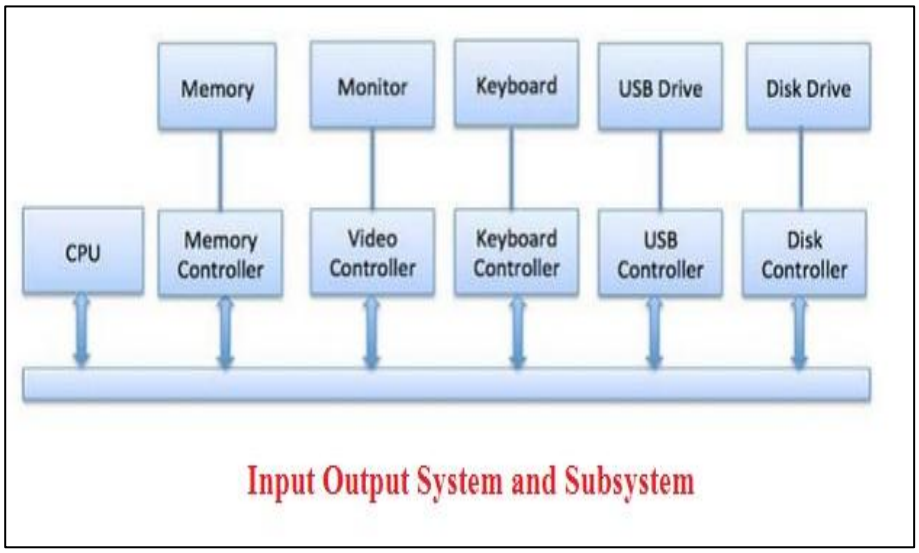


Figure1.1.11: Input Output Sub Systems

eliminates the need of the CPU in facilitating the transfer process between certain devices and memory locations). At the larger scale of I/O, file systems probably offer the most prevalent abstraction: the structuring of persistent storage as named files grouped in hierarchically arranged directory structures. Some more regarding the functionality of file systems they perform one of the most important jobs, map the logical file operations to the storage locations, keep track

of what space is occupied, manage free space, record the metadata for the files (such as creation dates, permission), maintain access control, and ensure data integrity as well through journaling or copy-on-write techniques that protect them from corruption even in case of the system crashing. All modern operating systems support multiple types of file systems, from general-purpose systems (e.g., NTFS, ext4, and APFS) to specialized systems that are better optimized for specific use cases (e.g., high-performance computing, network-attached storage, or even solid-state drives which wear out differently than conventional magnetic media). Some advanced file system features include snapshots (point-in-time captures of file system state), transparent compression and duplication to maximize storage efficiency, and encryption to protect sensitive data, as well as distributed designs that span multiple physical storage devices or network nodes.

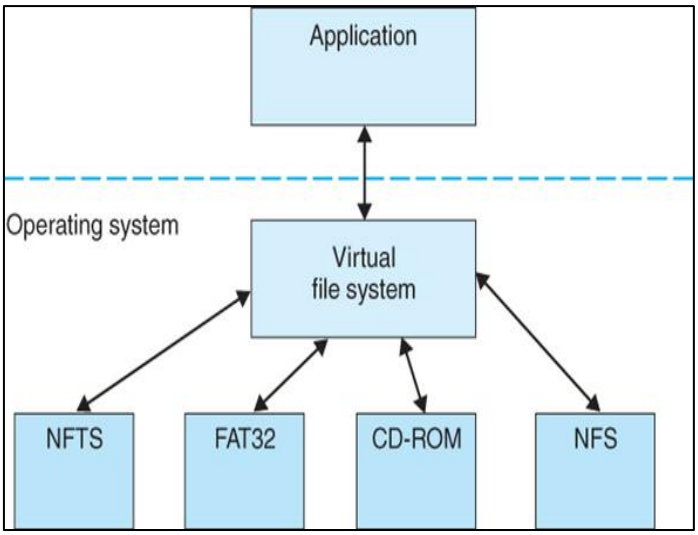


Figure 1.1.12: File System in OS

I/O management and file systems, taken together, allow applications to communicate with the physical world via a multitude of devices and discreetly store and retrieve data without needing to worry about the intricacies of the hardware implementation, making them one of the most useful services provided by the operating system from user and programmer points of view. Security and protection mechanisms pervade modern operating systems, a reflection of the evolution of computing from isolated, single-user systems to interconnected devices containing sensitive information and operating in possibly hostile networked environments. In terms of operating system security, the most fundamental aspect is the separation between user mode and

kernel mode (or supervisor mode) of operation, which establishes a privilege boundary, restricting applications from directly accessing hardware resources or manipulating memory regions not assigned to them, ensuring that transitions between modes are made safe by means of system calls.

For Example: Users are authenticated by passwords, who they are as a person, biometric factors, and cryptographic tokens, and authorization mechanisms then allow or restrict what resources they can access (typically implemented by ACLs or capability-based security models that associate permissions with objects or subjects, respectively). Process isolation ensures that one process cannot access the memory or resources of another process unless explicitly allowed to do so and achieves this through mechanisms such as virtual memory and hardware aids like protection rings or privilege levels. Memory protection takes that isolation even further by applying permissions on these memory regions and marking them either as readable, writable, or executable, while the hardware itself does not allow any operations that bypass these rules, catching a lot of potential attacks at the hardware level itself, before they get a chance to cause damage.

Modern operating systems are not statically defined but are tailored to adapt to the hardware, and use case, and user when the environments evolve, even if not always in a progressive manner, with at least a few of the following trends becoming dominant to each new release within current deployment matrices. Cloud-native OS are a radical break with their predecessors, optimized for the deployment of workloads in virtualized or containerized environments where resources are elastically allocated, workloads are assumed to be distributed over a number of nodes, and system services are accessed through standard APIs to the resources instead of hardware interfaces. On the system design front, containerization and micro services architectures have driven operating system implementation toward a more modular, lightweight style, in which system components are compostable on demand rather than deployed as monolithic images, generating resource overhead and reducing the flexibility to deploy distinct parts of the system independently. Time-sensitive workloads driven by requirements for deterministic performance guarantees and predictable latency even under changing load conditions such as control of autonomous vehicles, industrial automation and augmented reality,

have propelled real-time operating systems (RTOS) into more than just traditional embedded applications. From the design of operating systems, where security in enforcing compatibility has taken precedence over optimizations to rack mount servers behind firewalls in grey rooms, displaying cold metrics in measured temperatures, even to now newer patterns that incorporate validated modules of the OS through axioms or providing remote attestation methods as always verifying your mass surveillance hardware that forces your components on hardware to trust no device only its configurations, all areas throughout computing have been revised and are undergoing a much more rigid recliner to minimize surfaces and reducing injury narratives. At the same time, advances in aggressive power management, workload-aware scheduling, and heterogeneous computing models that partition workloads among the most energy-appropriate processing units are now routine even on cloud computing platforms since user experiences over this wide range of computing have moved now from purely performance-driven to considering price and environmental footprint as primary design considerations. To further mitigate this gap, where CPU throughput is orders of magnitude greater than that of memory, these advanced memory management techniques also permeate both the multi-layer memory hierarchies of DRAM, persistent memory and storage-class memory due to their varying performance characteristics along with sophisticated perfecting and migration policies that predict memory access patterns. Even when it comes to interface paradigms, they are evolving away from solely pointing to desktop metaphors and branching further into the world with conversational interfaces powered by natural language processing, ambient computing models where the interaction takes place through environmental sensors instead of explicit commands, or cross-device experiences where applications and workflows cross-pollinate various hardware form factors. The era of specialized hardware accelerators—Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), Field-Programmable Gate Arrays (FPGAs), and custom Application-Specific Integrated Circuits (ASICs) for specific workloads such as machine learning, cryptography, or video processing—has forced operating systems to create more complex models of resource abstraction and scheduling systems to manage the diversity of computing resources. With the advent of quantum technologies such as

quantum-randomness and quantum-superposition, we will witness the need for new programming models, different resource management strategies, fundamentally different operating systems, and error correction techniques that are going to shape this field for years to come and which need to be explored. Far from converging to a single dominant fruit-of-the-meeting-of-the-twain OS model, these diverse trends point to a continuing diversification of specialized systems, optimized for specific hardware environments, workload characteristics, and usage scenarios, around common theoretical foundations but increasingly differentiated in their implementation details and optimization priorities.

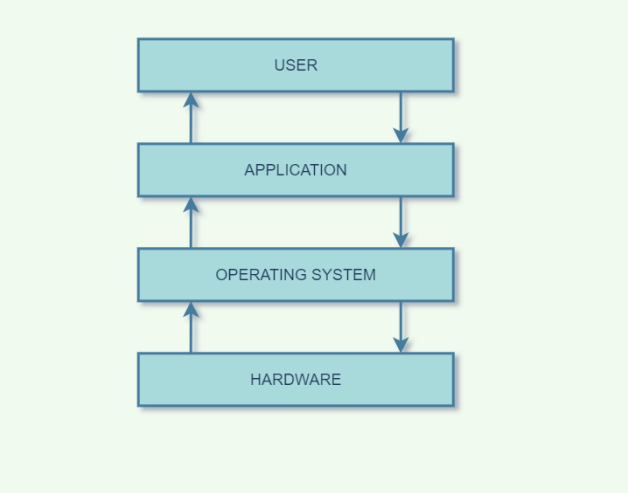


Figure1.1.13: Operating System
[Source - https://medium.com/]

Unit 1.2: Need and Functions of Operating Systems

1.2.1 Need and Functions of Operating Systems

The world of computing we experience today is built on a foundation of ever-evolving hardware and software, all doing its job in concert. Central to this ecosystem is the operating system (OS) a complex piece of software that acts as the vital bridge between computer hardware and the applications that operate on top of it. Operating systems are everywhere, running everything from the smart phones in our pocket to the supercomputer that are behind scientific breakthroughs. But at that same time, operating systems are deeply complex and critical and many users are shaped by their interactions with them without ever fully contemplating the bedrock of their complexity and criticality. Simply put, the operating system is a complex resource mediator and service implementer that exposes a simpler and safer way to run applications on top of real-world hardware. Without the layers of abstraction an operating system provides, any application would have to be responsible for directly manipulating hardware components from managing memory and processing resources to managing input/output operations to peripherals like keyboards, displays, and storage devices. This would result in insane redundancy, bloat, and security holes and would make the application development process a thousand times harder. Operating system development has run side by side with the progress of computing hardware, each generation responding to progressively more intricate problems. Some early computing systems were without operating systems or had very little system software, and so operators had to run the machine and manage the timing of operations manually. As computing power and other capabilities grew, operating systems integrated to manage and protect the more complex resources, becoming the multi-user, multi-tasking familial that they are today and running across everything from embedded micro-controllers to distributed cloud infrastructures.

1.2.2 The Need for Operating Systems: Bridging Hardware and Software

We can also note that modern computing hardware includes many different types of components multi-core, multi-threaded CPUs with multiple, complex version processors in overall instruction set, a

multi-level hierarchy of memory systems (registers, caches, RAM, disk), graphics processing units, networking interfaces, different input/output devices, etc. Each chunk operates on its own protocols and clocks: it's a nightmare of complexity. Without any abstraction, they would have to know all the details of the hardware: its specifications, its operational characteristics, and so on, making it extremely difficult to write software, and making that software only hardware specific. This basic missing piece is supplemented by operating system which provides abstraction layers over the hardware. Training makes a native operating environment by hiding the complexity of the underlying hardware and presents standardized interfaces. The abstraction allows application developers to implement functionality directly as opposed to worrying about the details of the actual hardware. For instance, when an application wants to persist its data, it can use high-level file system calls offered by the OS instead of accessing directly the disk controllers, managing sector allocations or creating error correction protocols.

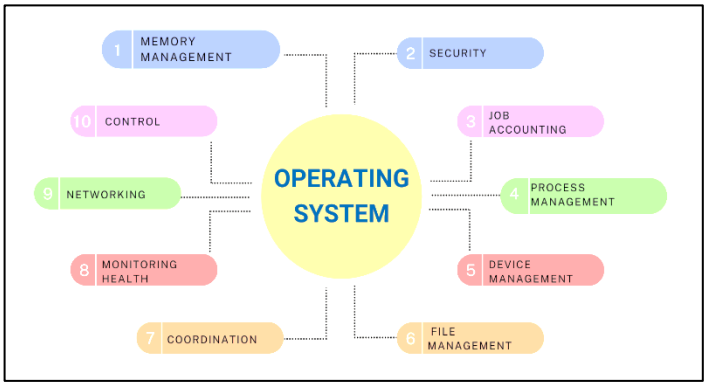


Figure 1.2.1: Functions of OS

1.2.3 Resource Sharing and Protection

Contemporary computing environments generally have multiple applications vying for limited system resources processor time, memory space, I/O bandwidth, and storage capacity. To avoid these conflicts, a mediating system stands between the applications running within the OS. Imagine two different applications wanting to access the same memory region or the same storage space at the same time or one application wants to capture the processor and not allow other applications to run.

The operating system implements mechanisms for resource allocation, scheduling, and protection to ensure:

1. **Fair access to resources**: Through sophisticated scheduling algorithms, the OS ensures that all applications receive appropriate access to the CPU and other resources.
2. **Memory protection**: Modern operating systems implement virtual memory systems that provide each process with its own address space, preventing unauthorized access to memory regions belonging to other processes or the OS itself.
3. **I/O management**: By centralizing control of input/output operations, the OS prevents conflicts in device usage and ensures that all applications can access peripherals in a controlled manner.
4. **File system management**: The OS provides a structured way to store and access data, preventing applications from directly manipulating storage devices and potentially corrupting data.

1.2.4 Hardware Independence and Portability

Operating systems are one of the most valuable software in the information technology world, as they allow portability for software across hardware platforms. Without this layer of abstraction, programs would have to be rewritten for every hardware configuration or platform. The operating system provides standardized interfaces (APIs) which are (for the most part) consistent between different hardware implementations; this enables applications to run on various systems with little or no change. The reason the OS can insulate its applications from the actual hardware of the computer is by taking generic requests made by the application and translating those requests into specific operations on the hardware. As an example, when an application wants to print something, the OS converts that into the printer specific protocol that the connected printer supports. When an application requests memory to be allocated, for example, the OS knows how to abstract the complexity behind managing physical memory resources, including virtual memory systems, paging and address translation.

1.2.5 Security and Access Control

In multi-user and networked computing environments, security concerns become paramount. The operating system plays a crucial role in implementing security mechanisms that protect:

1. **System integrity**: Preventing unauthorized modifications to the system itself.
2. **Data confidentiality**: Ensuring that sensitive information is accessible only to authorized users.

3. **User authentication**: Verifying the identity of users before granting access to resources.
4. **Access control**: Enforcing policies that determine which users can access which resources and in what ways.
5. **Isolation**: Containing potential damage from malicious or malfunctioning applications.

Applications and widespread vulnerabilities. If these protections were not implemented at the level of the operating system, each application would have been responsible for implementing its own security features, which would have had the result of inconsistent protection among Systems, file sandboxing for applications to limit inter-process cooperation, and walking-talking real-time attack monitoring. With computing systems becoming more networked and subject to a greater variety of attacks, operating system security functions became more advanced, adding secure boot procedures, encrypted file

1.2.6 Core Functions of Operating Systems: Process Management

Process Concept and Implementation

A process is the execution of a program, which contains the program code as well as its current activity (it is a unit of work). We focus on processes, one of the most basic abstractions provided by modern operating systems, which enable multi-tasking and a fundamental unit of isolation between executing software.

1. **Program code**: The executable instructions of the program.
2. **Data**: The variables and data structures used by the process.
3. **Process stack**: Containing temporary data such as function parameters return addresses, and local variables.
4. **Process heap**: Dynamically allocated memory during process runtime.
5. **Process control block (PCB)**: A data structure maintained by the OS containing process identification, state information, scheduling information, memory management information, accounting information, and I/O status information.

The operating system is responsible for creating processes when programs are initiated, managing their lifecycle, and eventually terminating them. This lifecycle typically follows transitions between several states:

1. **New**: The process is being created.
2. **Ready**: The process is waiting to be assigned to a processor.

3. **Running**: Instructions are being executed.
4. **Waiting/Blocked**: The process is waiting for some event to occur (such as an I/O completion).
5. **Terminated**: The process has finished execution.

Process Scheduling

Process scheduling is one of the most complex functions performed by operating systems, directly influencing system performance, responsiveness, and resource utilization. The scheduler determines which processes run when and for how long, based on scheduling algorithms designed to meet specific system goals such as:

1. **Maximizing CPU utilization**: Keeping the processor as busy as possible.
2. **Maximizing throughput**: Completing as many processes as possible per unit time.
3. **Minimizing turnaround time**: Reducing the time between process submission and completion.
4. **Minimizing waiting time**: Reducing the time processes spend waiting in the ready queue.
5. **Minimizing response time**: Providing quick initial responses to interactive users.

Operating systems implement various scheduling algorithms to balance these often-conflicting goals:

- **First-Come, First-Served (FCFS)**: Processes are executed in the order they arrive.
- **Shortest Job First (SJF)**: Prioritizes processes with the shortest expected execution time.
- **Priority Scheduling**: Assigns priorities to processes and executes the highest-priority process first.
- **Round Robin (RR)**: Allocates a fixed time slice (quantum) to each process in a circular queue.
- **Multilevel Queue Scheduling**: Partitions the ready queue into separate queues for different process types.
- **Multilevel Feedback Queue**: Similar to multilevel queue but allows processes to move between queues based on their behavior.

Modern operating systems often implement complex hybrid approaches that consider factors such as process priority, execution history, and system load to make scheduling decisions.

Process Synchronization and Communication

In contemporary computing environments, processes rarely operate in isolation. Instead, they frequently need to coordinate their activities and share data. This necessity introduces two critical challenges that operating systems must address:

1. **Race conditions**: When multiple processes access and manipulate shared data concurrently, the outcome can depend on the particular order in which the accesses occur, potentially leading to inconsistent or corrupt data.
2. **Deadlocks**: A situation where two or more processes are unable to proceed because each is waiting for resources held by another process.

Operating systems provide synchronization mechanisms to address these challenges:

- **Mutual exclusion**: Ensuring that only one process at a time can access shared resources or critical sections of code.
- **Semaphores**: Synchronization variables that control access to a common resource in a multi-processing environment.
- **Monitors**: High-level synchronization constructs that encapsulate both the shared data and the operations that manipulate it.
- **Message passing**: Allowing processes to communicate and synchronize by exchanging messages.
- **Deadlock prevention, avoidance, detection, and recovery**: Strategies to handle the deadlock problem.

Inter-process communication (IPC) mechanisms enable processes to exchange information and coordinate their activities:

- **Shared memory**: Allows processes to communicate by reading and writing to a common memory region.
- **Pipes**: Provide a unidirectional communication channel.
- **Named pipes (FIFOs)**: Similar to pipes but with a name in the file system, allowing unrelated processes to communicate.
- **Message queues**: Allow processes to exchange messages through system-provided queue structures.
- **Sockets**: Enable communication between processes running on different machines across a network.

These synchronization and communication mechanisms are essential for building complex, cooperative software systems where multiple processes work together to accomplish tasks.

Memory Management: Optimizing a Critical Resource

Memory Hierarchy and Management Challenges

Many computer memory systems have a hierarchy from fast, but more costly, limited capacity (registers and cache memory) to slower but larger and cheaper (main memory and secondary storage). Memory is a critical resource, and managing its use is paramount to system performance, as access times can vary by orders of magnitude across this hierarchy.

The operating system faces several key challenges in memory management:

1. **Allocation**: Determining how to assign available memory to processes as they are created and as they request additional memory during execution.
2. **Deal location**: Reclaiming memory when processes terminate or explicitly release memory.
3. **Protection**: Ensuring that processes can only access memory allocated to them, preventing unauthorized access to memory regions belonging to other processes or the operating system.
4. **Sharing**: Allowing controlled sharing of memory regions between processes when appropriate.
5. **Physical organization**: Managing the physical arrangement of data in memory to optimize access patterns and utilize memory hierarchy effectively.

1.2.8 Memory Management Techniques

Operating systems employ various techniques to address these challenges:

1. **Contiguous Memory Allocation**: In early systems, each process was allocated a single contiguous block of memory. While simple to implement, this approach led to fragmentation issues and inefficient memory utilization.
2. **Paging**: A memory management scheme that eliminates the need for contiguous allocation by dividing physical memory into fixed-sized blocks called frames and logical memory into blocks of the same size called pages. This allows the physical address space of a process to be non-contiguous, with the operating system maintaining a page table to map logical addresses to physical addresses.

3. **Segmentation**: Divides memory into variable-sized segments, each corresponding to a logical unit of the program such as the code segment, data segment, or stack segment. This approach aligns more naturally with how programmers think about memory but can lead to fragmentation.
4. **Virtual Memory**: An extension of the paging system that allows programs to execute even when they are only partially loaded in memory. The operating system keeps active portions of the program in main memory and transfers other portions between main memory and secondary storage as needed.
5. **Page Replacement Algorithms**: When implementing virtual memory, the operating system must decide which pages to remove from memory when space is needed. Algorithms such as Least Recently Used (LRU), First-In-First-Out (FIFO), and Clock algorithm help make these decisions to minimize page faults.
6. **Memory Compression**: Some modern operating systems compress infrequently used memory pages rather than writing them to disk, reducing the performance penalty associated with page swapping.

Virtual Memory Implementation

Multiple significant advantages: way memory management work. It offers One of the groundbreaking innovations of any operating system design is virtual memory, which changed the whole

1. **Programs can be larger than physical memory**: By keeping only portions of programs in memory, the system can execute programs that are larger than the available physical memory.
2. **Higher degree of multiprogramming**: More programs can run concurrently since each only needs part of its address space in physical memory.
3. **Less I/O for loading and swapping**: Programs can start execution after loading just their initial pages, rather than waiting for the entire program to load.
4. **More efficient use of memory**: Memory is allocated only when needed, not based on worst-case estimates.

The implementation of virtual memory involves several components:

1. **Page tables**: Data structures that map virtual addresses to physical addresses.
2. **Translation Look aside Buffer (TLB)**: A special cache that stores recent address translations to improve performance.

3. **Page fault handling**: When a program accesses a page that is not in memory, a page fault occurs, and the operating system must load the required page from secondary storage.
4. **Swapping mechanism**: The component responsible for transferring pages between main memory and secondary storage.
5. **Working set management**: Tracking the set of pages a process is actively using to make intelligent decisions about which pages to keep in memory.

(loading pages only when accessed), copy-on-write (initially sharing pages until they are modified), and memory-mapped files (mapping file contents directly into virtual memory). For example, modern virtual memory systems tend to contain advanced optimizations like demand paging

1.2.9 File Systems and Storage Management

File Concepts and Organization

Files are the basic building blocks of permanent storage in the computing world. We introduce the core function of the operating system for file management, which provides an essential layer of abstraction that protects applications from handling the details of physical storage devices.

Key file concepts managed by operating systems include:

1. **File attributes**: Information about files, including name, type, size, location, protection settings, creation time, last modification time, and access permissions.
2. **File operations**: Functions such as create, delete, open, close, read, write, append, seek, and get/set attributes.
3. **File types**: Regular files (containing user data or program data), directories (catalogs that organize files), special files (representing devices in UNIX-like systems), and other system-specific types.
4. **File access methods**: Sequential access (reading/writing records in order), direct access (random access to any block), and indexed access (using an index to locate records).

Operating systems organize files using directory structures, which have evolved from simple single-level directories to sophisticated hierarchical structures. Modern file systems implement:

1. **Hierarchical directory structures**: Organized as tree structures with directories containing files and subdirectories.

2. **Path names**: Absolute paths (from the root directory) and relative paths (from the current directory).
3. **Directory operations**: Creating, deleting, opening, closing, and traversing directories.

File System Implementation

The implementation of file systems involves several layers of abstraction:

1. **Logical file system**: Manages metadata information, directory structures, and file control blocks (inodes in UNIX-based systems).
2. **File organization module**: Maps logical blocks to physical blocks, manages free space, and allocates storage.
3. **Basic file system**: Issues commands to device drivers to read/write physical blocks.
4. **I/O control**: Device drivers that communicate directly with storage hardware.

File systems must address several implementation challenges:

1. **Allocation methods**: How to allocate disk space to files:
   - Contiguous allocation: Allocates consecutive blocks, providing excellent performance for sequential access but leading to fragmentation.
   - Linked allocation: Each block contains a pointer to the next block, eliminating external fragmentation but complicating random access.
   - Indexed allocation: Uses an index block containing pointers to data blocks, supporting efficient random access at the cost of additional overhead.
2. **Free space management**: Tracking available storage space using techniques such as bit maps or linked lists of free blocks.
3. **Directory implementation**: Typically implemented as files containing entries that map file names to their metadata.
4. **Efficiency and performance**: Using techniques like block caching, read-ahead, and delayed writes to improve performance.
5. **Recovery mechanisms**: Implementing journaling or other techniques to maintain file system consistency after system crashes.

1.2.10 Advanced File System Features

Modern operating systems implement sophisticated file system features to address evolving needs:

1. **Journaling**: Records changes in a journal before applying them to the main file system, ensuring consistency after crashes or power failures.
2. **Copy-on-write file systems**: Never overwrite existing data, instead writing modified data to new locations and updating pointers, providing snapshots and simplified backup.
3. **Logical Volume Management**: Abstracts physical storage into logical volumes that can span multiple disks and be resized dynamically.
4. **Encryption**: Protecting file contents through transparent encryption/decryption.
5. **Compression**: Reducing storage requirements by compressing file contents.
6. **Deduplication**: Eliminating redundant data to save storage space.
7. **Distributed file systems**: Allowing access to files from multiple hosts over a network.
8. **Object-based storage**: Managing data as objects rather than files or blocks, often incorporating metadata and access methods.

The choice of file system significantly impacts performance, reliability, and functionality. Modern operating systems typically support multiple file system types to accommodate different needs, such as NTFS and ReFS in Windows, ext4 and Btrfs in Linux, and APFS and HFS+ in macOS.

1.2.11 Input/output Systems and Device Management

O Hardware and Challenges

Input/output (I/O) operations are fundamental to computing systems, enabling interaction with users and the external world. I/O devices vary tremendously in their characteristics, presenting significant challenges for operating system design:

1. **Diversity of devices**: I/O devices range from simple character-oriented devices like keyboards to complex block-oriented devices like disk drives, each with different data rates, data formats, and control requirements.
2. **Varied data transfer modes**:

- **Programmed I/O:** The CPU executes instructions that directly control I/O operations.
- **Interrupt-driven I/O:** Devices signal the CPU via interrupts when they complete operations.

- **Direct Memory Access (DMA):** Hardware controllers transfer data directly between devices and memory without CPU intervention.

3. **Performance disparities**: The speed gap between CPU processing and I/O operations (particularly mechanical devices) can be orders of magnitude, requiring sophisticated buffering and scheduling.
4. **Error handling**: I/O operations are prone to various errors (media failures, transmission errors, device unavailability) requiring detection and recovery mechanisms.

Subsystem Architecture

Operating systems implement layered I/O subsystems to manage complexity:

1. **User-level I/O interfaces**: High-level libraries and system calls that provide device-independent interfaces for applications.
2. **Device-independent I/O software**: Performs common functions such as buffering, error handling, and managing device-independent naming.
3. **Device drivers**: Software modules that understand the specifics of particular devices and translate generic I/O requests into device-specific commands.
4. **Interrupt handlers**: Manage device interrupts, acknowledging completion of I/O operations and initiating next steps.
5. **Hardware**: The actual I/O devices and their controllers.

This layered approach provides several benefits:

1. **Device independence**: Applications can use generic I/O operations without concerning themselves with device specifics.
2. **Uniform naming**: Devices can be accessed through a consistent naming convention, regardless of their physical characteristics.
3. **Error handling**: Errors can be managed at appropriate levels of the hierarchy.
4. **Synchronous and asynchronous I/O**: Support for both blocking operations (where the process waits for completion) and non-blocking operations (where the process continues execution while I/O proceeds).
5. **Buffering**: Managing data transfer rate mismatches between devices and processes.
6. **Spooling**: Handling devices that can serve only one process at a time, such as printers.

1.2.12 I/O Performance Optimization

Operating systems employ numerous techniques to optimize I/O performance:

- **Caching**: Keeping recently accessed disk data in memory to reduce access times for subsequent requests.
- **Buffering**: Using memory areas to temporarily hold data during transfers, accommodating speed mismatches and allowing for more efficient batch processing.
- **Scheduling**: Reordering I/O requests to minimize movement in devices with mechanical components (such as disk head scheduling in hard drives).
- **Request merging**: Combining adjacent requests to reduce the number of separate I/O operations.
- **Anticipatory I/O**: Predicting future I/O requests based on observed patterns and prefetching data.
- **I/O parallelism**: Using techniques like RAID (Redundant Array of Independent Disks) to spread I/O operations across multiple devices.

1. **Quality of Service (QoS)**: Ensuring that critical I/O operations receive priority treatment.

of modern operating systems evolves to support new hardware types and connection types. The I/O subsystem

1.2.13 Security, Protection, and Advanced OS Functions

Security Fundamentals and Implementation

Functions will be implemented at several levels by modern operating systems: have become increasingly inter-connected and they store and process sensitive information, the security of the operating system has become even more important. Security As computing systems

1. **Authentication**: Verifying the identity of users through methods such as:
   - Password-based authentication
   - Multi-factor authentication
   - Biometric authentication
   - Token-based authentication
   - Certificate-based authentication
2. **Authorization**: Determining what authenticated users are permitted to do, typically implemented through:
   - Access control lists (ACLs)

- Role-based access control (RBAC)
- Mandatory access control (MAC)
- Capability-based security models

3. **Cryptographic services**: Providing encryption, decryption, and cryptographic hashing functions to:
   - Protect data confidentiality
   - Ensure data integrity
   - Verify the authenticity of software and communications
4. **Process isolation**: Preventing processes from interfering with each other or with the operating system itself through:
   - Memory protection mechanisms
   - Hardware-supported privilege levels
   - Containerization
   - Virtual machine isolation
5. **Security monitoring and auditing**: Detecting and logging security-relevant events to:
   - Identify attempted breaches
   - Support forensic analysis after security incidents
   - Provide accountability and non-repudiation
6. **Secure boot processes**: Ensuring that only authenticated and unmodified operating system components are loaded during system startup.

Deployment environment. These security mechanisms need to find a trade-off between protection, usability, performance, and manageability, which often leads to complex trade-offs depending on the security needs of the

1.2.14 Virtualization and Containerization

Virtualization has transformed modern computing by allowing multiple operating systems to execute simultaneously on a single physical machine, while containerization offers lightweight abstraction for applications running in the same operating system instance.

Virtualization refers to the creation of virtual (rather than actual) versions of computing resources, implemented through:

1. **Hardware virtualization**: Using a hypervisor that:
   - Presents virtual hardware interfaces to guest operating systems
   - Manages resource allocation between virtual machines
   - Provides isolation between virtual environments
   - Types include:

- Type 1 (bare-metal) hypervisors that run directly on hardware
- Type 2 hypervisors that run on top of a host operating system

2. **Para virtualization**: Where guest operating systems are modified to use special APIs for improved performance.
3. **Memory virtualization**: Techniques such as shadow page tables or hardware-assisted memory virtualization that manage the mapping between guest physical addresses and host physical addresses.
4. **I/O virtualization**: Methods for sharing physical I/O devices among multiple virtual machines.

Containerization provides application isolation without the overhead of full virtualization by:

1. **Sharing the host operating system kernel** while providing isolated userspace environments.
2. **Using namespace isolation** to separate container process trees, network interfaces, mount points, and user IDs.
3. **Employing resource control mechanisms** like cgroups to limit and account for resource usage.
4. **Providing standardized image formats** and deployment mechanisms.

Both virtualization and containerization have become fundamental technologies in cloud computing and modern application deployment strategies, enabling more efficient resource utilization, improved isolation, and greater flexibility in application hosting.

Distributed Operating Systems and Cloud Infrastructure

Contemporary computing increasingly spans multiple physical systems, leading to the development of distributed operating system concepts and cloud computing infrastructures:

1. **Distributed operating systems** extend operating system functions across multiple physical machines:
   - **Transparency**: Hiding the distributed nature of the system from users and applications
   - **Communication**: Low-level message passing and higher-level remote procedure calls
   - **Process migration**: Moving processes between nodes for load balancing
   - **Distributed file systems**: Providing a unified file namespace across machines

- **Distributed synchronization**: Mechanisms for coordinating activities across nodes
- **Fault tolerance**: Handling node failures gracefully

2. **Cloud computing infrastructure** builds on virtualization and distributed systems concepts to provide:
   - **Infrastructure as a Service (IaaS)**: Virtualized computing resources
   - **Platform as a Service (PaaS)**: Runtime environments for applications
   - **Software as a Service (SaaS)**: Complete applications delivered over the network
   - **Elasticity**: Dynamic scaling of resources based on demand
   - **Resource pooling**: Sharing physical resources among multiple tenants
   - **Measured service**: Tracking resource usage for billing and optimization
3. **Emerging operating system paradigms** adapt to these distributed environments:
   - **Microkernel architectures**: Minimizing kernel code and moving functionality to user space
   - **Unikernel approaches**: Creating specialized single-purpose applications that include only the OS functionality they need
   - **Server less computing**: Further abstracting infrastructure management away from application developers

Operating systems advanced to support more complex applications, requiring features like these to support interconnected and orchestrated systems over the networks built up around computers as they became pervasive. This is an example of how want to understand modern computing systems and/or build software that interacts with them in a meaningful way. Providing abstractions to simplify writing applications, mechanisms that guarantee your applications utilize resources as required, and protections to enable safe and reliable computing; as we have discussed throughout this book. It is vital to understand these basics and their functions if you the operating systems are the ultimate base on which all other software.

Unit 1.3: Computer System Operations

1.3.1 Computer System Operations

At the core of contemporary computing lies a complex choreography involving the cooperation of myriad hardware and software elements, working in unison to perform tasks from basic arithmetic to sophisticated data processing. Computer system operations involve how computers organization works under the guidance of standard processes that describe the operational condition of the system which is well defined by the standards represented through several protocols and architectures A computer system is built on top of four main layers: hardware, software, data, and users, with components of each layer communicating via designed interfaces and communication channels. These pieces of hardware share different characteristics and performance specifications the CPU, memory, hard drives, I/O interfaces, network devices etc.

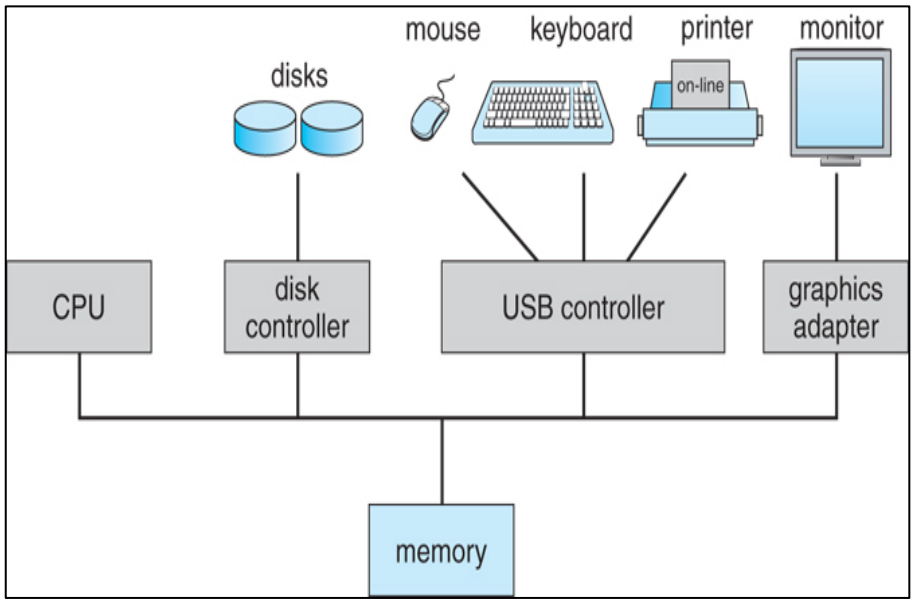


Figure 1.3.1: Computer System Operations

Hardware resources, user applications, operating system, utility programs, development tools, and application software to carry out certain tasks. Data lives in different states (input, information, layout tables, files, streams) across a variety of protected boundaries and access patterns within computer systems. Interfaces facilitate interaction between users, be they human operators or automated systems, and the individual components, translating intentions into actionable commands. The details of how these bits and pieces

cooperate are the domain of the science of computer system operation: processor scheduling algorithms, methods of memory management, input/output operations, file system organization, network communications protocols, all of the moving parts of physical computers at the core of these systems. System architects, software developers, IT administrators, and computer scientists use this information regarding operational frameworks to improve performance, security, reliability, and design new computing paradigms. By now, the paradigm of present-day computational environments has significantly changed from independent, self-sufficient computing units to interrelated, systematized infrastructures, that consist of cloud computing, virtualization, containerization, edge computing, and several challenges that come with them all. With computers increasingly dominating all facets of society, whether that is through corporate processes, scientific investigation, social connection, or entertainment, the importance of fast, secure, and reliable computation is further amplified. Inhabiting a singular text but spanning through many disciplines, this textbook explores what makes computer systems operate; it analyzes not only the theory behind it, but also the practical considerations and approaches to ensure that our computing systems operate as intended. Learning about how a computer system works at the low level enables students and professionals to create more efficient systems as well as solve complex problems, design new solutions, and help to push computing technology forward to meet the demands of modern society in ways that were previously unimaginable or impossible.

1.3.2 Processor Management and Scheduling

Processor management is the kernel of computer system operations a complex system of mechanisms that control the execution of instructions on a system's central processing units. The architectures of modern CPUs involve multiple cores, instruction pipelines, branch prediction, speculative execution, and multiple levels of cache. For this purpose, a CPU operates on a loop: fetches instructions from memory, decodes what the instruction means and executes operations on data as per the instruction. The control unit attempts to synchronize the time of operations and make sure the instructions executed through the arithmetic logic unit and registers inside the processor. Operating

systems use processor scheduling algorithms to decide which processes get CPU time and in what order, essentially juggling multiple requests for this scarce resource.

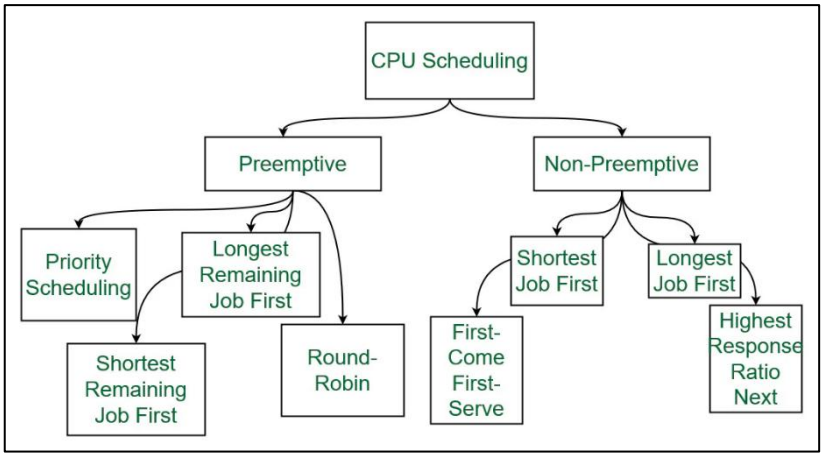


Figure 1.3.2: Process Management and Scheduling

Basic scheduling is about what to run processes, which are instances of programs that are in execution, along with the state for the execution of that process, such as the value of the program counter, the registers, stack, and memory that those processes have allocated, and the resources that such processes are utilizing. A process has different states in its life cycle: new (has been created), ready (waiting for CPU time), running (currently executing), waiting (waiting for an event or resource), and terminated (finished executing). The operating system kernel includes a scheduler that decides which processes get to execute using an advanced algorithm that strives to optimize a particular metric for the system. The simplest scheduling algorithm, First-Come-First-Served (FCFS), executes processes in the order they arrive in the ready state, assuring fairness but allowing short processes to be delayed by long-running ones, a problem called the convoy effect. The Shortest Job First (SJF) policy, which minimizes the average waiting time by executing the process that will finish the quickest (predicted), but it requires you to have the ability to make predictions, and it might starve some of the processes. Round Robin scheduling assigns each process a small time slice or quantum, and processes are served in a rotating order, preserving the doctrine of fairness and responsiveness while preventing a single process from dominating the CPU, although it does incur context switching overhead. A priority-based scheduling scheme marks every process with a priority level and only executes those with the highest priority before others, with preemptive policies (suspending

tasks with lower priority) or non-preemptive methods (waiting for task completion or voluntary release). It achieves this through the use of meta data, and real-time file systems use specific algorithms like Rate Monotonic Scheduling (RMS) or Earliest Deadline First (EDF) to provide timing guarantees for real-time applications like medical devices, automotive systems, or industrial control systems. With the advent of modern multi-core processors, this has changed again; scheduling becomes much more complex, as it must be able to deal with the affinities between the cores, the cache coherence, and their ability to be executed in parallel.

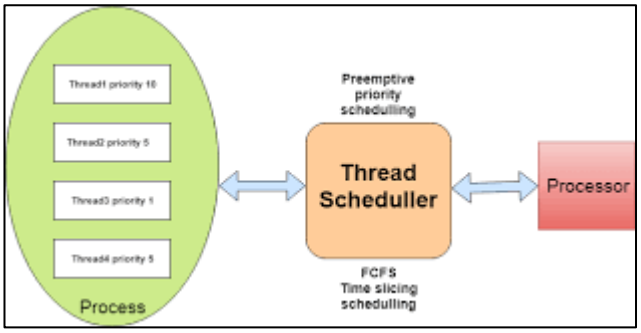


Figure 1.3.1: Thread Scheduling

Thread scheduling (finer-grained) handles scheduling at the level of threads, allowing multiple threads to execute in the same process concurrently, sharing the same memory space and resource. The sophisticated scheduling methods include multilevel feedback queues that increase or decrease the priority of processes based on their execution history; affinity-aware scheduling that keeps workloads on the same processors to take advantage of cache reuse; and heterogeneous computing scheduling that allocates concurrent workloads to specialized processing units such as graphics processing units (GPUs), field programmable gate arrays (FPGAs) or artificial intelligence (AI) accelerators. Load balancing algorithms balance the amount of computational work taking place at any given time, by distributing the processes that execute across multiple systems or processors, to maximize throughput & minimize response times. There has been a growing need for energy-aware scheduling in mobile devices as well as in data centers to perform intelligent trade-offs between performance and power consumption through mechanisms like dynamic voltage and frequency scaling (DVFS), core parking, workload consolidation, etc. Practical implementations of scheduling must also deal with cases of priority inversion (a high-priority process

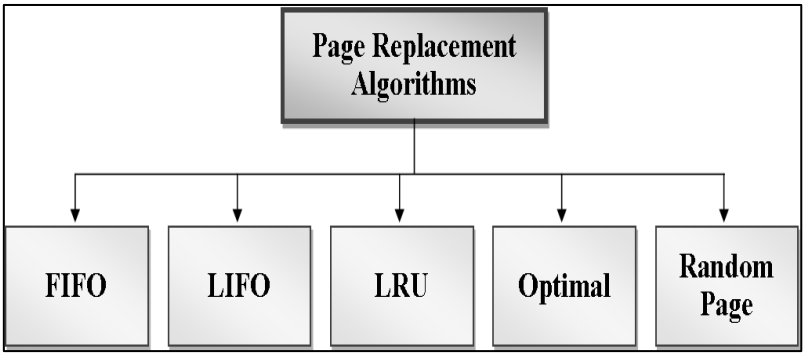
is waiting on (a resource controlled by) a low-priority process), which can be managed with protocols such as priority inheritance or priority ceiling. Processor management also includes interrupt handling—the method used by external events to notify the CPU that it should temporarily stop normal execution in order to address time-critical tasks, such as state changes in hardware, the completion of an I/O operation or error conditions. The enhanced sophistication of modern processor management systems is a direct result of these challenges, as a diverse set of workloads with widely differing requirements from background batch processing to interactive user applications to time-critical control systems must be able to be run and execute efficiently in the same environment on common hardware resources.

1.3.3 Memory Management Systems

Memory management represents an essential component of computational system functionality, involving the various methods and processes operating systems utilize to oversee, distribute, and structure the primary memory resources of a computer. Tiers of the memory hierarchy: registers, cache memory, main memory (RAM), and virtual memory (in secondary storage devices). To effectively do this, it needs to address some basic issues: how to allocate memory to processes when they need it, how to free the memory when it no longer needs it, track usage to ensure performance, prevent unauthorized access and maintain memory coherency in a multi-processor system. Virtual memory is at the heart of modern memory management an abstraction that gives to every process the illusion of owning its own large, contiguous space of addresses without regard for available memory or competing processes. The logical representation of memory allows developers to get a finer-grained view of hardware, flexibility in memory allocation, access protection between processes, and the ability to run programs independent of physical memory (e.g. if an executing program size exceeds physical memory). Virtual and Physical Addresses the translation between virtual and physical addresses is done through a mix of hardware and software mechanisms, where address translation is performed through memory management units (MMUs), with the help of operating system-maintained page tables. The most common implementation of virtual memory is called paging, where both physical and virtual memory are

divided into fixed-sized units called blocks or pages (usually between four and sixty-four kilobytes in modern systems). Access to a virtual address by a process would translate to a corresponding physical location using a page table that stores a mapping between a virtual page number to a physical frame number. When the page requested isn't present in physical memory (page fault), the operating system suspends the process, retrieves the page from the secondary storage (women's clothing warehouse or hard disk), updates the page table, and resumes execution, but these details are invisible to the application; even so, it's key to extending memory.

Page replacement algorithms decide which pages to evict when physical memory is full, with popular approaches being Least Recently



Used (LRU), which evicts pages that have not been accessed for the longest time; First-In-First-Out (FIFO), which evicts pages in the order

Figure 1.3.4: Page Replacement Algorithms

they were loaded; and Clock algorithm, which approximates LRU without the high overhead by maintaining a circular list of pages with reference bits. Its advanced forms are approximations (e.g., CLOCK) and improvements (e.g., CLOCK-Pro) which both try to get the simplicity from FIFO but the performance from more advanced caching algorithms. Paging divides physical memory into fixed-size units, typically 4 or 8 KB pages, and maps logical address space pages to physical pages without considering the program structure. Segmentation is an alternative or complementary method of memory management; it organizes memory according to the logical structure of programs (procedures, data structures, etc.) rather than fixed-size space. It allows more granular protection and sharing mechanisms as each segment is a logical unit with attributes such as read only or executable. Most modern systems use a combination of segmentation to provide logical organization and paging for physical memory.

Processes manage their own memory allocation in ways that can range from basic contiguous allocation to complex dynamic memory management. A part of memory known as the heap is devoted to runtime allocation, and algorithms are required to process requests for variable shaped allocations without causing fragmentation. These strategies include first-fit (using the first sufficiently large free block), best-fit (choosing the smallest block that still satisfies the request), and buddy system allocation (dividing memory into contiguous power-of-two sized blocks to simplify coalescing of free space). For instance, advanced memory managers use either segregated fits (two or more free lists for various classes of sizes, which keeps the fragmentation of memory in check), or generational garbage collection in managed-language settings, which keeps track of the time-to-die, since the majority of objects exist for a short time only. Memory protection mechanisms use hardware features such as protection bits in page tables and memory protection keys to prevent processes from accessing or modifying memory allocated to other processes, or the operating system. Modern systems also use Address Space Layout Randomization (ASLR) to protect against security vulnerabilities by randomizing the locations in memory of program components. (Cache management is almost always implemented in hardware, but it has a lot to do with what the operating system does regarding memory policies.) By influencing the manner in which virtual pages map to cache lines, techniques such as cache coloring aim to improve cache usage. Memory compression is a new paradigm for adding effective memory capacity by compressing infrequently touched pages instead of writing them to disk and later reading them back from disk to reduce latencies for future accesses. Multi-processor systems with Non-Uniform Memory Access (NUMA) architectures add further complexity by making memory access times dependent on the processor's proximity to the memory location, thus necessitating the use of NUMA-aware allocation policies. To address this fragmentation, heterogeneous memory management systems that take advantage of the characteristics of different types of memory have been implemented, such as placing data intelligently into different memory regions as a function of access frequency and performance needs. Despite these advances, effective memory management is still critical to system performance and stability, with contemporary operating

systems continuing to develop increasingly sophisticated memory management techniques, managing the conflicting requirements of capacity, performance, protection and power efficiency across a complex and ever evolving hardware architecture.

1.3.4 Storage Systems and File Management

Storage systems are crucial for saving data after the computer is turned off, transforming it into persistent data, whereas the file management framework allows you to save that persistent data within the storage system. The storage hierarchy ranges from fast, costly, and low-capacity storage technologies (like solid-state drives (SSDs)) to slower, cheaper, and larger-capacity technologies (such as hard disk drives (HDDs), optical drives, and tape drives), each offering distinct trade-offs in terms of performance, cost, and longevity. We rely on file systems to offer this fundamental abstraction layer that turns raw storage trafficking capabilities into structured, well-defined hierarchies of organized blocks that users and applications can easily traverse and maneuver in. At the hardware layer, storage devices use different principles; HDDs rely on magnetic recording stored on spinning platters and accessed by mechanical read/write heads so performance is dependent on rotational latency, seek time, and transfer rates; SSDs with no moving parts leverage flash memory cells laid out in pages and blocks with rapid random access times, though introduce complications like write amplification, wear leveling, and garbage collection; emerging technologies such as 3D XPoint (Intel Optane) bridge the gap between memory and storage with their own performance metrics. Storage device drivers and I/O subsystems in the operating system interact with the devices, abstracting hardware-specific details and providing a standardized interface for higher-level components. RAID (Redundant Array of Independent Disks) configurations that use multiple physical drives as a single logical unit (potentially for performance, capacity, and/or redundancy via striping (RAID 0), mirroring (RAID 1), or parity-based redundancy (RAID 5, RAID 10), are common in modern storage architectures. Block-level storage virtualization abstracts physical devices in blocks, presenting logical volumes that span multiple physical devices and process thin provisioning, snapshots and replication, while storage area networks (SANs) and network-attached storage (NAS) extend those capabilities

over a networked space. File systems abstract these block-level capabilities into the hierarchical realm of files and directories, which serve as the main interface for organizing and accessing the data. And different file systems, based on the type used by the computer, could take different approaches to primary problems like space allocation, metadata, directory structure and crash recovery (for example FAT File Allocation Table, ext4 Fourth Extended File System, NTFS New Technology File System, and HFS+ — Hierarchical File System Plus). You will need to retain critical metadata about the files, such as names, timestamps, ownership, permissions, and the mapping of logical file structures to physical storage locations. Modern file systems have developed such sophisticated capabilities to meet new needs: logging file systems such as ext4, XFS, and NTFS write metadata about data changes in addition to the data itself, ensuring consistent operations during catastrophic failures; copy-on-write file systems such as ZFS and Btrfs never overwrite existing data locations, they write updates to new locations and atomically update pointers in metadata, allowing features like snapshots and providing protection from corruption after unexpected power failures; log-structured file systems such as F2FS map random writes to sequential writes to maximize performance for SSD and other flash-based file systems, thereby improving write performance and reducing write amplification. Obtaining knowledge and status: File management activities encompass the file's entire lifecycle, including creation, naming, access control, modification, backup, and finally deletion or archiving. Each file can vary widely in characteristics: Executable binaries require specific formats and alignment; Databases often use their own internal storage structures optimized for access patterns; Multimedia files utilize various compression algorithms; and text files require character encoding support. Different ways of access provide optimization opportunities as well as challenges (sequential processing versus random access patterns). Virtual File Systems (VFS) play an essential role in modern operating systems by exposing a uniform interface to applications while supporting a wide variety of underlying file system implementations alike the network file systems and the local ones having extremely diverse internal structures. When the same file is accessed frequently, file caching improves performance by keeping data and metadata in memory, using sophisticated algorithms that

attempt to hold on to data that is useful without exceeding available memory. Modern file systems support advanced features such as encryption which secures sensitive data even if the physical storage is compromised, deduplication which minimizes the storage of identical data blocks, the ability to compress data to save space, and specification of quotas to restrict the amount of resources consumed by users or groups thereof. With the arrival of cloud storage, whole new paradigms for files arose, with object storage systems like Amazon S3, Google Cloud Storage, and Azure Blob Storage employing flat namespaces of objects and (meta)data associated with them instead of hierarchical file structures, along geared for scale, durability and while being easily accessible over distributed environments. These systems include well-known distributed file systems (e.g., Google File System (GFS), Hadoop Distributed File System (HDFS), and Ceph), which take the concepts of traditional file systems and apply them over clusters of machines, using replication, fault tolerance, and parallel access mechanisms to achieve scalability and performance impractical with single-system approaches. Emerging storage technologies further obfuscate traditional categories: persistent memory provides byte-addressable access with durability; storage-class memory delivers near-DRAM performance with non-volatility; and computational storage moves processing closer to data to mitigate data movement and improve efficiency for select workloads. As workloads change, as hardware capabilities and reliability requirements shift, so must file and storage management systems, and researchers are focusing efforts in areas such as improved performance for emerging non-volatile memory technologies, secure transparent encryption for enhanced security, low power consumption for massive storage arrays, and self-healing mechanisms to maintain data integrity when hardware fails or is victimized by a cyber-attack.

1.3.5 Input/Output Systems and Device Management

Input/output (I/O) systems and Strategies for effective Device management antigen are the vital intermediary link between computing systems and the outside environment, including the hardware components, software subsystems and operational protocols that allow computers to interact with peripheral devices, sensors, networks and storage systems. I/O devices are the most diverse class of peripherals and can be as simple as human interface peripherals such as keyboards

and mice or as complex as communication equipment, graphics processors, and special purpose controllers used in industrial control; thus providing standard interfaces while attempting to provide some range of performance characteristics, communication mechanisms, and functionality can prove to be quite a challenge for the system designer. In the context of hardware-level I/O communication architectures, these authors describe four common modes: programmed I/O where the CPU explicitly instructs devices to transfer blocks of data, interrupt-driven I/O where devices can interrupt in the event of needing the processor's attention, freeing up the CPU; DMA (Direct Memory Access) which leads to devices pulling or pushing data from memory without requiring the CPU to watch over; and channel I/O commonly used in mainframe systems, where entire I/O programs can be sent to special processors to be executed without the requirement of the attention of the main CPU. Wired connections for devices and computer systems have transitioned from parallel buses like ISA and PCI to serial connections such as USB, PCIe, and Thunderbolt delivering higher speeds, fewer pins, and even the ability to hot-plug the devices. Such connections are made via controllers, hardware that can translate between the internal signals of a computer and the specialized protocols of the devices, usually with some form of buffers to account for differences in timing between CPUs and slower devices. Through a layered system of device drivers, software components that allow devices to communicate with the operating system and abstract away device-specific implementation, this hardware is managed by the operating system. There are driver frameworks built into modern operating systems that describe application development patterns that third parties can use to implement drivers that will be compatible without needing to learn about the internal architecture of the system. Such frameworks generally provide interrupt handling, memory management, power management, error recovery, and other low-level services so that driver developers can concentrate on device-specific details. Purpose: ACPI, UEFI, PnP Device discovery and configuration mechanisms that help in automatic detection, configuration and allocation of resources for devices without any manual intervention. From the application perspective, OSes expose devices through abstraction layers that make interaction simple: character devices (like keyboards and serial ports) transfer data byte by byte in streams; block

devices (like disk drives) transfer fixed size blocks of data; and network devices with their own interfaces for packet-based communication. Even higher-level abstractions reduce development complexity file system interfaces for storage devices, graphical frameworks for display devices, and audio subsystems for sound equipment expose active APIs that abstract applications from hardware specifics. Particularly for devices such as disk drives, where the physical characteristics are a major factor in their performance, I/O scheduling is an important component of device management. Reordering requests based on physical location using elevator algorithms (SCAN) and its derivatives helps to keep mechanical movement to a minimum; anticipatory scheduling has been shown to be useful in predicting future requests based on previous patterns; and completely fair queuing ensures that bandwidth is allocated fairly across processes. Most recent systems employ a deadline-based mechanism that optimizes throughput while providing service guarantees at a predefined level to time-sensitive operations. Buffering and caching layers exist all throughout the I/O stack to handle timing discrepancies between the various components that all operate at different speeds: device controllers have hardware buffers; OSes have buffer caches for block devices and network stacks; and applications have their own buffering schemes. By using double-buffering approaches, you can read and write to different buffers, such that one can be used to write data while another is rendered or transmitted, making them suitable for streaming like video playing or audio recording. With the advent of virtual machines, containerized applications, and other forms of system virtualization, the need for a different approach for virtualized devices was created, including device emulation (where hardware behavior is simulated through software), par virtualization (modified drivers in guest systems interact with the hypervisor) and direct device assignment (allowing virtual machines to have direct, exclusive access to physical devices). You are familiar with virtualized environments, where one physical device can be seen by one or more guests; SR-IOV (Single Root I/O Virtualization) allows a single physical device to advertise up to n virtual devices with dedicated resources. However, the increasing adoption of mobile and energy efficient systems has led to the advent of device power management. Examples include selectively powering down unused hardware components, dynamically adjusting its performance to match

current needs, and aligning device states with global power management policies. USB Power Delivery is one example of a standards-based specification that allows for intelligent negotiation of power requirements between devices and hosts. Domain-specific I/O subsystems are designed for their specific type of needs, e.g. cameras and graphics use APIs like DirectX, Vulkan, and Metal with increasingly complex rendering pipelines, audio subsystems mix, convert formats, and align playback across many channels, and human interface device frameworks (HIDs) manage arbitrary input from many sources with accessibility and internationalization considerations. The two main differentiating features of true real-time I/O are deterministic response time which is critical in industrial control systems, medical devices, and automotive systems. As such, real-time systems use dedicated I/O stacks with bounded latency guarantees, priority-based IRQ servicing, and very little jitter. As IoT devices suitable for various purposes can be very light-weight, I/O management has always been crucial for those devices with bandwidth constrained protocols, energy-efficient communication patterns, etc. Edge computing architectures allow the processing of data at or near the source, decreasing latency and bandwidth consumption but also introducing new challenges for device management across distributed environments. Security touches every part of modern I/O systems: device attestation ensures the hardware is what we expect; secure boot verifies device firmware; access control restricts which processes may interact with sensitive devices; and encryption protects data in flight. As computation expands into new spheres, I/O systems evolve further, with technologies such as neuromorphic interfaces directly wired into biology; quantum I/O enveloping the extreme environmental needs of quantum processors; and brain-computer interfaces (BCI) transforming neural activity into computational input, all presenting new levels of difficulty for device management systems on which I/O profiled devices depend.

1.3.6 Network Operations and Distributed Systems

Network operations and distributed systems are the fundamental threads that connect the fabric of computing today, facilitating communication, resource sharing, and collaborative processing across components that are geographically or physically separated, whether they be local clusters or global-scale infrastructure that spans continents. A layered architectural view is essentially the foundation of

computer networking and the most common manifestation of that is the TCP/IP model: A link layer, which provides a similar physical connection and media access to the alternate layer, a layer responsible for addressing and routing between networks (Internet Layer), a transport layer for reliable delivery and flow control of data, and finally activated by a layer for user applications and network services (Application Layer). Disparate physical characteristics of the transmission mediums make a big difference in terms of bandwidth, propagation delays, fault tolerance, etc. For these physical media, we need some data encoding techniques to take our digital information and convert it into signals suitable for those types of media, such as using a scheme to enable the appropriate type of encoding like Manchester, PAM-4 and QAM modulation, maximizing the density of the data and minimizing errors. Media access control (MAC) mechanisms organize when shared channels can be used, from deterministic techniques like time-division multiplexing, to contention-based schemes such as CSMA/CD (Carrier Sense Multiple Access with Collision Detection) in classical Ethernet or CSMA/CA (Collision Avoidance) in wireless. Network addressing schemes form the backbone on the way to the identification and location of devices: MAC addresses uniquely identify physical network interfaces on the link layer; IP addresses (both IPv4 and growing IPv6) allow global routing in the internet layer; and finally, domain names create human-readable identifiers (resolved into IP addresses by the Domain Name System or DNS). Routing works as a process of establishing routes and data transmission across the interconnected networks through paths that are determined by using a set of algorithms that strike a balance between the distances, reliability, heaps and administrative policies. Routing protocols such as RIP (Routing Information Protocol), OSPF (Open Shortest Path First), BGP (Border Gateway Protocol) employ distinct methodologies for exploring and administering routes; interior gateway protocols concentrate on routing within organizations, while exterior gateway protocols control routing across the internet between diverse autonomous systems. The transport layer provides essential functionalities such as connection management, reliable delivery, flow control, and congestion avoidance. The Transmission Control Protocol (TCP) uses connection-oriented transmission with reliable, ordered service with mechanisms for acknowledgments of received data,

retransmissions of lost packets, and dynamic adjustments of transmission rates to conditions on the network. UDP (User Datagram Protocol) is a connectionless protocol that provides communication without the overhead of establishing a connection and is often used when low latency is more important than reliability, such as for real-time streaming and DNS lookups. Newer protocols like QUIC merge elements from both strategies, offering reliability and security from the application layer above UDP layers to minimize connection creation latency and optimize performance across difficult link conditions. Network security involves many specialized processes: encryption preserves the confidentiality of information via protocols such as TLS (Transport Layer Security); authentication validates communicating endpoints through certificates, pre-shared keys, or multi-factor systems; access control mechanisms like firewalls and segmentation ensure communication routes are restricted according to rulesets and policies; intrusion detection/prevention systems inspect traffic patterns for the presence of malevolent behaviors. DDoS protection uses traffic analysis, rate limiting and traffic spreading to keep service available in the face of an attack. Quality of Service (QoS) involves mechanisms that would allow traffic to be prioritized based on type, source, or requirements of the application, and it works by implementing techniques like packet classification, queue management, traffic shaping, and reservation of resources to make sure that critical communications are properly treated even when the network is congested. The SDN is a new networking architecture that separates the control and data planes, allowing for centralized control, programmability, and more efficient resource allocation (most commonly used with the Open Flow Protocol). Network virtualization, then, takes these concepts and applies them at the network level, allowing logical network abstractions of sufficient complexity to exist independently of the physical infrastructure beneath them, facilitating multiple isolated networks to share hardware concurrently. Different flavors of this are radiating out in the form of virtual LANs (VLANs), Virtual extensible LANs (VXLANs), and Network Function Virtualization (NFV) which virtualizes hardware appliances (e.g., a router) and firewalls and load balancers that are implemented as virtualized software rather than hardware. Writing detailed networking code does not usually lead to success; instead, distributed systems are

built on top of these foundations for networking to create coherent programming environments over multiple physical machines, using middleware, protocols, and architectural structures to overcome the inherent problems in distributed computing: heterogeneity of components, open-ness to extension, security across trust boundaries, scalability to increasing demand, failure handling, concurrency enabling, and transparency that hides distribution from the programmer Complexity of users and applications inside. There are varying designs of distributed system architectures; client-server architecture separates service providers from consumers, peer-to-peer distributes services among participating nodes, hybrid architectures like edge computing position processing at the edge of the network close to the sources of data, and cloud computing offers resources that are virtualized and accessible through standard interfaces. Distributed systems communicate in multiple ways: remote procedure calls (RPCs) with their object-oriented variant allow remote procedures to be invoked as if they were local; message-oriented middleware's implement queuing, routing and transformation services to enable asynchronous communication; publish-subscribe systems allow for many-to-many communication with loose coupling between participants; streaming platforms process continuous data flows across distributed components. Consistency models govern what to expect around visibility and order of data across other distributed components; they stretch from strong consistency (all nodes see the same thing at the same time) through eventual consistency (the data will converge with time but does not require synchronization in the moment). The CAP theorem presents absolute trade-offs for distributed systems by saying that they can only offer two of three guarantees: consistency (all nodes see the same data), availability (the system responds to requests) and partition tolerance (the system still operates even when networks do not). Modern distributed databases all implement different consistency models depending on applications requirements: classical relational databases tend to enforce ACID properties (Atomicity, Consistency, Isolation, Durability) through two-phase commit protocols and distributed transactions; NoSQL systems often embrace BASE properties (Basically Available, Soft state, Eventually consistent) for improving partition tolerance and scalability; and NewSQL approaches try to combine ACID guarantees with horizontal scalability. Distributed

coordination services such as Apache ZooKeeper, etcd, Consul, etc provide primitives for leader election, configuration management, service discovery, distributed locking that make it easy to build reliable distributed applications. Container orchestration platforms (like Kubernetes) automate the deployment, scaling, and management of containerized applications across a cluster of servers with advanced scheduling, load balancing, service discovery, and self-healing capabilities. Distributed file systems and object stores, such as Hadoop HDFS, Ceph, Amazon S3, and Google Cloud Storage, offer storage services that span machine boundaries with replication, fault tolerance, and scalability. Block chain technologies are a specialized subclass of distributed system that enables decentralized consensus protocols to have consistent state without a trusted central authority, leading to applications from crypto currency to supply chain tracking to digital identity management. Fundamental challenges in connecting and coordinating computational resources across physical, organizational, and trust boundaries continue to be tackled by evolving practices of network operations and distributed systems underlying the operation paradigms that have emerged; server less computing abstracts away (even managing) the infrastructure; 5G and beyond wireless technologies enable new classes of distributed applications; zero-trust security eliminates implicit trust due to network location; edge computing pushes the processing closer to the data source.

1.3.5 Security, Performance Optimization, and System Reliability

Security, performance optimization, and system reliability are critical dimensions of computer system operations they define how well systems protect sensitive assets, provide timely service, and continue to operate consistently under stressful conditions. These three form a triad of operational concerns that intersect in many complex ways security practices can impact performance, performance improvements can add reliability compromises, and reliability mechanisms can influence both security posture and performance efficiency. Systematic approaches that balance competing priorities, such as institutional, macroeconomic, sectoral, and organizational factors, to robustly implement practices across all three dimensions in specific operational settings. Computer security involves the securing of hardware, software, data, and communications of system assets from unintended

access, use, disclosure, disruption, modification, or destruction. Fundamentally, security enforces the CIA triad; Confidentiality ensures that no one accesses sensitive information; integrity preserves information integrity from intentional or accidental tampering; and availability ensures authorized users can reach their resources when needed. These objectives are accomplished through various defensive mechanisms targeting distinct facets of the security dilemma: cryptographic schemes that safeguard data utilizing encryption ciphers, such as AES, RSA, and elliptic curve cryptography, alongside hashing algorithms like SHA-256 that ensure data integrity(ies); authentication systems that validate claims of identity through knowledge factors (passwords, security questions), possession factors (hardware tokens, mobile devices), and inherence factors (biometrics like fingerprints, facial recognition); authorization schemes that delineate action permissions for authenticated subjects through models such as discretionary access control (DAC), mandatory access control (MAC), role-based access control (RBAC), and attribute-based access control (ABAC); secure communication protocols like TLS/SSL that establish encrypted channels impervious to interception and modification; and network security mechanisms encompassing firewalls, intrusion detection/prevention units, and VPNs that confine communication pathways to legitimate channels. Vulnerability management processes are designed to identify, assess, and remediate security weaknesses in a software application, often through activities such as static and dynamic code analysis, penetration testing, and regular patching. Security monitoring and incident response abilities that recognize and respond to security occurrences (through log analysis, behavior monitoring, and established threat-handling procedures). Zero-trust architecture and similar approaches move away from perimeter-based security models and instead analyze each access request, wherever it originates from or however attached to a network, to verify that it's still valid. Optimizations improve overall performance as measured by multiple metrics: throughput (amount of work done per unit time), latency (time taken to finish given operations), resource utilization (helping make use of relevant points of computing resources), energy efficiency (amount of work accomplished per amount of energy consumed). Optimization exists at all levels of the system from hardware choices and configurations balancing compute resources

with workload needs, to processor optimizations (instruction pipelines, branch predictors, speculative execution, and synchronization of work across cores), memory hierarchy tuning (including cache sizes, memory alignment, perfecting and NUMA awareness), I/O (effective buffering, asynchronous I/O, and device selection), and networking performance (protocol, buffering and topology). Software-level improvements involve the implementation of more efficient algorithms to minimize computational complexity, optimizing compilers for specific target architectures to high-quality machine code, improvements on databases with indexing, query rewriting, and execution plan selection, and application-specific improvements that focus on the hot paths in the code graph. Load Balancing and Capacity Planning Techniques; Load balancing techniques distribute work across multiple resources to prevent bottlenecks, while capacity planning processes ensure sufficient resources for anticipated demands. Performance monitoring and analysis tools help improve data-driven optimizations by performing profiling, tracing, and benchmarking for locating a performance bottleneck and validating the impact of improvements. The system is reliable if, under typical operation, it ensures consistent, correct operation regardless of the failures, flaws, or environmental stresses that might occur in components. At a high level, reliability engineering encompasses a few broad elements: defect prevention avoids the introduction of defects through strict design practices, formal verification, and quality methods; fault tolerance enables continued operation in the face of component failures with redundancy (keeping duplicate components to take over when primary components fail), diversity (multiple different implementations of the same solution to avoid common failure modes), isolation (restricting the failure propagation to a limited scope), and graceful degradation (maintaining the best possible level of service during partial failures); fault detection finds problems through health monitors, watchdog timers, checksums or error detection codes; and fault recovery restores normal operations post-failure with techniques like rolling back to known-good states, failing over to backups, and self-healing where certain failure classes are automatically repaired. Reliability metrics measure how dependable a system is: Mean Time Between Failures (MTBF) indicates the average amount of operational time between one failure and the next; Mean Time To Repair (MTTR) measures the

average time taken to return a system to an operational state after a failure; availability communicates the percent amount of time a system is functioning; and durability represents the percentage chance that data will remain intact over a certain period. High-strength structures use active-passive or active-active setups between dispersed geographical areas to keep the service running, regardless of localized failures or such disasters. Chaos engineering tests reliability proactively by sneaking in controlled failures into production systems and checking to see if recovery mechanisms kick in, as expected. Approaches to security, performance, and reliability are traditionally developed in isolation, as though the three are independent; this plain non-sense. Performance optimizations that avoid safety checks or reliability mechanisms that leak diagnostic information are common sources of security vulnerabilities. Security controls which add more processing steps or reliability features that keep redundant state can cause performance bottlenecks. Security mechanisms that raise the complexity of the system or performance optimizations that narrow the tolerable fault margins may lead to reliability challenges. In practice, systems must be oriented to meet varying criteria across these dimensions depending upon use case requirements—system must balance security and reliability against raw performance (i.e. mission critical systems will typically favor non-performant options over less reliable systems); systems must maintain performance guarantees whilst ensuring adequate security and reliability (e.g. real time systems); or, systems must maintain optimal performance while maintaining adequate security and reliability (e.g. consumer applications). This evolution of computer systems continues to fundamentally alter the operations landscape(s): Cloud introduces shared responsibility models for security, performance, and reliability, where some level of responsibility is managed by a service provider with others maintained by the customer; containerization and micro services architectures divide these concerns into smaller, more manageable modules; DevSecOps incorporates security into the development lifecycle (i.e. not an afterthought); site reliability engineering (SRE) applies software engineering paradigms to operationalize problems; and artificial intelligence increasingly encroaches to help humans identify and profile security threats, performance parameters, and reliability components that pose a risk to

service and product offerings. As systems grow more complex and interconnected, the strategic orchestration of the management of security, performance, and reliability operations is becoming a core function for delivering systems to support the increasing expectations of both organizations and individuals in our digital society.

1.3.6 Emerging Trends and Future Directions in Computer Systems

The operations of computer systems are evolving at an unprecedented pace, highlighting the importance of writing semantics in the continuous integration and deployment process. At the same time, several disruptive trends are reformulating the very fabric of computer systems, heralding a new realm of capabilities and new operation challenges that will characterize the next generation of computing infrastructure. Quantum computing is perhaps the most disruptive change on the horizon, as it computes fundamentally differently than classical computation, by utilizing quantum mechanical effects like superposition and entanglement. In contrast to conventional bits with distinguished states of 0 or 1, quantum bits or "qubits" can be in superposition with multiple possible states at once, potentially allowing for exponential parallelism on certain problems. Go to any specialized quantum system from companies like IBM or Google or D-Wave or other new starts, and you can find ways in which these experimental systems demonstrate capabilities in areas such as cryptography, optimization, simulation of quantum systems, and even some machine learning functions. Modules. Operationally, quantum computing has enormous implications: quantum algorithms need completely novel ways of programming; quantum decoherence makes error correction exponentially harder; dedicated environments with extremely low temperature requirements lead to new types of infrastructure problems; hybrid architectures with classical and quantum processors need new interface paradigms. Although general-purpose quantum computers are many years away from practical use, the security consequences are already causing changes to quantum-resistant cryptographic algorithms that would be safe against future quantum systems. Neuromorphic computing mimics biological neural systems using hardware architectures that more closely mirror the structures in the brain, in contrast to traditional von Neumann architectures. Such systems use massively parallel processing elements that integrate memory and

computation and provide large performance improvements for pattern reorganization, Sens.

1.3.7 Types of Operating Systems: Batch Processing, Multi-Programming, Time Sharing

An operating system (OS) is a crucial software layer that acts as a bridge between computer hardware and its users, on top of which users can conveniently and efficiently run programs. Operating systems have come a long way since the birth of electronic computers, continuously adapting to new hardware capabilities and user needs. It represents a basic shift from primitive, single-function applications to advanced, multi-feature settings that can run several simultaneous processes in a resource-efficient manner. The earliest computers didn't contain anything that resembled an operating system, as we understand the term today; they required programmers to talk directly to the machine hardware through physical switches and lights. This hands-on approach proved insufficient with the increasing complexity behind our computing hardware and the increasing expectations users had of their applications. The ever-increasing sophistication of operating system designs was driven by the need for more efficient resource utilization and improved user experience. This Unit provides insight into three base operating system paradigms that approximate significant evolutionary stages in computer history: batch processing systems, multiprogramming systems, and time-sharing systems. These systems all addressed the shortcomings of their predecessors and provided new abstractions that still shape modern operating systems today. Understanding these fundamental operating system types help us appreciate the principles that underlie many of the modern computing environments we use today and the historical context that drove their evolution. Batch processing, multiprogramming, and time-sharing represent not just an evolution in technology but also a shift in computing priorities; in other words, from maximizing the usage of hardware to maximizing the matched interactivity of the system. In this article, we will delve into each of these types of operating systems individually, highlighting what defines each one, their main architectural components, advantages, disadvantages, and historical significance to give you an all-encompassing perspective

on how the evolution of operating system design has catered towards the complex needs of computing in modern times.

1.3.8 Batch Processing Operating Systems

Batch processing represents the earliest systematic approach to operating system design, emerging in the 1950s and early 1960s as a response to the limitations of manual program loading. In a batch processing operating system, similar jobs are grouped together into "batches" and executed sequentially without user interaction during processing. This revolutionary approach addressed significant inefficiencies in early computing environments, where computer operators had to manually load and unload programs and data, resulting in considerable idle time for expensive hardware resources. The fundamental architecture of a batch processing system consists of several key components. First, the job scheduler maintains a queue of submitted jobs, determining their execution order based on predefined criteria such as priority or resource requirements. Second, the batch monitor supervises job execution, loading the appropriate program into memory, allocating necessary resources, and collecting output for later retrieval. Third, job control language (JCL) provides a standardized mechanism for users to specify job requirements and execution parameters. The operational workflow typically begins with users submitting programs and associated data (often on punch cards or magnetic tape) to computer operators. These jobs are then grouped by operators into batches with similar resource requirements. The batched jobs are loaded onto input devices, and the batch processing system automatically executes them in sequence, producing output that is subsequently distributed to the appropriate users. This approach offered several significant advantages over manual program loading. Primarily, it improved throughput by reducing transition time between jobs and eliminating the need for human intervention during execution. It also enhanced resource utilization by keeping expensive computing hardware operational for longer periods. Additionally, batch systems introduced the concept of accounting and resource allocation, enabling organizations to track and manage computing resources more effectively. Despite these benefits, batch processing systems suffered from notable limitations. The lack of interaction during program execution meant that debugging was cumbersome, often requiring multiple submission-execution cycles to identify and correct errors.

Furthermore, turnaround time the interval between job submission and result delivery could be substantial, ranging from hours to days depending on system load and job priority. These systems also typically operated with a "first-in, first-out" (FIFO) scheduling approach or simple priority schemes, which could lead to inefficient resource allocation. Historical examples of influential batch processing systems include the IBM 7094 with its Fortran Monitor System (FMS) and the IBM System/360 running OS/360. These systems demonstrated the viability of automated job processing and established fundamental concepts in operating system design, including job scheduling, resource allocation, and system monitoring. Although pure batch processing systems are rarely used in contemporary computing environments, their core principles continue to influence modern computing, particularly in high-performance computing centers, scientific computing

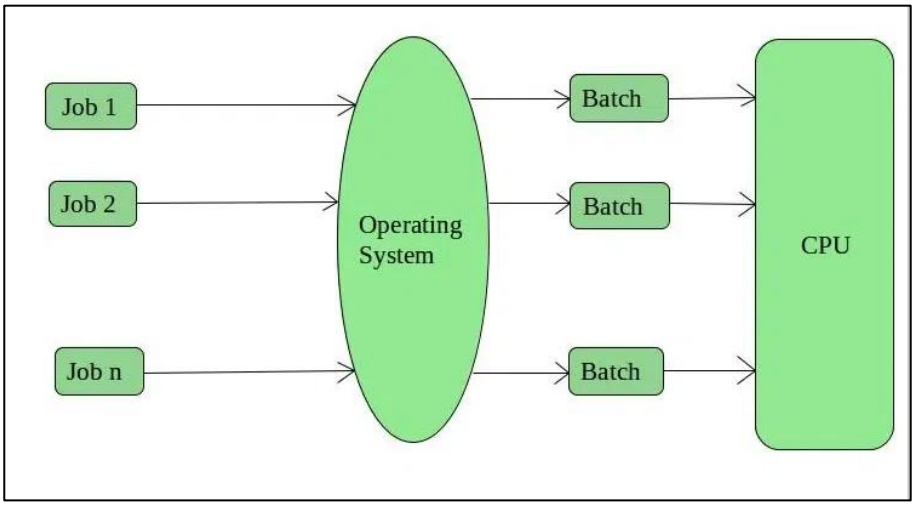


Figure1.3.5: Batch Operating System
[Source - https://www.geeksforgeeks.org/]

applications, and financial processing systems where large volumes of data must be processed without user interaction.

1.3.9 Multiprogramming Operating Systems

Multiprogramming was introduced in 1960s which was a leap over batch processing systems, as it addressed one of the hot topics of CPU underutilization. Batch systems ran jobs one at a time, but multiprogramming brought the radical idea of having multiple programs in memory together at once and transforming numbers between jobs in a process that the OS could switch back and forth among and save CPU cycles lost to I/O. This core adjustment increased system throughput and resource usage dramatically. Multiprogramming systems have many more features in their

architecture than batch systems. Managing memory becomes a lot harder, we need to make sure branches marked with load instructions are protected against being scratched by other programs that are in memory at the same time. Process management systems maintain the state of each loaded program and coordinate transitions between them. In advanced CPU scheduling algorithms meaning which ready process should get processor time depending on factors such as priority, resource needs, and fairness. System level types of operations that allow for I/O requests and refinement. And complex I/O management systems that allow for multiple active programs. From the operating system perspective, when a program initiates an I/O operation, the multiprogramming operating system will do a context switch, saving the current program state and handing control to a different program that is ready to execute. This context switching operation means saving the contents of registers, program counters, and other relevant information about the execution states of the blocked program and loading that of the other program to be executed. When the I/O completes, the first program is re-eligible for execution, enabling the operating system to return control to it at an appropriate time. This was a great improvement over simple batch processing. Most importantly, it greatly enhanced CPU utilization because with this way the processor never visited the idle state if programs were blocked waiting on an I/O operation. More jobs could be completed in the same amount of time, thus increasing system throughput accordingly. It also offered more complex mechanisms for allocating resources such as: memory, peripheral or even processor time among different workloads running in parallel. These developments were in addition to the challenges and limitations using multiprogramming systems. Memory limits became especially real, since you needed enough physical memory to run multiple programs at once. Now, with multiple processes running on the system, there was contention for the various resources that a process could use, such as I/O devices. Fairness, priority, and throughput considerations required more complex scheduling algorithms. They also introduced the possibility of deadlock, where two or more programs each had resources that the others needed, creating a standstill. Notable example of multiprogramming systems are IBM's OS/360 MFT (Multiprogramming with a Fixed number of Tasks) and MVT (Multiprogramming with a Variable number of

Tasks), UNIVAC's EXEC 8, and derivatives of Unix. These systems introduced essential concepts that would become the basis for contemporary operating systems, such as process management, memory protection, and resource allocation. Multiprogramming is still a basic paradigm of modern computing and is built into the core principles of almost every operating system in use today. Multiprogramming laid the groundwork for concurrent computing, which would be further realized in the form of time-sharing systems, a

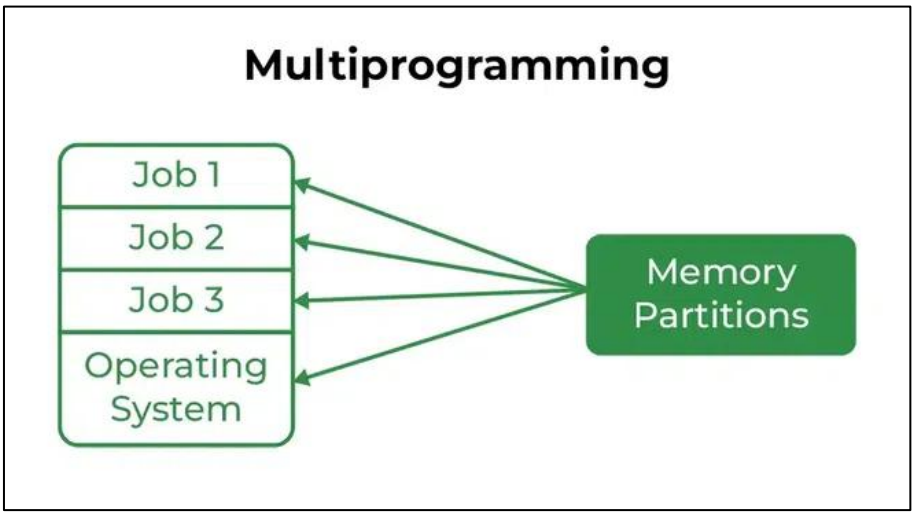


Figure 1.3.62: Multiprogramming Operating System
[Source - https://www.geeksforgeeks.org]

subsequent category of operating system specifically designed to support interactive computing experiences.

1.3.10 Time-Sharing Operating Systems

Time-sharing operating systems, which developed in the mid-1960s, were a significant paradigm shift in computing. They overcame a fundamental constraint of batch and early multiprogramming systems: they did not offer interactive computing capabilities. Multiprogramming only increased the needs of the hardware, and time-sharing systems changed this by creating an illusion of exclusive accessibility of the system by each user. This development fundamentally changed human-computer interactions, allowing people to directly and interactively utilize computers in ways that vastly broadened computing use cases and made computing accessible to many more people. The interactive nature of time-sharing comes from its implementation method—context switch at a high rate between several programs that belong to users. Using time-slicing (usually in milliseconds) this creates the illusion that programs are being executed in parallel (this does not mirror the underlying hardware, which is

inherently sequential). This methodology is distinctly differentiated from multiprogramming through its primary intent as opposed to multiprogramming, which focuses on maximizing CPU utilization by swapping control between programs during I/O tasks, time sharing switches programs based on the time that has been allotted to them versus the waiting on I/O process the architecture of a time-sharing operating system boasts numerous enhancements over multiprogramming executing systems. It needs better CPU scheduling algorithms that balance responsiveness and fairness among many interactive users. In such cases, virtual memory systems become vital, enabling the aggregate memory requirements of all users in active status to be larger than available system memory. Terminal handling subsystems are responsible for interfacing with possibly hundreds of attached user terminals. The file systems of time-sharing environments also utilize concurrency controls to allow multiple users to access shared files at the same time without causing conflicts. The actual time-sharing works with slightly different processes. When a user starts a session, a process is created to represent that user's environment. The system grants short processor time slices to the corresponding process, as the user inputs commands. A process is allowed to run in the CPU until its time slice expires, and if it does not finish its work in the time slice, the process is forcibly suspended and the operating system saves its state and switches to the next one in the ready queue. This is how preemptive multitasking works to prevent a single user from hogging the system. Compared to its predecessors, the time-sharing delivered revolutionary benefits. It pioneered interactive computing, providing a means for users to enter commands and receive immediate feedback. This interactivity made possible new classes of applications, including real-time communication, interactive programming environments and early computer-aided design systems. In addition, time-sharing democratized access to computing resources by enabling multiple users to share expensive hardware simultaneously, making it feasible for many users who could not afford dedicated use. In addition, it allowed many users to work on related tasks in a cooperative manner, sharing both data and resources. Early time-sharing systems did face considerable challenges, however, despite these advantages. Context switches do incur overhead, so if they become too frequent, they could impact overall system performance

and should be avoided at large numbers of active users. These systems needed many megabytes of memory and megahertz worth of processing power to even approach acceptable response times compared to their counterparts. Moreover, the prominence of security concerns added another layer, as the system needed to defend users from unauthorized access to each other's data and processes. Some of the pioneering time-sharing systems include the Compatible Time-Sharing System (CTSS) at MIT, Dartmouth Time Sharing System (DTSS), which introduced the BASIC programming language, and MULTICS (Multiplexed Information and Computing Service), which served as the model for many future operating system designs, especially Unix. In addition to its broad applicability to modern computing, these systems introduced basic ideas such as interactive user interfaces, preemptive multitasking, and user-oriented computing environments. The concept of time-sharing was a profound leap forward in making computers accessible to a wider audience and more useful, paving the way for principles that

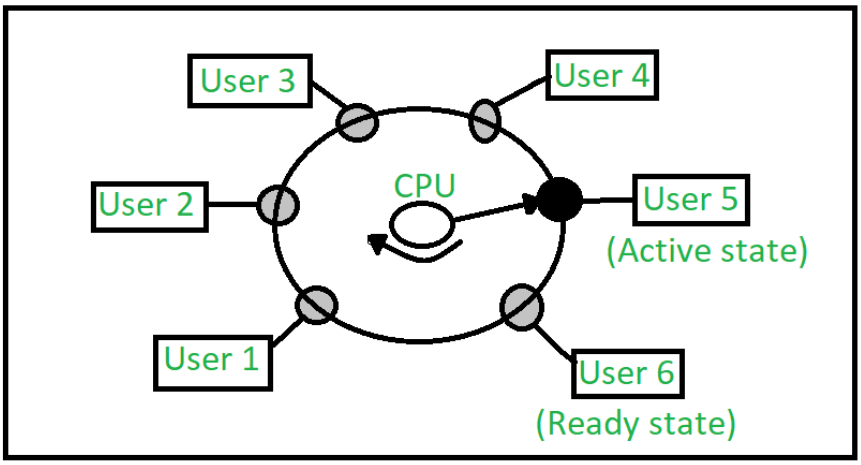


Figure 1.3.7: Time-Sharing Operating System

[Source - https://www.geeksforgeeks.org/]

still undergird modern operating systems and their interaction with users.

1.3.11 Comparative Analysis: Evolution and Trade-offs

This evolution from batch processing to multiprogramming, and then to time-sharing systems, represents a fundamental shift in computing philosophy and capability, as each technique built on its predecessors to overcome their limitations while adding new capabilities and challenges. Plotting out the evolution of operating system design reveals how balancing competing objectives such as hardware

utilization versus system throughput versus response time versus user experience have continued to drive optimization. One dimension to compare these operating system paradigms is resource utilization. Batch processing systems were designed to maximize resource utilization on expensive computing hardware by minimizing idle time between jobs, but at the expense of interactive capabilities. Multiprogramming systems took this one step further by overlapping I/O operations with CPU activity, thereby minimizing idle time for the processor. While time-sharing systems managed to use their resources fairly well on average, it did require some sacrifice of the raw efficiency of the hardware in favor of interactive capabilities, accepting the overhead of switching contexts often to remain responsive. Another important difference between these operating system types is their treatment of users. Batch processing systems created a great separation between users and the computing environment, with operators acting as intermediaries and users typically getting results hours or days after their submission. Multiprogramming may have alleviated this separation to an extent but still required limited direct intervention. Before the arrival of time-sharing systems, this relationship was what I would call sort of a batch processing thing, where there wasn't a lot of interaction on demand because there were two degrees of separation between the human and the resources available. These paradigms also differed considerably in their performance metrics. Batch systems optimized for throughput the number of jobs completed per unit time and of necessity, low overhead processing of batch jobs, preferring high-volume processing over minimizing per-job completion time. Multiprogramming had already improved throughput but added a new metric, device utilization. The emphasis of time-sharing systems moved sharply toward response time the elapsed time from the user request until the system response even when this sometimes had a negative effect on overall throughput. These changing priorities are reflected in the evolution of scheduling algorithms. Batch systems mostly used either basic first-come-first-served or simple priority types. This led to scheduling techniques such as shortest-job-first, priority-based preemptive scheduling, and so forth in multiprogramming in order to maximize throughput and CPU utilization. Time-sharing systems implemented round robin scheduling with preemption and complex priority aging mechanisms. Therefore, memory management

techniques naturally evolved among these paradigms. Batch systems usually handled a single program at a time with primitive memory management. Multiprogramming required memory protection facilities and introduced partitioned allocation strategies. Time-sharing systems introduced virtual memory techniques that allowed programs to run as if they had access to more memory than (actually) existed, and facilitated new strategies to allocate memory in a more flexible way. This evolution persists with modern operating system designs. Modern systems are designed according to aspects of all three paradigms: batch processing (for background tasks and processing of high-volume data) and multiprogramming (to increase resource use efficiency) and time sharing (for interactive users). Familiarity with these historical paradigms exposes the origins of how contemporary operating systems balance competing goals and make explicit tradeoffs necessary to enable a wide range of computing applications. And as batch went to multiprogramming to time-sharing, one didn't the other, but expanded capabilities that enabled the operating system to handle an increasingly wide variety of computing needs onto increasingly complex hardware environments.

1.3.12 Modern Implementations and Legacies

Modern operating systems are characterized by some combination of batch, multiprogramming and time sharing, and many aspects of these historical models have adapted and carry through to their modern counterparts. Rather than discarding such approaches to innovate these paradigms, modern systems seamlessly incorporate these in a single unifying architecture capable of addressing a broad variety of computing demands, from high-throughput processing for background jobs all the way to highly interactive applications for end-users. Aspects of modern operating systems are still reminiscent of batch processing. It's the background processing capabilities, which let resource-hungry work happen in the background at the system level, usually when the system is idle. In larger environments, job scheduling components orchestrate the execution of batch administrative work, data processing jobs, and systems maintenance functions. Print spooling systems gather document printing requests and process them in the order in which they were received or as resources allow, without user intervention. These batch-oriented capabilities are still critical for

operational efficiency in enterprise computing environments, illustrating how ideas presented in early batch processing systems have proved very useful and in-play even today. Multiprogramming principles are pervasive throughout the design of almost every modern operating system. Modern process management subsystems build on this foundation of multiprogramming, and when the need arose for thousands of concurrent processes, sophisticated scheduling algorithms were introduced that allow balancing of throughput, fairness, and responsiveness. Hardware virtualization techniques allow memory management systems to contain such advanced protection mechanisms which lets multiple processes run in parallel without one process's data being affected by the other. I/O subsystems manage shared devices by multiple processes, using techniques like buffering, caching, and asynchronous I/O to balance throughput against wait time. These functionalities are a natural extension of the fundamental ideas developed in early multiprogramming systems but are also designed to scale parallelism up to the broader parallelism required in contemporary computing environments. Modern computing is a highly interactive affair and so time-sharing principles have evolved to support them. The use of dynamic sections and immediate usability, for example, allows modern user interfaces to give a sense of almost dedicated responsiveness due to always having blocks of resources seemingly available regardless of underlying contexts. Preemptive multitasking allows interactive applications to remain usable even under resource strain from background actions. The advanced scheduling paradigms incorporate ideas such as multilevel feedback queues that shall dynamically change priorities in response to process behavior and tend to favor interactive processes while ensuring progress in compute-intensive background processes. These features also represent the direct descendants of early time-sharing systems, mapped into the context of personal computing, and extended to nurture a wide set of interaction models, across devices and form factors. In the present day, many operating system types illustrate the evolution and convergence of these paradigms. General-purpose operating systems (OSes) such as Windows, macOS, and desktop Linux distributions provide both interactive components and significant background processing by supporting both end-user applications (e.g., browsers, editors) and system services (e.g., drivers). Deterministic

response times are extended to time-critical applications through the use of real-time operating systems, in which many of the scheduling concepts from the three paradigms are extended, and they continue to be used in industrial control, aviation, and medical devices. Concepts such as these are further expanded in distributed operating systems, which work over a network of computers and control processes that may exist on multiple physical machines, providing a single image for users and applications. Cloud operating systems take these ideas a step further, managing resources across entire data centers and dynamically allocating computing capacity to serve variable workloads while ensuring tenants can't interfere with one another. Batch processing was the earliest method of running programs and subsequently evolved into multiprogramming and more recently, time-sharing, providing a foundation for emerging technologies and future trends in operating systems development. In multiprogramming systems, the concepts of process isolation were first introduced, from which containerization and micro service architectures extend. Server less computing platforms combine elements of all three paradigms, delivering responsiveness in an interactive style but managing background processing over shared infrastructure with efficiency. It also helps improve the responsiveness of edge computing systems, which leverage time-sharing principles to partition computing resources closer to users, subject to resource limitations. Though quantum computing environments will need to implement aspects of these classical paradigms likely augmented with new mechanisms to handle the inherent properties of quantum processing. The evolution of operating system paradigms over the years has laid the groundwork for the sophisticated systems we enjoy today, and offers insight into the trends that will define the future of computing, as software must contend with an increasingly complex interplay between technology and human interaction to deliver seamless user experiences.

1.3.13 Conclusion and Future Directions

From the historical perspective of operating systems, the evolution of computing systems from batch-processing to multiprogramming to time-sharing gives us a broad sense of where computing priorities were focused, from hardware maximization to CPU utilization efficiency to interactive-responsiveness. This evolutionary path reflects not only progress in technology, but also changes in views regarding the purpose

of computing and how computing resources should be made available. This next paradigm was born out of the limitations of its predecessors, providing innovative solutions that extended the power of computation at the expense of other priorities. They all share the unifying theme of managing complexity through abstraction and coordination of resources — the core operating system functions. The advent of batch processing brought about the notion of program automation and job management, positioning the operating system as an intermediary or mediator between users and hardware. Multiprogramming extended this mediating role to manage multiple overlapping activities, introducing features of process management and protection that are still core to modern computing. Then came time-sharing that built even further on this foundation; computers engaged users directly and human-computer interaction established patterns (which survive today) in how we interact with a computer. As this evolution surfaced, principles stood out that continue to be surprisingly relevant and well-suited to the future. Operating systems became more composed of and layered upon these initial building blocks and abstractions, which have yet to be challenged by fundamentally better alternatives (excepting certain resource-constrained or bare-metal use cases). These fundamental abstractions have not only persisted since its inception, but have continued to remain relevant even as computing hardware evolved from mainframes to personal computers to distributed systems, proving to be both conceptually powerful and flexible. One can look into the future where operating system design attempts to solve new problems while following the principles laid in the past. The increasing need for security and privacy protection is a result of widespread awareness of the vulnerability of computing systems and the critical nature of the data that they process. The explosive growth of networked and distributed computing environments caused the focus of the operating system be extended from individual machines, to communication, coordination, and resource sharing among complex networks. As new computational paradigms such as quantum computing, neuromorphic systems and ambient computing emerge, they will introduce new operating system requirements while still leveraging the core principles created by the historical evolution of classical systems. Operating systems are evolving as a process and not a product. As technology capabilities grow and user needs change,

operating systems have to evolve alongside them, walking a tightrope between efficiency, security, usability and other competing priorities. The need to evolved from batch processing to multiprogramming to time-sharing, being necessary for a better understanding of that, the ever-evolving need gained is more appreciation of the existing structure, which always has core functionalities then addressed under such needs. That historical view provides useful lessons for making sense of existing systems and predicting future ones. Exploring how operating systems have adapted to competing priorities and responded to emerging needs is revealing of perennial principles likely to inform the design of operating systems across future technology transitions. The evolution from batch processing to multiprogramming to time-sharing is not solely a matter of historical interest but rather, is living history whose effects continue in the form of computing environments, and by extension, modern society's interaction with information technology.

1.3.14 Operating-System Services

An Operating System (OS) is vital software that sits between the hardware of a computer and a user, enabling the user to effectively run programs in a user-friendly environment. It is a software that manages the computer's hardware resources, including the processors, memory, storage devices, and input/output devices, and provides them to all users and tasks. It has to reconcile the often-conflicting objectives of user convenience and efficient utilization of the computer system's resources. Over the years, operating systems have evolved and adapted to new hardware technologies, materials and processing environments, giving birth to specialized operating system designs to meet the diverse needs of different computing scenarios. Operating systems have grown increasingly complex in order to be expanded functionality-wise, security-wise, and providing a support framework for cutting-edge applications, ranging from the first batch systems that processed jobs in a serial way without user-extent to the multi-user, multitasking operating systems of today. Operating systems also have to provide a user interface through which people interact with the computer, and a set of services that programs can utilize. These services are the more technical, lower-level functions that most users never directly use that are necessary for the system and the programs that run on it to operate correctly. Teaching some of the different types of operating systems

and their respective services is a staple in the computer science curriculum as it illustrates the interactions between software and hardware and provides insight into how modern computing environments operate. This is especially important for those who will design or maintain computer systems, develop applications or make decisions about computing infrastructure in an organizational context.

1.3.15 Types of Operating Systems

A range of operating system has evolved over time to cater to specific computing requirements and environments. The first historical type of operating system is a batch operating system, which takes jobs that are similar to one another and minimizes user interaction in order to keep the processor as busy as possible and minimize idle time between jobs; the modern equivalent and still very relevant to any environment where repetitive processing is needed in volume can be found in batch systems of early mainframe systems (e.g. OS/360 from IBM) that keep jobs running through job queues in environments like payroll systems or an environment where something like a scientific batch computation is needed. However, Multi-user operating systems are based upon the idea of time-sharing systems where multiple processes from multiple users can be run on a single computer more or less simultaneously as the processor can switch among user programs extremely fast thereby giving the impression that each user has an exclusive access to the system, it was first implemented in CTSS (Compatible Time Sharing System) in 1960s, streamlining multi-user access in the computer system, this success led to the development of Multics system. For single-user operating systems, the complexity lies in creating user-friendly interfaces and maximizing responsiveness without compromising too much on performance, examples include Microsoft Windows, macOS, and many Linux distros, which cater to individual users (but might not be as optimized as possible for resource utilization). Multi-processor operating system types handle systems with more than one standalone processor or with multi-core processors by utilizing higher algorithms to execute multiple processes over the processing units, while sustaining the system order and stability, which becomes much tougher with every additional processor because it needs to synchronize and allocate simultaneously different resources. RTOS (real time operating systems) provide guarantee that something will happen within a specific period of time; timing is critical in some

applications like industrial control systems, medical devices or avionics; RTOS focuses on deterministic behavior rather than throughput or other performance metrics with examples such as VxWorks, QNX, or RTLinux. Distributed operating systems, e.g., Amoeba, Mach, and recent cloud operating environments manage resources spread over several physically separate computers to create the illusion of single, unified system, no matter the complexity of anthropogenic cally relevant inter-computer (networks of computers) valid operations. Embedded operating systems are tailored for dedicated systems with limited resources, like smart appliances, automotive networks, and Internet of Things (IoT) devices; these systems are optimized for resource-constrained environments and emphasize simplicity and reliability over complex feature sets, with examples like embedded Linux distributions, ThreadX, and FreeRTOS. Network operating systems. Network operating systems (NOS) are primarily concerned with managing network resources and providing connectivity services such as file sharing, printer sharing, user authentication, and network traffic management; e.g., early Novell NetWare, Microsoft Windows Server, and portions of various Unix/Linux distributions configured as network servers.

1.3.16 Core Operating System Services

OS Core Services Overview Every operating system consists of a core services layer. Process management is central to multitasking environments, where the operating system needs to create, schedule, synchronize, communicate between and terminate processes while maintaining a balanced allocation of resources and system stability; this process juggle involves a harmonious process of scheduling algorithms that ascertain which process runs when, by priority, its execution time, requirements for resources, etc. Memory management services allocate and deallocate portions of memory as needed by processes and implement mechanisms such as paging and segmentation to provide virtual memory which gives the illusion to the user that they have more space of available memory than the physical memory present in the system; now to further ensure that processes read only from their own memory, memory protection mechanisms are in place that prevents processes from accessing other processes' memory or the memory of the kernel. Specifically, file system management introduces an important layer of abstraction whereby a user interacts with files and

directories, instead of with the tracks or sectors of a physical storage device (such as a hard disk drive, solid-state drive, or network storage device) it is responsible for managing entities, maintaining the file metadata required and enforcing access control and sharing across users and processes. Device drivers allow applications to interact with different types of hardware components by providing a consistent interface, which reduces the need for applications and the operating system to know the intricate specifics of each device; this abstraction helps operating systems support a wide range of devices, while also shielding software developers from needing to manage each devices unique features. Input/output (I/O) management is the process of handling the transfer of data between the system memory and peripheral devices, through the use of buffering, caching and spooling mechanisms to maximize the performance of the system when the I/O is performed, reconciling the speed discrepancy between the CPU and slower external devices; the efficiency of I/O significantly affects overall system performance, especially in applications that rely heavily on data. They secure the system, the applications, and the data from unauthorized access or modification by means of verifying user identity, with authorization mechanisms that decide what authenticated users can do, and audit logging that records security-relevant events for later examination; in addition to this modern operating systems incorporates several other isolation mechanisms, such as process sandboxing, to mitigate potential security breaches. Error detection and handling mechanisms detect hardware and software faults and trigger recovery procedures if possible or terminate the faulty component so it cannot bring down the whole system (which also applies to exceptional conditions such as division by zero or invalid memory access that would otherwise crash an application or system).

1.3.17User Interface and Interaction Services

Interface Operating systems offer many interfaces from the command line to complex GUI. CLIs refer to text-based interaction using shells like Bash in Unix/Linux environments, Power Shell in Windows or Zsh in macOS by which users enter predefined commands that the OS knows how to interpret and execute; typically not as user-friendly for novices as graphical interfaces, CLIs give you exactness, script ability, and often speed for seasoned users, thus especially useful for automation and tasks in IT. Graphical user interfaces (GUIs) provide

a framework of visual components such as windows, icons, menus, and pointers, permitting intuitive interaction with pointing devices while abstracting away underlying technical complexities and making computers more accessible to non-technical users; notable GUI environments include Microsoft Windows Desktop Environment, Apple's Aqua interface in macOS, and diverse Linux desktop environments such as GNOME, and KDE. Modern operating systems have incorporated voice user interfaces (VUIs) and natural language processing capabilities, enabling users to provide spoken commands and queries by means of assistants like Microsoft's Cortana, Apple's Siri, or Google Assistant in Android; there are advantages to this hands-free operation in certain scenarios, but these interfaces continue to be refined with respect to their accuracy and capabilities. Features that make sure operating systems remain usable by people with a variety of disabilities — screen readers for people with no sight, keyboard-only access for those with motion limitations, closed captioning for people who are hard of hearing, visual modes that help people with low or high contrast sensitivity rely on accessibility services, whether that be Microsoft's Narrator, Apple's Voiceover, or the Orca screen reader for Linux; accessibility services implement frameworks on top of which applications can build to ensure a high level of usability for their software. These include help systems and documentation, which provide contextual assistance, tutorials, and references to help users understand system functionality and troubleshoot problems; these resources have evolved from rudimentary manual pages to interactive, searchable knowledge bases integrated directly into the operating system. Notification services let users know about events in the system, application updates, new communications, or situations that might need attention; these systems have become progressively more elaborate, with fine-grained user control over the notifications that are displayed, and the means by which they're delivered to limit disruption while filtering in important information that the user needs. Configuration and customization services let users change how systems work, how they look, and which applications are default; these systems may include control panels, settings applications, and profile management, which may support maintaining separate configurations for individual users on shared systems.

1.3.18 Resource Management and System Performance

Operating system is tested on a very large and hefty The CPU scheduling algorithms decide which process receives CPU service and how long it does, consequently applying complex policies balancing delivery of throughput, response time, fairness, and prioritizing these requirements; and include round-robin access to each process at everything fixed time quantum, priority-based scheduling where higher-priority processes are favored, and various hybrid methods when workloads are known in advance. Virtual memory and memory allocation virtually expands limited physical memory by effectively treating portions of disk space as an external cache area for running processes and applying page replacement algorithms like Least Recently Used (LRU) or Clock to systematically decide which memory pages to trade out of RAM when the physical-memory is over-allocated; efficient memory management needs to facilitate keeping frequently accessible data in faster physical memory while keeping costly disk operations as low as possible. Storage management services manage the allocation and location of disk space, building file system structures (for example, NTFS on Windows, ext4 on Linux, APFS on macOS) to optimally organize data for storage and retrieval, and may offer advanced data protection features such as journaling to help prevent data corruption in the event of system crashes, as well as volume management to allow multiple physical devices to be combined into a single logical storage unit, and transparent compression or deduplication to optimize and maximize available space. In more detail, energy management services create a profile of components in a system to monitor and control their power consumption by dynamically scaling back processor speed, for example, dimming displays, suspending inactive devices, and employing advanced sleep states; energy management services maintain a balance between performance requirements and battery life issues, frequently tuning themselves to the idiosyncrasies of active workloads as well as available energy. Load balancing, state and memory monitors are tracking the actual utilization of CPU cores, memory and swap utilization, disk I/O and network resource usage, redistributing workloads to prevent bottlenecks as well as visibility to the system to the admins using tools like Windows Task Manager, top command under Linux or Activity monitor on Macs. QoS mechanisms are implemented to ensure that certain applications or services are prioritized, guaranteeing that important functions

receive the necessary resources even when the system is heavily loaded; for instance, a video conferencing application could be assured of sufficient bandwidth and processing priority, avoiding disrupted communication even when other heavy applications may be running on the system. Caching services keep and serve frequently accessed data from their faster memory layers, greatly improving performance by minimizing fetches from slower storage devices, with sophisticated tracking that discriminates not only between processor caches and disk buffer caches, but can also implement sophisticated algorithms that predict what will be needed next based on access patterns and program state.

1.3.19 Networking and Communication Services

Like (or worse than) host rewriting fun, it's well defined by modern Operating systems with a stack of services up to local networks and the world. At the higher level, network protocol support provides the necessary foundation for communication standards such as TCP and IP, ensuring that data transmission across different networks occurs uniformly, irrespective of the underlying hardware differences; most operating systems contain a protocol stack responsible for encapsulating data, directing it toward the appropriate destination, and ensuring reliable delivery despite issues such as network failure or congestion. Network configuration and management services are responsible for other networking tasks such as assigning an IP address (either statically or through DHCP), subnet mask configuration, gateway settings, and DNS server settings; they may also include diagnostic tools responsible for finding and fix connectivity issues through programs like ping, traceroute, or network configuration panels. Remote access services enable users to log into a system from far away and run commands or access files supposedly as though they were in person; they include terminal services (such as SSH in Unix/Linux systems), remote desktop protocols (such as Microsoft's RDP or VNC in cross-platform environments), and Enable secure connections between the user and the system across the public infrastructure, called the virtual private network (VPN) capability. Distributed file systems and network file sharing make files stored on remote computers available as though they were on the user's local machine, typically supporting devices running the SMB/CIFS and NFS protocols in Windows and Unix/Linux environments, respectively, or

AFP, previously, in Apple environments, such services handle the complexities of performing file operations on remote storage, caching, and managing consistency with regard to shared files and multiple users. Network security services safeguard systems against unauthorized access and malicious activities by introducing the likes of firewalls to filter incoming and outgoing network traffic according to a system of predefined rules, intrusion detection systems to monitor for suspicious patterns, encryption services to maintain the confidentiality of data during transit, and other mechanisms; these defenses have evolved in sophistication as network threats have grown more complex. Directory Services allow you to authenticate users in a centralized location and also allow users to search for resources; these authentication systems include items such as Microsoft Active Directory, Open DAP, and Apple Open Directory, which maintain large databases of user accounts, group memberships, and callable resources on the network. Internet and web services integrate browser and related applications and tools into the operating system, providing API for applications to access internet resources; most new operating systems ship with libraries to common internet protocols like HTTP, FTP, and email to facilitate application development and promote consistency of network behavior.

1.3.20 Advanced and Specialized Operating System Services

Beyond essential functionality, modern OS provide advanced services that accommodate specialized requirements and emerging technologies. Hypervisors (such as those used in Microsoft's Hyper-V, VMware, or KVM on Linux) allow multiple operating systems to run at the same time on a single piece of hardware by creating isolated virtual machines with their own allocated resources—leading to server consolidation, testing environments and improved system utilization. Container support, as evidenced by Docker support in many server OSs, provides a lightweight application isolation approach without the performance overhead of full virtualization; alongside, container services manage namespace isolation, resource limits and communication between container-based applications, yielding deployment consistency between development and production setups. Cloud integration is the integration of local operating systems with remote cloud resources, which includes synchronizing files across devices, offering backup services, and even hybrid computing since

processing can occur between local environments and the cloud; examples include Microsoft's Azure integration with Windows, Apple's iCloud services within macOS and iOS, and multiple forms of cloud connectivity within Linux distributions. Modern operating systems increasingly include artificial intelligence and machine learning services, which provide application programming interfaces and frameworks that applications can take advantage of for speech recognition, analysis of images, natural language processing, and predictive functionality; these services often include a combination of on-device processing for privacy and responsiveness and reliance on the cloud for more compute-intensive tasks. These multimedia services are responsible for handling audio and video processing, including hardware acceleration, supporting codecs, and streaming capabilities that enable applications to provide rich media experiences without concerns about low-level details (Windows uses DirectX, macOS has Core Audio and Core Video, and various frameworks are available in Linux distributions). Database and information management services provide structured data storage and retrieval capabilities, either via embedded databases (e.g. SQLite) or standardized interfaces to external database systems; some contemporary operating systems include indexing services that scan catalog file contents for rapid queries, facilitating user workflow by increasing user productivity when finding data. Software update and maintenance services automatically check, download and install updates to the operating system and applications, balancing security vulnerabilities patches and new features with user control by configuring update policies to suit organizations or users; examples of these services are Yum and APT. Services are there to help use different ecosystems, e.g., WSL that runs Linux code on Windows, compatibility layer like Wine to run Windows programs on Linux, or software solutions integrated into OS (virtualization software) that allow the same or another OS to run in the main one.

1.3.21 Conclusion and Future Trends

Operating systems of today are in an ever-evolving phase, aren't they? Multi-core processors and specialized hardware accelerators such as GPUs, TPUs, and custom chips are becoming widespread; this leads to a new generation of operating systems that can effectively allocate resources and schedule jobs to both homogeneous and heterogeneous

workload, offering streamlined, cohesive interfaces to applications and users. Such profound growth of Internet of Things (IoT) devices introduces specific challenges for operating system (OS) design, particularly when you consider that these constrained environments must leverage minimal resources, but still meet unprecedented scale with regards security, connectivity, and manageability, as we're witnessing the arrival of specific IoT OSs alongside the adaptation of existing platforms to operate at some of these limitations. Unlike the traditional centralized cloud model, edge computing necessitates operating systems able to function well with sporadic connectivity, variable resource availability, and strict latency constraints; the distributed nature of the edge model challenges operating system core assumptions about how resources are available to applications and how they communicate. Security and privacy still propel operating system really of their features, notable adoption of hardware-backed security features, and strong encryption of data (the system and application information), containerization for application isolation and a fine-grained permission model for sensitive user information and this will only get better in the future provides with more complex threats. The lines separating diverse computing environments desktop, mobile, cloud, embedded—are rapidly disappearing, with operating systems moving towards more common systems that deliver consistent experiences and application mobility across multiple device clases; notably, Windows running on desktops, tablets, and servers; Linux variants found everywhere from embedded devices to supercomputers. Autonomous computing,

1.3.22 System Calls

Operating systems act as the crucial link between hardware components that execute instructions and the software programs that users interact with on a regular basis. One of the greatest feats in computer science is the design of operating systems that manage the resources of computer hardware while providing standard interfaces that hide and abstract away the complexities that underlie the hardware. The concept that lies behind this interaction is that of system calls. They are the interface between user applications and the protected kernel space, giving controlled access to hardware resources while maintaining system stability and security. This Unit discusses the wide

phonotypical spectrum of OS types and their mechanisms to implement system calls. To this end, we first present the basic principles regarding operating system architectural designs and the importance of system calls within this context, followed by a survey of operating system paradigms (i.e., monolithic, microkernel, hybrid kernel, exokernel and virtualisation). We will explore how the design and implementation of system calls affects important operating system properties including performance overhead, security boundaries, extensibility, and hardware abstraction. This should give students an understanding of subtle differences in system call mechanisms among operating system architectures and highlight key tradeoffs and decisions found in system software in general. You learn how computers operate at a systems level, context that is critical for writing, optimizing, and securing software on various computing platforms.

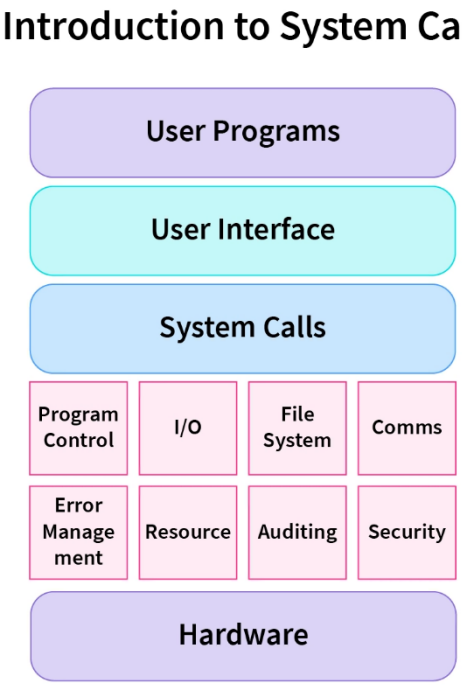


Figure 1.3.9: : System Call
[Source - https://www.scaler.com/]

1.3.23 The Foundation of Operating Systems and System Calls

Operating systems is the most important software in any computing environment and is the fundamental software layer on top of which all other software runs. To appreciate the importance of syscalls, we'll first have a look at how modern operating systems are organized and what their responsibilities are. Operating systems do several important

things: they manage and allocate the underlying hardware resources (such as the CPU, memory, or I/O devices); they isolate and protect separate processes from each other; they implement a file system and networking stacks; and they expose standard interfaces that let application developers write programs without a detailed understanding of the underlying hardware specifications. An operating system is divided into high level and low-level components with respect to privileges. Application programs run with limited privilege and have restricted access to system resources in user mode, but have unrestricted access to memory and hardware devices in kernel mode. It is important to note that this separation is not only a software construct, as it is usually enforced by the hardware itself, for example, by hardware implemented protection rings implemented by the CPU. System calls are designed to be a controlled entry point to the underlying system kernel from the user level applications, allowing user applications to request services that require elevated privileges or access to protected resources. If we look at the history behind system calls, we can see they were born within early time-sharing systems like MULTICS and early UNIX, where these systems needed to manage resource access among multiple users, requiring a more formal approach toward system services. System calls have been a consistent abstraction for decades The fundamental concept of system calls hasn't changed much in decades of operating system designs, although the implementation details and specific interfaces have improved quite a bit. System calls usually require a context switch, which means the processor must switch from user mode to kernel mode to perform the requested privileged operation and then switch back to user mode. This switch is tightly managed and is considered a major milestone along the execution life of any application. Unlike regular library functions, system calls do not return at the entry to user mode and the actual implementation of these is often through a combination of interrupts (software interrupts/ trap instruction or by special CPU instruction depending on the hardware architecture). For example, on contemporary x86-64 machines, the SYSCALL instruction provides a fast mechanism upon an entry of user and kernel mode, while ARM implementations may use the SWI (Software Interrupt) instruction. Hardware mechanisms make sure the transitions go well, going to different layers do not permit access to unwanted memory protected.

System Calls: The system calls can be broadly grouped into the following functional categories: process control (used to create and terminate processes), file management (used to read, write, and manipulate the file system), device management (used to interface with hardware peripheral devices), information maintenance (for data transfer between user and kernel space), and communications (used to communicate between processes and networking). Each operating system implements a different set of system calls, but there are many common operations that run on both systems. For instance, process generation on UNIX-like systems is performed with a sequence of fork() and exec(); on Windows, Create Process is used. Similarly, file operations like open(), read(), write() and close() have equivalents in most OSs, but the parameters and specifics may vary. The number of system calls varies widely by OS an embedded OS might implement only a handful of system calls, while a complex general-purpose operating system like Linux would have hundreds of specialized system call functions. The performance of any operating system depends on the design and the implementation of system calls, since each system call has an overhead due to context switch from user mode to kernel mode. To minimize this overhead, we use a number of optimization techniques; modern operating systems actually use system call batching to combine multiple operations into one system call, as well as fast paths for common ones. In order to understand how the design and implementation of each system call interface differs from those found in other types of operating systems, it is essential we explore this foundation.

1.3.24 Monolithic Kernels and Their System Call Architecture

Unlike the most optimized, derived Micro-kernel, monolithic kernels are the default architecture for historic Operating System design strategy. This architecture defines many mainstreams OSs like classic UNIXs, Linux and the old Windows. By avoiding the overhead that comes with inter-process communication (IPC) or message passing, the monolithic approach provides substantial performance benefits since various components can directly communicate with one another from within the kernel. The primary interface between user applications and kernel services is through system calls in a monolithic kernel. For a monolithic environment, when a user program makes a system call (which is the first step in any system call), a pretty simple chain of

events happens. The first phase is for the application to fill CPU registers with the system call number and any required arguments, and then invoke a special instruction (such as SYSCALL on x86-64 processors) that makes the transition to kernel mode. Using the system call number, the kernel's system call handler finds the right function in a dispatch table, checks the arguments, and performs the requested operation with full kernel privileges. Results are then stored in registers or memory locations that may be accessed by the user program after completion, and control is returned to user mode. The execution path in this fashion leads to reasoned performance efficient patterns that are a hallmark of monolithic designs. Linux (the operating system) is a perfect example of the monolithic approach to system-call implementation but with many modern improvements to the kernel concept. If you get the same name as an application binary interface (ABI) or processor architecture (due to Linux), the Linux system call interface has grown: We maintain a wide range of entry points. For example, while legacy 32-bit applications would use the INT 0x80 instruction to perform system calls, contemporary 64-bit applications usually employ the more efficient SYSCALL instruction instead. Linux takes additional steps to improve system call performance with things like the vDSO (virtual dynamic shared object), which maps certain parts of the kernel memory directly into user space so some system calls can bypass the full context switch overhead. As an illustration, operations such as gettimeofday can be performed fully in user mode if the conditions are proper, leading to a reduction in latency by several orders of magnitude. In Linux, you have a system call table with a unique number for each system call. This table has grown significantly over the years, and Linux kernel version 5.10 has support for more than 400 unique system calls. New system calls come into the kernel as part of a carefully controlled process to minimize backward incompatibility, because system call numbers and interfaces are a key part of the kernel's guarantees about ABI stability. Looking at individual system calls shows us a little bit about this monolithic approach. Take the open system call in Linux which creates or opens a file. The open() system call is the widely known interface, but underneath, the kernel's implementation does so much more: it resolves the file path and permission, traverses the file system directory hierarchy, works with the correct file system driver, allocates the file descriptor and updates

multiple internal data structure, all within the kernel's address space. This highly cohesive, single-context execution is a prime example of the monolithic philosophy behind tightly integrating various system services directly into the kernel. While it has performance advantages, the monolithic approach has some challenges. The single address space design; A bug in any part of the system say in a third-party device driver can crash the whole system or even take control of the entire system. Moreover, the monolithic structure can make it challenging to develop and test new kernel features due to the need to integrate changes into the monolithic codebase, potentially necessitating full system reboots during development. Policies get more complicated too since kernel is running with highest level privileges, and therefore presents a bigger attack surface. These restrictions have driven alternative approaches to kernel architecture, but the performance gains and practical benefits of monolithic kernels have ensured their continued dominance in nearly all computing platforms. Many of these concerns have been mitigated in modern monolithic kernels similar to Linux, which utilize a modular design where components can be dynamically loaded and unloaded, allowing for some of the flexibility of a microkernel while still retaining the performance benefits of the monolithic design. From monolithic kernel

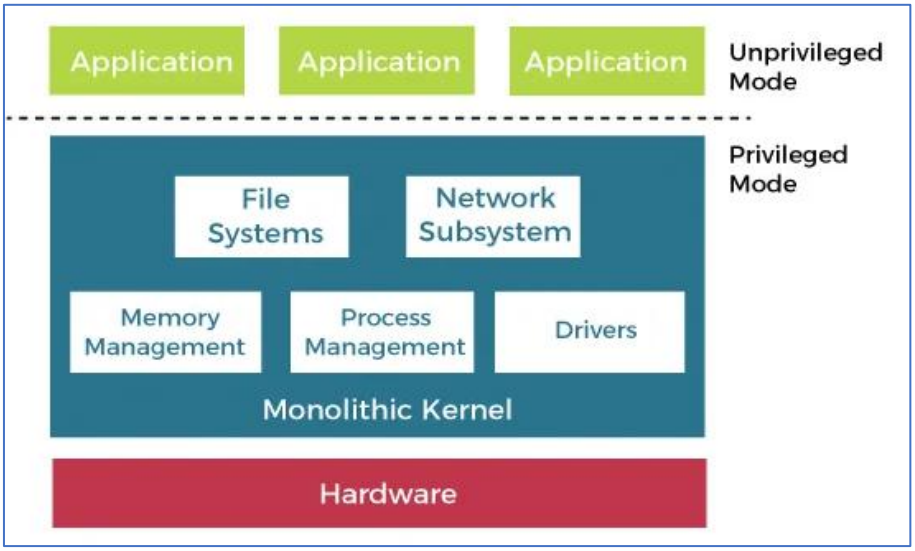


Figure1.3.10: Monolithic Operating System
[Source - https://tutoraspire.com/]

perspective, the implementations of various system calls are still evolving towards some fancy implementations which reduces the latency, enhance security and extensibility at high level, while retaining its kernel architecture for general purpose computing.

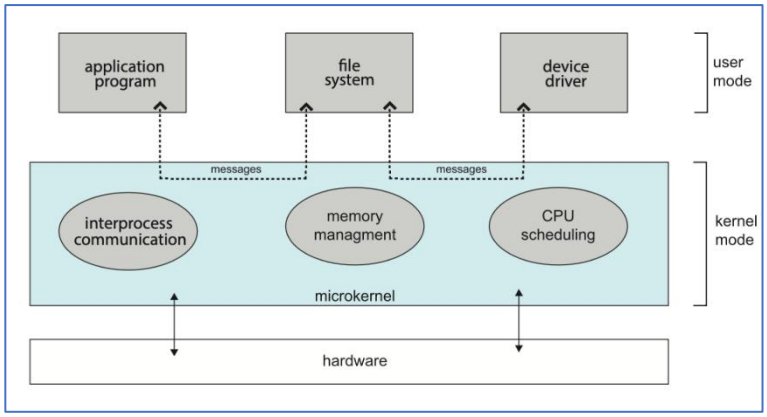
1.3.25 Microkernels: Minimalist Approach to System Calls

Microkernels, in the clearest possible departure from monolithic design philosophy, put only the bare essentials in the privileged domain of the kernel, embodying the most minimalist approach possible to operating system architecture. This set of new ways of thinking about how an operating system works came out in the 1980s, and there were a few systems that became the first microkernel systems like Mach developed at Carnegie Mellon University. Under the microkernel philosophy, only those functions that are absolutely necessary to have kernel privileges usually address space management, thread scheduling, and simple inter-process communication (IPC) are kept within the kernel itself. And, for the most part, all conventional operating system services (file systems, device drivers, networking stacks, process management, etc) are implemented as user-space servers running with regular privileges. This architectural separation also fundamentally changes properties and implementations of system calls as compared to monolithic systems. Microkernel based operating systems ideally reduce the system call interface to less than around 20 core system calls as opposed to hundreds of systems calls in general monolithic kernels. Instead of implementing diverse functionality this building component allows only for system calls that bridge the user application to the numerous separate server processes that facilitate the operating system services. Instead of passing all the arguments which causes a lot of redundancy, the microkernel will just expose system calls that you can call from a process that communicates within the kernel the file operations to be run by this file system server process. When a program wants to read a file, for example, it constructs a message that describes the requested parameters and sends it via the microkernel's IPC mechanism to a file system server; that server processes the request and returns results to the application using the same IPC channel. That indirection makes system service execution paths radically different from monolithic systems. One such brilliant and successful project is MINIX, originally

developed as an educational tool by Andrew S. Tanenbaum, and over the years, evolved to grow into a robust microkernel-based operating system. The entire MINIX 3 kernel comprises only the minimal core functionality: interrupt handling, process scheduling, and basic IPC. System services are implemented by independent processes with least-privilege assigned. And the Virtual File System (VFS) server, which sits atop the actual file system implementations, is responsible for coordinating file operations, delegating work to individual server processes for the file system implementations. Device drivers run as separate user-space processes and communicate with hardware over controlled interfaces offered by the kernel itself. This rigorous isolation results in an architecture where even essential elements, such as device drivers, are prevented from directly accessing memory beyond their designated boundaries, which greatly improves the system immunity. Another well-known example of commercially deployed microkernel architecture, particularly in embedded, automotive, and safety-critical environments, is the QNX Neutrino real-time operating system. QNX employs a message-based architecture with system calls primarily providing synchronous IPC between clients and servers. Using a very small microkernel (100KB or less) that handles memory protection, thread scheduling, and message passing, all other features of the system are implemented in user-space processes. Pros of the microkernel approach here on systems calls Second, it improves system reliability via fault isolation — a crashing device driver or file system server can never corrupt kernel memory; the system can detect and restart individual components without bringing the entire system down. Second, the architecture enhances security by limiting the privileged code base (the “trusted computing base” or TCB) and enforcing least privilege for system components. Third, microkernel’s enable extensibility, as new services can be added without any kernel code changes. Lastly, this architecture has the potential for lowering the cost of the formal verification of the kernel, as exemplified by seL4, a formally verified microkernel designed by NICTA (now a division of Data61) that offers mathematical proofs of its correctness. But the microkernel approach comes with performance challenges mostly concerning IPC overhead. Microkernel systems have historically taken a performance penalty versus monolithic designs, since even basic operations may involve context switching between user processes

multiple times. Many of these issues were addressed by the implementation of modern microkernel's using several optimization methods. As an example, L4 family microkernels sew up IPC paths very tight and efficient via direct process switching and messaging as registers for short messages. Until these optimizations, the performance gap between microkernel and monolithic systems was huge, but still, although it was reduced to the size of a knife, it was never eliminated. Mechanically, the system call implementation is different between microkernel and monolithic systems. Although the basic hardware primitives (like SYSCALL instructions or interrupts) are not much different, the work done in the kernel is often much simpler. Instead of executing complex operations directly, the microkernel typically checks the validity of the system call parameters, delivers the parameters to the corresponding user-space server via a message passing mechanisms and manages the responses. This separation leads to clean interfaces and minimizes the attack surface within the privileged kernel code. The microkernel approach to system calls is a philosophically different vision of operating system design – one focused on modularity, reliability, and security, rather than maximum performance. This leads us to the second point: Microkernels are not the dominant architecture among general-purpose operating systems: Because microkernel architectures involve a higher level of indirection and typically introduce intercrosses communication (IPC) overhead, they have not displaced their monolithic counterparts for general-purpose workloads. Microkernel0 systems are still evolving, with projects such as seL4 and Genode taking this one step further by proving the security / separation guarantees achieved through formal verification.

Figure 1.3.12 Microkernel Operating System
[Source - https://en.wikipedia.org]



1.3.26 Hybrid Kernels and Pragmatic System Call Implementations

Hybrid kernels are a practical solution born out of the theoretical elegance and practical difficulties of pure microkernel designs, incorporating elements from both monolithic and microkernel systems to provide a compromise between modularity and performance. This reflects an understanding that while the strong delineation of components provided by microkernel provides significant advantages for reliability and security, the performance penalties thereof especially for I/O-intensive operations make this a deal-breaker for many real-world workloads. Therefore, hybrid kernels allow performance-critical subsystems to be implemented in kernel space while keeping the microkernel philosophy of modularity and separation for other components. This architectural trade-off has a strong impact on how system calls are designed, implemented and behave in those operating systems. Beginning with Windows NT, Microsoft Windows is probably the most commercially successful example of hybrid kernel architecture. The Windows NT kernel was first designed as a microkernel, separating the kernel-mode Executive services from user-mode subsystems. Though, to mitigate performance issues, the following components previously held in user space in a pure microkernel design (Window Manager, graphics drivers, portions of the file system) were placed in kernel space. To facilitate performance optimization, this pragmatic adaptation achieved a situation where the theoretical boundaries between kernel and user components were blurred. This Windows system call interface, referred to as "syscall" or "Nt" functions (e.g., NtCreateFile, NtReadFile) is the basis for the Windows API. Note that applications normally do not call these native API functions directly, but instead call higher-level libraries like kernel32. dll, which offer the more familiar Win32 API functions

(CreateFile, ReadFile, etc.). This abstraction allows Windows to implement multiple API personalities (Win32, POSIX, OS/2) on top of a single system call interface, and subsystem independence also means centralized access control and validation within the kernel. Now, in Windows, there is a mechanism called the System Service Descriptor Table (SSDT) through which the system calls are addressed in the Windows. When an application advances a system call, the processor switches to kernel mode with the specific hardware instruction (X86–64 uses SYSCALL and older X86 systems use INT 2E), and the kernel's system service dispatcher will use the syscall number to look up and invoke the corresponding handler function. This pattern is somewhat hybrid, as the dispatch mechanism is similar to that of a monolithic kernel, while the actual architecture provides some degree of separation between kernel objects. Another well-known example of a hybrid kernel architecture is Apple's macOS (formerly OS X). The XNU (X is Not Unix) macOS kernel is a non-microkernel that combines the Mach microkernel, a core component of the NeXT STEP operating system, with parts from FreeBSD and Apple's proprietary I/O Kit framework. In this hybrid exercise, the Mach parachute delivers low level facilities like memory management, thread scheduling, and IPC, for its part, the BSD level implements the UNIX system call interface and \networking \stack. The I/O Kit, managed in the kernel, but operating with an object-oriented design, helps increase modularity of device drivers. An interesting case with hybrid design is the system call interface in macOS. Applications access system services through traditional UNIX system calls inherited from BSD, and implemented by XNU directly in the kernel. Many of the macOS-specific services use Mach messages instead of traditional system calls, thus promoting a microkernel-like interaction model for those services. This hybrid philosophy is in action for performance sensitive operations so they are implemented directly in the kernel but other services maintain a more separated message-passing architecture. For example, hybrid kernels implement system calls with various techniques to reduce the user-kernel barrier performance penalty. This can include batching together related operations into single system calls, user-space libraries that reduce system call frequency, or special fast paths for common operations. Both Windows and macOS, for example, provide mechanisms for speeding up certain graphics

operations by allowing user applications to directly access frame buffer memory, with user-mode access being controlled by the kernel, as long as certain conditions are met, thus avoiding the need for kernel access with every draw operation. Sandboxing hybrid kernel system calls: Security minefield Hybrid kernels, while preserving the basic interface between user and kernel modes, may present more opportunities for exploitation than pure microkernel because of the greater architectural complexity and a larger codebase in the kernel itself. In response to these issues, contemporary hybrid kernels adopt several hardening mechanisms, including kernel memory ASLR (Address Space Layout Randomization), control flow integrity techniques, and rigorous parameter validation for system calls. For example, Windows 10 and later use Virtualization Based Security (VBS) features to drive this same theme by using hardware virtualization to further isolate critical kernel components from the rest of the system, achieving a more microkernel-like division for security-sensitive subsystems alongside the performance benefits of the hybrid architecture in normal operation. System calls in hybrid kernels have evolved as a result of this pragmatic approach to changing needs. Both Windows and macOS, for example, have added mechanisms to allow kernel extension (filter drivers, kernel extensions, or the entire open-source core kernel) which third-party software can utilize to view and telescope back into system call behavior without modifying the underlying core kernel. However, both have evolved over time toward kernel extensibility models that are more constrained than what either system started out with (Driver Kit in macOS and Windows Driver Framework), moving a lot of this functionality into user mode, suggesting a slow return toward a microkernel model for these particular things. Hybrid kernels in short capture an interplay between the theoretical ideal and the practical upper bound of performance based on considerations of security, compatibility, and architectural purity.

Specialized Operating Systems and Unique System Call Paradigms

Outside of the standard categories of monolithic, microkernel, and hybrid architectures are specialized operating systems that are tailored for niche computing environments and use cases. Such specialized systems often tailor their system call implementations in a way that bears little resemblance to traditional system calls, focusing instead on

characteristics like real-time constraints, execution security, or limited resources. Exploring these other paradigms, in turn, illustrates the inherent wiggle room in the system call concept, and its ability to trade off different requirements. RT systems (real-time systems) are one example of a specialized system with a unique implementation of the system call interface. In hard real-time systems where missing a deadline can mean failure or even disaster determinism and predictability are more important than average-case behavior. Real-time operating systems (RTOS) implementations such as VxWorks, FreeRTOS, and RTLinux modify the classic system call way to provide bounded rate execution and minimized interrupt latency. For example, many RTOS designs do disable interrupts in critical sections of system call processing, preventing lower-priority interrupts from preempting high-priority tasks. Also, RTOS system calls usually implement priority inheritance protocols to avoid priority inversion situations in which a high-priority task is blocked waiting for a resource from a low-priority one. RTOS environments, for example, usually provide a system call interface that includes dedicated APIs for fine-grained timing control; absolute and relative sleep functions; high-resolution timers; and predictable scheduling APIs. These specialized interfaces take into account the unique needs of real-time applications, where the timing of the response is just as important as the function of the response. Exokernels are a radically different way of thinking about operating system design, representing an extreme minimalism that exceeds even microkernel. Originally conceived by researchers at MIT in the mid-1990s, exokernels remove almost all abstraction from the kernel, exposing hardware resources to applications directly through a narrow interface of multiplexing primitives. Exokernel systems avoid using traditional system calls such as read or write and use low-level hardware access operations instead. Exokernels expose only physical resources like disk blocks, memory pages and network interfaces, instead of higher abstractions like files or processes, and their system calls are centered on safely multiplexing these resources. Applications (or their accompanying library operating systems (libOSes)) implement higher-level abstractions depending on what is needed in the application layer. For instance, rather than a traditional read system call which acts on abstract files, an exokernel might expose primitives to directly manipulate specific disk sectors, and the file abstraction is fully

user-space. In such a model, domain applications gain the highest possible level of control and performance, because kernel abstractions are removed, and they can implement exactly those resource management policies that the applications need. Exokernel MIT implementation showed better performance for specific applications but with more competence code development. A more recent specialized approach, unikernels take us even further and destroy the classic distinction between operating system and application altogether. A unikernel system compiles the application together with only those parts of the operating system that it needs, into a single-address-space executable that runs directly on virtualized hardware. Unikernel implementations, like MirageOS (in OCaml), IncludeOS (C++) and Unik, often cut out traditional system calls altogether, substituting them with calls to functions in the OS libraries slotted right in with the actual application. Because the whole system operates on a single privilege level, this scheme leaves very little overhead for user-kernel transition. Although unikernels give up general-purpose functionality (like multi-tenancy), they do offer large benefits around security (lowering the attack surface), performance (removing mode transitions) and resource efficiency (images are measured in MB, not GB). The system call interface essentially becomes the API of the included OS libraries and the boundaries between application and OS code become fuzzy or get completely obliterated. For example, secure operating systems such as seL4, Genode, and KeyKOS utilize syscalls that have been explicitly designed to ensure strong security and isolation properties. In these systems, capabilities (unforgeable tokens used to denote access rights to resources) typically replace or augment conventional system call interfaces. In contrast to specifying resources by identifiers (like the numbers of file descriptors or process identifiers), system calls in capability-based systems operate on capability references that implicitly represent both the resource identity and the operations that it permits. This radically redefines the security model of system calls, as access rights are proved via possession of capabilities rather than via explicit permission checks in the system call implementation. Instead of opening the file by path and checking permissions against user credentials (as done in traditional UNIX systems), a capability system would have the application present a directory capability, and obtain a derived file capability through a controlled operation. Container-style

OSes and library OSes are another flavor of system call implementation. Other systems, such as gVisor from Google, intercept system calls made by containerized applications and reimplement them in the Go programming language, providing a higher level of isolation and compatibility. gVisor intercepts these system calls through its potential PTRACE platform (using ptrace) or its KVM platform (acting as a KVM guest), essentially providing a backing implementation of every single system call and mediating access to the host kernel. By going with the existing concept of system calls to know how to approach the security, Flexi gate can turn a traditional process into a safer process. LibOSs, such as Graphene-SGX, do the same, running applications inside Intel SGX enclaves and interposing on system calls to the host system that can be reached via a secure interface. A common approach adopted by many networks operating systems (e.g. Cisco IOS, Arista EOS or Cumulus Linux) is to build up a specialized system call interface mostly covering network configuration and monitoring instead of general-purpose computing. Because of the proprietary nature of the hardware and the needs of network equipment in general, these systems tend to present proprietary APIs alongside more typical interfaces. In some cases, these systems employ restricted or modified standard system call interfaces to block operations that would otherwise impact the networking functionality or security. Perhaps the biggest deviation from the standard system call story is for embedded operating systems that run on highly constrained devices. In very severely resource-constrained environments like microcontrollers with kilobytes of RAM, traditional system call mechanisms may be outright too costly in terms of the amount of code they require and the execution overhead that they incur. In systems such as TinyOS and Contiki, traditional system calls are largely replaced by event-based programming models (e.g., with event queues) or direct function calls, both of which eliminate the mode transitions and context switches often seen with more conventional system calls. Sometimes the entire operating system might run at a single privilege level, and be protected using features of the programming language or just careful code review rather than through hardware means. Particularly for the operating systems that do not follow the traditional Unix architecture and their individual techniques to system calls, this underscores the intrinsic pliability of the system call idea and allows

for adjustments to myriad needs. By exploring these other paradigms we can learn more about the tradeoffs embedded in system interface design and how a unique system call mechanism can be suited to particular operating environments and needs.

1.3.27 Virtual Machines, Containers, and Layered System Call Implementations

The rise of virtualization technologies has added new tiers to the system call model in terms of functionality and behavior, as system calls cross numerous barriers in multi-layer architectures. Neither is a trivial question, especially in our modern computing infrastructure where applications tend to run in evermore nested environments than sitting directly on the metal. To add further complexity, you must know how system calls work inside these layered sectors to comprehend the performance, security, and compatibility features of these systems when they undergo virtualization or containerization. In contrast, hardware virtualization uses hypervisors such as VMware ESXi, Microsoft Hyper-V, Xen, and KVM, which implement a series of resources in a virtualized type of virtual machine (VM), imitating a complete computer, including virtual CPUs, memory, and devices. From a system call perspective, this architecture introduces a massive complexity: system calls made by applications within the VM are first handled by the guest OS running inside the VM, not by the host system controlling the physical hardware. This kind of indirection establishes a multi-layer execution path which operations go through to eventually access physical resources. When an application running inside a VM makes a system call, the usual mechanisms (SYSCALL instruction, software interrupt, etc.) trap to the guest OS kernel. Lack of information from the host machine means the guest kernel processes the system call in a normal way, as if it were running on physical hardware. However, when the guest kernel tries to access hardware i.e. if it wants to write to a disk or send network packets it interacts with virtual hardware devices that the hypervisor provides. These interactions most often lead to additional transitions from guest to hypervisor (or VM exits/hypercells) and inject additional context switches on the execution path. As an example, here is a possible scenario for a simple write to a file from an application running in a VM: (1) the application performing a system call to the guest kernel; (2) the guest kernel creating an I/O request to its virtual disk; (3) a VM exit to the

hypervisor when the guest tries to talk to its virtual disk; (4) the hypervisor translating this request to something dealing with the underlying storage hardware, which may involve making system calls to the host OS; and (5) completion of the physical I/O and walking back through all of those layers. This layering comes at the price of performance overhead, especially for I/O bound workloads. To overcome these limitations, modern day virtualization systems use several optimization techniques. Para-virtualization is a technique to modify the guest operating system so that it communicates with the hypervisor through special hyper calls and cannot directly access the virtualized hardware, thus aiming to decrease the overhead of trapping and emulating privileged instructions. Features like Intel VT-x and AMD-V (surprisingly, those don't always get detected properly) enable more optimized transitions between the guest and host contexts. Further, methods like direct device assignment (pass-through) enable VMs to communicate directly and use physical hardware for critical devices, bypassing some of the layering overhead. Instead of relying on this last deployment model, containers provide an alternative virtualization strategy by using OS-level mechanisms without using hardware emulation to make isolated environments. Container technologies, such as Docker, LXC, and Kubernetes pods, take advantage of kernel features like namespaces and control groups (cgroups) to build isolated process environments without needing the complexity of full hardware virtualization. Containers provide a very different model from hardware virtualization at the system call level. In containers, applications perform system calls directly to the host kernel, with no intervening guest operating system layer. On the other hand, these system calls are filtered, redirected, and translated between namespaces in various ways to change their behavior with respect to non-containerized programs. Container runtimes use system call filtering modes to limit the set of system calls available to any given containerized application (e.g., seccomp-bpf for Linux). This filter decreases the kernel attack surface visible to the potentially malicious app, so it is more secure. A containerized web server, for instance, could be allowed to perform certain network-related system calls and disallowed to perform others that modify kernel modules or access unauthorized file systems. Namespace virtualization changes the semantics of many system calls when inside containers. When a

containerized application makes a system call that references global resources such as process IDs, network interfaces, or mount points the kernel resolves these references to the global resources in accordance with the mappings set for the namespace associated with the container. An example of this can be found when considering that the process inside the container would see itself as PID 1 (the init process), while in the global namespace of the host system effectively assigning the container process a different PID. Likewise, when the containerized process tries to reach the root file system, these operations get mapped to a container's designated root directory through mount namespace mappings done by the kernel. These translations are invisible to the application but radically change the impact of system calls made by an application depending on how the container's namespaces are configured. Advanced container security mechanisms such as gVisor and Kata Containers provide extra layers of system call handling. gVisor is a user-space kernel that intercepts and reimplements the system calls from containerized apps while providing an isolation boundary beyond ordinary container isolation. Instead of sending container system calls directly to the host kernel, gVisor emulates them in its Sentry component, and fulfill them over the more limited interface to the host. Kata Containers follows a similar pattern, whereby containers are executed inside lightweight VMs, a hybrid of sorts where the system calls of the container are handled by a guest kernel inside a tailored virtualization VM. Server less computing and Function-as-a-Service (FaaS) platforms have added yet another layer of system call complexity. The code gets executed in highly controlled environments, sometimes with custom system call interception and a virtualization (e.g., AWS Lambda, Google Functions, Azure Functions) when developers deploy functions to the platforms listed above. Thus, these platforms basically employ a mixture of container technologies, special library interposition, and custom so-called runtime environments to deliver secure and isolated execution environments while still trying to ensure high efficiency for short-lived function invocations. These technologies stack on top of each other and can result in complex paths for the system calls a function running in a server less platform might be running in a container that runs in a VM, with system calls potentially traveling several layers of interception, filtering and translation before reaching physical resources. System call security has

special significance for virtualized and containerized system. Every layer of virtualization generates attack surfaces for more security boundaries but also potential attack vectors at the borders between layers. Hypothetical example: Hypervisor vulnerabilities may allow guest operating systems to escape their VM boundaries, whereas container escape vulnerabilities are providing examples of how system call implementation bugs can be exploited to bypass namespace or capability restrictions. Current research efforts in this domain focus on concepts like hardware-enforced isolation, formal verification of security properties and least-privilege models with respect to system-call permissions. Understanding the complex interplay of system calls across virtualization boundaries is necessary for performance analysis and optimization in these layered environments. Tools like Linux's eBPF (extended Berkeley Packet Filter) tracing enable developers to discover which system calls dominate in a mixed environment and where the most important performance bottlenecks appear (as they often cross container and virtualization boundaries). Likewise, various hardware capabilities, such as Intel Performance Monitoring Units (PMUs), can allow the detailed measurement of the impact that virtualization has on the performance of system calls. Innovation in system call implementation and optimization is still ongoing, driven by the evolution of virtualization technologies. Emerging solutions like Firecracker (used by AWS Lambda), lightweight hypervisors that are tailored for container workloads, and unikernel-based isolation techniques are example of continued attempts at striking the right balance between the security advantages of enforced isolation and the performance needs of modern cloud applications. It is important to understand how system calls work across these layers of abstraction in order to design, deploy and debug your applications on modern virtualized infrastructure.

1.3.28 The Future of System Calls: Innovations and Emerging Paradigms

In light of a wide variety of hardware, security needs, and application trends, the operating system and system call interface has continued to evolve. These frontier technologies & research areas indicate a paradigm shift in how applications will interact with O/S, and the core idea of system calls, which has been quite steady for decades. The final section examines emerging technologies and theoretical concepts that

may reshape system call design and implementation paradigms on diverse operating system architectures in the years to come. One important trend is a growing deployment of hardware extensions to improve system call security and performance. Modern processors include special features that enhance and protect privileged transitions. Intel CET (Control-flow Enforcement Technology) and ARM PAC (Pointer Authentication Code) prevent return-oriented programming (ROP) and jump-oriented programming (JOP) attacks that could abuse the system call interfaces. Likewise, AMD's Secure Encrypted Virtualization (SEV) and Intel's Trust Domain Extensions (TDX) add hardware-enforced divide between the virtual machines, resulting in the modification of how system calls work in virtualized environment and providing cryptographic isolation of guest's memory. Such hardware innovations enable new methods of implementing system calls that do not compromise security for performance. An example of such an optimization is the usage of user-interrupt by Intel — it reduces the number of contexts save and restore calls when going from user mode to kernel and the other way round. The increasing need for these types of systems is helping shape new approaches to system call design as part of efforts to create new confidential computing and trusted execution environment (TEE) initiatives. Intel SGX, ARM Trust Zone, and AMD SEV are environments that establish execution contexts in which even the operating system kernel is untrusted. Such models often use specialized "ocalls" (calls from enclave to outside) and "ecalls" (calls from outside to enclave) that might replace (or augment) traditional system calls, with cryptographic protection that guarantees that sensitive data is protected even if we have to utilize services from the untrusted operating system. Technologies like Asylo from Google, the Open Enclave SDK from Microsoft and the Enarx project are showing how these new systems call paradigms could end up being transformed to accommodate confidential computing over a range of hardware technologies. The recent proliferation of programmable I/O devices, most notably smart NICs and computational storage devices, is forcing a rethinking of the syscall interface for I/O. Instead of funneling all of their I/O through the operating system kernel using traditional system calls, applications will increasingly communicate directly with smart peripherals through memory-mapped interfaces, RDMA (Remote Direct Memory Access),

or specialized programming frameworks. SPDK (Storage Performance Development Kit) and DPDK (Data Plane Development Kit) are some of the technologies that allow existing high-performance applications to bypass system calls for the common I/O operations in favor of more direct hardware access, which will only become more common as devices are equipped with dual-purpose CPUs capable of executing those workloads onboard. So, the programming languages environment and runtime systems are also shaping system call evolution. With runtimes like Was time, Wasmer, and WAMR (Web Assembly Micro Runtime), Web Assembly, originally defined to only run compiled code in the browser, is now growing into the server as well.

1.3.29 Operating-System Structure

An operating system (OS) is software that acts as an intermediary between applications and the computer hardware, managing hardware resources and providing a user environment in which programs can be performed conveniently and efficiently. The operating system structure describes how the components of the operating system interact with each other and with the hardware underneath. Over the course of computing history, OS designs have changed from monolithic OSes, to multi-layered and distributed OS architectures. The earliest operating systems date back to the mid-1950s as simple control programs for batch processing on mainframe computers, performing little more than sequencing through jobs and managing input/output. As computing technology progressed through the decades, operating systems expanded to support interactive time-sharing, real-time processing, distributed computing, and the wide variety of personal and mobile computing environments we have today. The architecture of an operating system has a great impact on its performance attributes, fault tolerance, maintainability, and application to specific computing environments. Different structural approaches make different trade-offs between these attributes, but there is no single best design for every use case. In this Unit, we will analyze the main types of operating system structures we have, their strengths, weaknesses, and when to use the structure. We'll explore monolithic systems where the code base is tightly integrated, layered systems that organize functionality hierarchically, microkernel architectures with minimal privileged code, modular designs that minimizing loose coupling with component

isolation. Per each architectural type, we will discuss the philosophy behind the design, implementation aspects, performance implications, and real-world examples. We will also look at how new emerging technologies, such as virtualization, containerization, and cloud computing are shaping the range of available operating systems and their layout. The students will also know when and why certain features will become important along the history, various trade-offs made in order to achieve a workable system, and understand how Operating system organization is achieved in many systems.

1.3.30 Monolithic Operating Systems: Comprehensive Integration

The first and arguably simplest approach to operating system design, monolithic operating systems, refer to a set of operating system components that reside in a single location, with all system services running in kernel space with hardware access. Monolithic: The entire operating system, including the kernel, device drivers, file systems, memory management, process scheduling, and inter-process communication mechanisms, runs as a single program in a single address space in a privileged mode. This architecture was prevalent from the 1960s on, with systems like UNIX and its descendants, and is still reflected in contemporary systems like Linux, FreeBSD, and, at least in a part, Windows, although the latter has taken on aspects of other architectural ideas too. The primary benefit of the monolithic approach is performance, as components of the system can communicate by calling functions rather than having to pass messages or utilize other inter-process communication methods, which tend to be more expensive. All components operate in the same address space, and so data structures can be shared directly without the overhead of copying data between protected memory domains. In early computing environments, where hardware was scarce and expensive, monolithic systems dominated due to their performance. In a monolithic kernel, the functionality within the kernel itself is often organized as many logical layers, with the low-level hardware interfaces at the bottom and higher-level application interfaces at the top, but this layered implementation is not so much enforced by hardware protection mechanisms, but by software conventions. Early UNIX systems can be thought of this way, where the lowest layer was hardware management and the next layer up was memory management, process scheduling had its own, and file systems had theirs, and at the highest level was a syscall interface where each level was separate but depended on lower layers. Monolithic architectures, despite their performance advantages, pose considerable challenges for development, maintenance, and reliability of the system. Because the code base is unified, a bug in any component from a device driver to the virtual memory system can potentially crash the entire operating system since all code runs with full hardware privileges. With so much dependency between components, the system can be particularly vulnerable, and debugging can be a challenge, as bugs that originate in one subsystem

can rear their ugly heads elsewhere in the system. In addition, the construction and development of monolithic systems need to be coordinated carefully between teams working on various components with one subsystem change can have domino effects all around kernel. The most common modern approach has been to maintain good performance whilst still implementing some of the benefits of modularity through loadable kernel modules, as the monolithic architecture evolves. This allows components like device drivers to be dynamically loaded and unloaded from the kernel at runtime, thereby increasing system flexibility and incremental updates without a complete boot. For instance, Linux implements a rich module system that allows it to support a tremendous number of hardware devices and specialized functionality while keeping its base kernel relatively small. Modernizing monolithic operating systems such as Linux learn and thus apply rich development and testing processes to address the intrinsic weaknesses of their design. Additionally, a variety of testing and debugging tools, like automated tests and code inspection frameworks, can catch bugs before they go live, combined with an extensive review process, help keep the system stable through such a massive, complex code base. To solve this, there are certain techniques that have been introduced, like kernel preemption and fine-grained locking to improve the responsiveness and scalability on multiprocessor systems, which used to be the weaknesses of the monolithic design. Although monlithic architectures are more problematic than new comers to computing and newer architectural paradigms have been added to mitigate those problems, monolithic systems such as Linux continue to be popular, suggesting that the performance advantages and practical effectiveness of monolithic architectures can remain relevant in modern computing environments, especially in the arena of hardware classes geared toward server systems in which are senriced by applications where performance and hardware support breadth are the primary products of arguable utility. It is a great example of the evolution of monolithic systems, showing how a solution that looks relatively simple from an architectural viewpoint can be improved and progressively developed to cope with new needs while keep its weaknesses under control and, most importantly, keep its advantages unscathed.

1.3.31 Layered Operating Systems: Hierarchical Abstraction

Layered OS type is a structured way to design the system, where functionality is divided into brows of functionalities, high level functionalities are provided using lower layer, And abstraction of services of lower layer are provided to upper layers. This architectural paradigm is derived from some of the early theoretical work on structured programming and systems design in the early 1960s that was applied first through systems such as the (TechnischeHogeschool Eindhoven) operating system developed by EdsgerDijkstra and later in commercial systems like Multics. A layered architecture is one wherein a strict hierarchy is maintained, so a component at layer N can only make use of services offered by components at layer N-1 and lower N (i.e. a layer N service cannot access a service or data structure provided directly by a layer N-2 component. The main theoretical benefit of this strict layering is that we can work on and validate each layer in isolation, with clearly defined interfaces between adjacent layers giving us well-rounded boundaries for testing and validation. The concept of a layered operating system also typically involves functionalities to form layers, from the lowest hardware dependent level, to the highest user-oriented level. The first (bottom) level may deal with physical hardware resources and interrupts, the next layers manage memory, processes, inter-process communications, virtual memory, file system, the higher most layers with user-interfaces and applications. You are provided with layers of abstraction, where each layer obscures the complexities of the layer(s) below and translates the naked hardware into the ornate computational environment experienced by users and applications. A major goal of an early operating system called THE, built in the late 1960's, was to implement this paradigm, and THE itself was divided into five levels: process management, memory management, console management, input/output buffering, and user programs. With this level of clean separation, once the lower layers had been verified, the upper layers could be independently tested in systematic debugging of the entire system. But the IBM PL/1 system in the 1970s ushered in a new model with the Venus operating system, which had six distinct layers to tackle the many facets of process and resource management. Although conceptually elegant, the strictly layered model has practical challenges that, in practice, have made strictly following it a challenge in modern systems. The loss of spatial

locality and strict hierarchy can add a huge performance cost, because something that could happen in a single monolithic system must now cross multiple levels of indirection, each of which might involve a context switch or even transform data. Also keep in mind that many Operating System functions do not have a natural hierarchy: services such as security, logging, or power management cut across various layers of the system and do not really fit a single layer. Additionally, rigid layering can make it challenging to implement efficient inter-process communication and synchronization mechanisms, which frequently depend on direct interaction between the components residing within diverse layers. In the face of such practical constraints, most modern operating systems take a more flexible approach to layered architecture but still retain their organizational principles, with carefully controlled breaches of strict layering where performance or functionality require them. For instance, while Windows uses a layered kernel architecture, with kernel components nested in various tiers, it enables some cross-layer optimizations to improve system throughput. Layering does end up looking something like this in modern systems, but more through a combination of practices, interface definitions, and documentation than through strict hardware boundaries enforced between all layers. This more pragmatic strategy retains much of the software engineering advantages of layering while avoiding many of its worst performance penalties. The multi-layered OS model still inspires OS design, especially in contexts like certain real-time and embedded systems where reliability and verifiability are more important than sheer performance. The idea also manifests in the way software development teams are organized and documentation trees are structured for complex operating systems, even where the underlying implementation likely offers more freedom than a rigidly layered model might imply. In practice, contemporary systems often integrate layered design components with various architectural styles, resulting in hybrid architectures that capitalize on the advantages of different paradigms and offset their respective drawbacks.

1.3.32 Microkernel Operating Systems: Minimalist Core Design

Microkernel operating systems are an architectural evolution from monolithic operating systems, based on a philosophy of reducing the amount of code that executes in privileged mode to the minimal set of components necessary to facilitate computing. This is an architectural

style that is developed in the 1980s and early 1990s with systems such as Mach, which was developed at Carnegie Mellon University, and has formed the basis of many systems including QNX, MINIX, and parts of macOS through its XNU kernel. Microkernel architecture's primary realization lies in the fact that only the services that truly require privileged or specialized access (generally IPC, basic memory power management, and minimum scheduling) should be implemented within the kernel itself, while the rest will run as user interaction processes which will have limited control, hence limiting the risk of impacting the entire system. This strict separation is intended to ensure greater reliability, security and maintainability of the system by limiting the trusted computing base (TCB) and isolating failure-induced components. The microkernel approach has a solid, multi-sided theoretical advantage. Because it minimizes the amount of code that must run in privileged mode, the system becomes less susceptible to catastrophic failure — for example a crash of a user-space file system server need not bring down the entire operating system, as it would have to do in a monolithic design. This provides better fault containment, as you can restart individual servers without bringing down the entire system. Similarly, security benefits arise due to the reduced attack surface that the minimal kernel exposes and provides fewer opportunities for privilege escalation attacks by targeting kernel vulnerabilities. From a software engineering perspective, the microkernel approach makes it easier to implement systems with noticeable modularity, allowing development teams to focus on specific servers with well-defined channels between components. This modularity furthermore allows for extensibility of the system, as new services can be introduced as user-space servers without any need to update the microkernel itself. Also, the architecture in theory provides greater portability, with hardware-dependent code mainly residing at the microkernel level and in low-level device drivers, meaning that porting the system to new hardware platforms is easier. Microkernel System Design The advantage of microkernels is their small size; everything most applications could need is implemented as a distinct service that a monolithic kernel would contain, leading to high levels of modularity but high communication costs as the interposes communication needs to be deeply efficient given the nature of a microkernel architecture where each service operates in its own space.

A user application sends a message to the file system server, for instance, when it has to perform a file system operation; the file system server can send another message to the disk driver server, and the microkernel takes care of this communication. Early hardware microkernel implementations (the most notorious entry here being Mach) suffered painful performance penalties due to the overhead associated with such message passing and the resulting context switches between address space. QNX, a commercial real-time operating system with a more efficient implementation, yet more forgiving of lower-performance hardware, especially in embedded systems where timelines take precedence over other performance statistics. The shortcoming of the pure microkernel approach from a performance standpoint triggered many refinements and hybrid implementations. L4 (originally developed by JochenLiedtke in the 1990s) was a second-generation microkernel that achieved astonishing rates of interposes communication by virtue of careful design and implementation, demonstrating that much of the theoretical overhead of microkernel could be eliminated by amazing amounts of optimization. macOS (formerly known as OS X) comes with a hybrid approach on its XNU kernel, which incorporates the Mach microkernel and a monolithic UNIX kernel into a single address space, trading some of the fault isolation benefits for a performance improvement. Although Windows NT has been designed with microkernel principles, more and more components were incorporated into kernel space to address performance issues. Notwithstanding the above compromise, the conceptual impact of the microkernel architecture is far-reaching. Andrew Tanenbaum's MINIX 3 was another early but significant example, originally developed as an educational tool but then substantially evolving into a research system, providing demonstration of how microkernel principles continue to be refined, with a focus on reliability through isolation of components. The seL4 microkernel, developed by NICTA (now part of Data61), is possibly the most important recent development in this area and allows the formal mathematical verification of certain properties that could only have been accomplished at all due to the very smallness and clean design of a microkernel. Though many recent systems may incorporate elements beyond the classic microkernel philosophy, the microkernel's optimization towards reducing privileged code and decoupling

systems have undeniably left a mark on contemporary OS design, specifically where security is of the utmost importance, such as in embedded systems and those with high reliability constraints. Microkernel systems are one place where this same idea plays out and demonstrates how an architectural solution can continue to take shape and drive innovation in the field even while the pure idea struggles to be relevant in certain contexts.

1.3.33 Modular Operating Systems: Component-Based Architecture

Modular operating systems are an architectural shift based on clean interfaces between well-defined systems (modules) rather than strict layering or minimal priv. execution. This approach came to prominence in the 1990s and 2000s with systems such as Solaris (and its Spring research predecessor) and Windows NT, which included substantial modular design facets in them, even though they weren't strictly modular in every way. The first of these innovations, Low-level modular architecture focuses not on the vertical stack (layered systems) or privilege levels (microkernel) used to organize elements of a system, but rather on the set of interfaces defining the interactions between components of a system, and allows any component of a system to be developed, tested, and replaced independently of the other components it interacts with, provided they adhere to agreed-upon specifications of interaction. Modular architectures allow separate components at the same conceptual level (i.e., layers) to interact horizontally with each other in ways that are easier to express away from strictly layered systems, enabling natural expression of cross-cutting concerns and cross-layer functions. The principles that drive the design of modular operating systems are based closely on object-oriented programming principle, where system components implement their internal workings behind a well-defined interface which describes both services offered by the component and services the component require from other components. Thus, it creates a system of modules, which are interdependent on each other but are connected via explicit interface declarations as opposed to implicit dependencies, enabling better comprehensibility and maintainability of systems. In the ideal modular architecture, the system is represented as a graph of components, with edges representing module dependencies mediated through interfaces, instead of a stack of layers. Modular operating systems usually feature

a component framework that handles loading, initialization, and communication between modules. For example, Microsoft's Windows Driver Model (WDM) and later Windows Driver Framework (WDF) enable device drivers to work together in a way that was previously impossible by establishing standardized interfaces and support infrastructure that allow dozens able devices to be implemented as independent drivers but still interact in an orderly manner within the driver stack. Jigsaw, as the Java-based project is called, inspired Java 9's module system, which embodied the same principles regarding module dependencies and encapsulated implementations within the context of programming language runtime environments. The modularity approach, spurred by Solaris 7 and 8 major redesigns of the solaris operating system, adopted the heuristics of ServicePlex architecture a means of layering system significance into distinct removable parts with standard interfaces. This allowed for things like dynamic reconfiguration of system services without the need for a reboot a market first, for enterprise systems where availability requirements often mean upgrading the whole system cannot be taken down in order to perform the upgrade. The subsequent implementation makes use of methodologies like dynamic linking, runtime service discovery and component registration to design a flexible yet strong system architecture. Benefits of modular design go beyond just software engineering to operational considerations. In a modular architecture, it may be possible to implement “hot-swapping” of components, allowing for updating or reconfiguration of the system without downtime which is an important property of high-availability environments, such as telecommunications systems or financial services infrastructure. Moreover, the modular structure enables the use of different configurations for various use cases or hardware platforms by allowing components to be included or excluded as needed without major changes to the rest of the system. However, we find many of these same advantages pale in comparison to the nominal performance efficiency of non-modular, tightly integrated designs. Runtime overhead in these frameworks may be caused by interface compliance checking, dynamic binding between components, or even potential need to convert data from one module to another. Moreover, the challenges of developing and maintaining an application tend to escalate proportionally to the number of components involved when

there are explicit dependencies to manage. The challenge of comprehensive testing becomes more complicated with the increase in the number of interchangeable modules, as the number of potential combinations of the components grows exponentially. A hybrid approach often features in modern modular operating system designs, which retains a unified kernel core while enabling modular extensibility via precisely designed frameworks. The Linux kernel, despite being fundamentally a monolithic kernel, introduces a substantial layer of modularity via its loadable kernel module system allowing for dynamic extension of kernel functionalities while preserving performance within the core system. Windows also has such a driver model, but keeps a much more tight base system with options for modular extension. For Example, The prevalence of micro service architecture in distributed systems as well as containerization technologies are part of the continuing evolution of modular approaches, applying similar principles of componentization at a higher level. So again, considering the evidence that we have been exposed to, it would seem that pure modularity has potentially created some pseudo-components that ultimately do not yield the fruitful experience one may want but the overall principles of modularity component isolation, interface-based design and explicit dependency management remain bedrock of every level of the system that we interact with. And as computing environments further diversify and specifications deepen (even if only at particular segments of a community), I suspect that the flexibility provided by such modular design approaches will remain valuable; as long as it is married pragmatically to efficiency with respect to performance and complexity.

1.3.34 Hybrid Operating Systems: Pragmatic Integration

Hybrid operating systems serve as a pragmatic amalgamation of the various architectures that can be seen on the operating system spectrum, containing some monolithic, layered, micro, and modular features to balance performance, reliability, maintainability, and flexibility. Hybrid systems do not follow strictly any communication structural philosophy but choose portions of each architecture that fit the specific system functions and operational needs. This pragmatic idea has held sway over commercial OS design since the late 1990s, and all mainstream OSes today from Windows, macOS, iOS, and Android to

modern Linux distributions are, to varying degrees, hybrids. The key argument for hybrid architectures says that because different parts of the system exercise different demands with regard to performance, availability and development flexibility, you cannot lag the same architectural solution for everything in the system. While performance or stability may benefit from implementing network protocol stacks in-kernel, experimental file systems may be better written as user-space components that crash without taking down the rest of the system. A hybrid system allows different subsystems to follow different architectural models, enabling the most appropriate design approach to be used on a given part of the overall system in pursuit of performance, pragmatism, and real-world usage instead of theoretical purity. One of the more recognizable examples of the hybrid approach is the macOS (formerly OS X) operating system, which features a hybrid kernel called XNU that integrates components of the Mach microkernel and BSD Unix in a single privileged execution environment. Although this design loses some of the fault isolation benefits of the pure microkernel approach, it greatly increases performance by removing the message-passing overhead for frequently used services. At the same time, the I/O Kit driver framework, the BSD subsystem, and Mach-based underpinnings are kept distinctly separate within the system, creating internal boundaries that provide a great deal of potential for maintainability with minimal impact on performance. Likewise, the Windows version implemented by modern Windows products also follows hybrid architecture principles, with a mixture of aspects of monolithic integration, modularity and layering. Running in privileged mode, the Windows kernel delivers essential services: the Hardware Abstraction Layer (HAL), which insulates a lot of the system from specifics of the hardware, memory management, process scheduling, and an elaborate object manager. On top of this foundation the Executive services provide higher-level functionality such as the registry, security reference monitor, and I/O system. Even though the various components execute in kernel mode for performance purposes, they adhere to well-defined abstractions with a modular organization that supports independent development and testing. The Win32 subsystem, along with various other environment subsystems, operate partly in user mode, illustrating a practical separation of function across privilege levels determined by security and stability concerns rather

than strict architectural dogma. Linux has developed an extremely successful hybrid, retaining much of the performance advantages of its monolithic roots while also integrating concepts from alternative architectural paradigms. We will use the term core kernel to refer to such a high-privilege mode codebase, as the core kernel operates as a single privileged-mode entity, but implements an extensive module system to allow components such as device drivers, file systems and networking protocols to be loaded and unloaded dynamically. This method maintains performance efficiency with improved extensibility and maintainability. Moreover, much of Linux's functionality has been gradually pushed to user space when it makes sense to do so, with systems such as FUSE (File system in Userspace) allowing file systems to be written and run without modification to the kernel, as well as container technologies such as Docker and Kubernetes that offer user-space isolation mechanisms achieving many of the objectives of microkernel-process separation without compromising on performance. Notable advances in the hybrid model have indeed been made, particularly in mobile environments such as Android and iOS, providing privilege separation and process isolation of third-party applications which would be the primary threat of an untrusted environment. Therefore, the logic behind Android is to isolate applications into their own process spaces with limited permissions over the core system services that run with elevated privileges. Running on an XNU-derived kernel fused with Mach and BSD components in a security model that embraces sandboxing at the per-app level, iOS also takes a layered approach to security just like the Android variant. The performance benefits of hybrid designs are significant in multi-purpose operating systems that have to accommodate a wide range of often conflicting requirements. Hybrid systems have the potential to apply different architectural principles to different aspects of the system, allowing them to optimize performance for performance-critical paths, provide reliability through isolation of less stable components, enable development through modularization where it makes sense, and maintain backwards compatibility within existing software ecosystems, all in a single, coherent operating system. They illustrate that real systems are not mere implementations of theoretical models but rather the solutions of engineering problems whose challenges outstrip elegant abstractions — such that real systems are

more like hybrid architectures, borrowing from different architectural paradigms. It is this pragmatic synthesis that continues to define modern operating system design, with each new generation assimilating the lessons of multiple architectural traditions, while responding to new hardware capabilities, security threats, and application demands. Due to the continued diversification of computing environments across a wider range of form factors extending from embedded systems to cloud infrastructure, the versatility of hybrid approaches may prove useful in the construction of systems that fulfill their intended purpose, rather than over-commit to a single design (which is associated with a set of trade-off in the articulation of competing objectives of design).

1.3.35 Specialized and Emerging Operating System Structures

In addition to the mainstream architectural paradigms described earlier, a large number of specialized and new operating system architectures have emerged to meet specific computing configurations, workloads, or design objectives. These tailored architectures often serve as narrowly considered modifications of established methods toward specific goals or as novel constructs made possible by technological advances and changing computational models. The evolution of computing — from "general purpose computing" across embedded systems, mobile devices, cloud infrastructure and new advanced platforms, including wearable's and IoT devices — has made these particular structures more and more relevant in the operating system landscape. For instance, real-time operating systems (RTOS) are tailored for predictable, deterministic behavior, as opposed to maximum average throughput. Such operating systems (OSs) test and operate to strict specifications to guarantee response times for time-critical operations, and often employ specialized scheduling algorithms, such as rate-monotonic and earliest deadline first scheduling, rather than the fair-share algorithms found in general-purpose OSs (like those in the UNIX family). This has architectural implications that tend to polarize: you want to reduce non-deterministic system behaviors like dynamic memory allocation, virtual memory page faults, or complex caches that contribute to timing variability. Some real-time systems use microkernel designs to increase reliability (as in QNX) while others focus on minimal execution intervals and slim designs that are akin to stripped-down monoliths showing a degree to

which functional requirements can dictate architectural design stronger than theories of software organization can. What are embedded operating systems? Embedded operating systems are the systems created for such resource-constrained environments as industrial controllers, automotive systems or consumer electronics and often end up having highly tailored architectures tuned for their limited memories, processing power and energy budgets. An example is TinyOS, which implements a component-based architecture but adopts static composition; the system is built at compile time rather than run time, allowing developers to avoid the cost to bind components dynamically, while losing flexibility. Embedded Linux variants commonly minimize the standard kernel removal of excess components, and static methods where dynamic mechanisms are unnecessary. These systems are examples of scaling limits that are pushing innovations in architectures that would never work for general-purpose computing but are exceptionally suitable for their targets. Network Operating System: Another specialized category is called distributed operating system which is a distributed version of an operating system, meaning that the OS services are extended to multiple networked computers and make it appear as a single coherent system to its client. Distributed computing systems such as Amoeba (1980-1999) took process migration, distributed shared memory, global resource naming, and similar features from single user distributed systems and implemented them across multiple physically independent networked computers. Although pure distributed operating systems have had limited commercial success, fundamental aspects of their architecture have been incorporated in most contemporary cloud infrastructure and cluster computing frameworks. First, Google's Borg system (the inspiration for Kubernetes) comes in as a brilliant solution for its distributed resource management and scheduling across a cluster of data center machines, functioning as a distributed operating system at the cluster level even while regular OSs are running on single machines. Virtualization has even given rise to entire new hypervisor architectures that reconfigure the OS actualization on the hardware. Hypervisors like VMware ESXi, Microsoft Hyper-V, and Xen serve as thin abstractions atop physical hardware to multiplex it among multiple guests operating systems, providing them the illusion that they are operating on exclusive hardware. In addition to requiring efficient

mechanisms for hardware abstraction, these systems emphasize having effective inter-virtual machine isolation and low performance overhead, often resulting in designs that closely resemble microkernel with a small trusted computing base but specialized to virtualization primitives over generic operating system services. Architectural implications of virtualization undergo changes at the syscalls layer such as binary translation, par virtualization and/or hardware-assisted virtualization that fundamentally alter the behavior of operating system code interacting with the underlying hardware. Container-based systems provide a lighter-weight form of virtualization and have led to additional innovations in architecture. Unlike VMware, Virtual Box, or similar technologies, which virtualized at the hardware level, Docker, Kubernetes, and other related technologies virtualized at the operating system level, allowing multiple isolated user-space instances to share the same kernel. This model requires namespaces and the architectural support to have isolated namespaces, resource control mechanisms and multi-tenancy in the kernel level, needs that have already driven mainstream kernel development and facilitated new deployment and orchestration avenues in higher levels of the stack. The principle of separating the protection of resources from their management is pushed even further by Exokernels and library operating systems, which enable abstraction of resources at the application level. In these systems, demonstrated by MIT's Exokernel research and recently commercialized through systems like Unikernel, the kernel simply gives very low-level protection and multiplexing for resources, while applications link directly to library implementations of standard operating system services. This design takes away the distinction between application and operating system, which may further reduce overhead and enable applications to impose resource management policies according to their own requirements. Things like MirageOS compile high-level application code alongside only the OS components a particular application needs into a specific image to run directly on virtualized hardware, the commercial embodiment of those principles in practice; The emergence of heterogeneous computing architectures with specialized accelerators (e.g. GPUs, TPUs, FPGAs and other domain-specific processors) has provided significant new pressures for operating systems innovations. Systems now have to not just manage traditional CPU resources but also allocate, schedule, and

provide programming models for these heterogeneous compute engines. This has resulted in traditional operating systems being extended with new subsystems that mediates device- specific memory management, task scheduling and data movement, forminga hybrid architectures that incorporates multiple computational paradigms within a single system. The fundamental nature of computing is changing, and with it new architectural approaches. From unikernel designs that package applications with minimal operating system services inside specialized virtual machines to server less computing models that remove operating system concerns entirely from the developer workflow to edge computing paradigms that distribute computation across networks of devices from sensors near the physical world to cloud servers in the way of their own using novel storage and networking abstractions, these challenges have forced innovation on the structure of operating systems such that they are a vibrant space of ongoing design engineering. These specialized and evolving architectures show that operating system design is still a lively field that continues to change in response to new hardware capabilities, new application needs, and new computing paradigms. Instead of converging on one optimal shape, operating systems continue to diversify to meet an ever-growing set of computing needs and scenarios, and architectural innovation occurs across the spectrum from microcontrollers to global-scale distributed systems. These approaches highlight the notion that operating systems are engineering artifacts: elegant in theory but compromised by practicalities and imperatives that are often very different from the original requirements.

1.3.36 Conclusion and Future Directions

Indeed, the subsequent coalescing of various operating systems structures is an ongoing process, driven by the intersection of theory, engineering, technology and application needs. Operating system structures thus have a long evolution from early monolithic systems optimized for performance and hardware utilization to modern hybrid architectures that selectively embrace elements across multiple paradigms, continuously adapting to new demands yet balancing the inevitable tradeoffs of conflicting design goals.7 So, this adaptation process reflects both the timelessness (tymaldb probably has things like partitions, normalization, sharding, etc, etc) of some basic architectural concepts plus the pragmatic flexibility needed to use them in different

computing environments. Feeding into the ongoing development of OS architectures are a number of trends that the future seems to hold. The rise of heterogeneous computing architectures integrating specialized processors alongside general-purpose CPUs creates new resource management challenges that may catalyze even more structural innovation. Operating systems must increasingly orchestrate computation across diverse processing units with their own programming models, memory architectures, and performance characteristics, a need that tests the limits of traditional process and memory management abstractions designed for homogeneous systems. This trend might hasten the adoption of the more explicitly parallel and distributed architectural models even within single-machine operating systems. Security and reliability issues become more important as computing systems are more deeply integrated into critical infrastructure (power systems, transportation, etc.) and daily life. These priorities often favor architectural approaches that emphasize isolation, least privilege, and minimal trusted computing bases — principles long promoted by microkernel and capability-based designs. With hardware support for virtualization, memory protection and secure execution environments steadily improving, the performance penalty historically associated with these more formally secure architectures is diminishing, making it feasible for them to be used widely in mainstream systems. The growth of edge computing the distribution of computation between everything from IoT devices to cloud data centers challenges traditional operating system boundaries and resource management models. By 2030, future operating systems may have to work efficiently across such distributed settings, coordinating resources, moving data, and determining where computation occurs across heterogeneous networks rather than on single machines. This may compel the fusion of traditional OS structures with distributed system paradigms, leading to emergent hybrid adjacently woven architectures spreading across device ecosystems, while exposing consistent interfaces to applications and end users. Virtualization is still remodeling how applications, operating systems and hardware interact. Applications downloading services provide for greater compos ability, and can serve as the basis for a move toward more library-like systems, allowing applications to have only the system services they need. The distinctions between

application and operating system become less and less clear in this transition. Domain like machine learning, augmented reality, autonomous systems have emerged where the workloads exhibit different characteristics and have different requirements that can drive domain specific architectural innovations Specialized operating system structures that would be radically different from general-purpose ones optimized for traditional interactive and server workloads might be needed for real-time constraints, massive parallelism and probabilistic computing models. Enabling persistent memory technologies that weaken the traditional boundaries between volatile memory and persistent storage break common operating system abstractions and might drive architectural updates in file systems, memory management, and process models. Systems tailored to make the most of these new technologies might take on structures that differ substantially from those optimized for the strict hierarchy of memory-storage elements that's been the hallmark of computing for decades. These trends indicate that we have just scratched the surface, and operating system structures will only become more and more diverse rather than converging on the one true path. We believe different computing environments and workloads will continue to require specialized architectural approaches, although particular fundamental principles modularity, appropriate abstraction, separation of mechanism from policy, and efficient resource utilization will remain applicable to many different implementations. Perhaps the best lesson regarding the history of operating system structures is one of pragmatism: the ability to adapt to changing requirements and capabilities rather than faithfully adhering to any given architectural paradigm is what best characterizes successful systems design. Operating system developers must therefore appreciate both the theoretical underpinnings of these varieties of structure and the engineering requirements that colour their application in individual circumstances. The ideal system architect will combine principle and pragmatism, creating a new generation of operating systems which will effectively serve the needs of all users, applications and computing environments. But as we end this abstraction on operating system structures, and it is necessary to state that operating systems are a field of continuous development, full of new problems and innovations. (The architectures explored in this Unit are not just dead ends, but growing traditions that remain in the DNA of

contemporary system design and will help guide future a direction as computing continues to evolve into exciting new domains, form factors, and application spaces.)

1.3.37 Design Goals

Introduction and Fundamental Concepts

Indeed, the subsequent coalescing of various operating systems structures is an ongoing process, driven by the intersection of theory, engineering, technology and application needs. Operating system structures thus have a long evolution from early monolithic systems optimized for performance and hardware utilization to modern hybrid architectures that selectively embrace elements across multiple paradigms, continuously adapting to new demands yet balancing the inevitable tradeoffs of conflicting design goals.7 So, this adaptation process reflects both the timelessness (tymaldb probably has things like partitions, normalization, sharding, etc, etc) of some basic architectural concepts plus the pragmatic flexibility needed to use them in different computing environments. Feeding into the ongoing development of OS architectures are a number of trends that the future seems to hold. The rise of heterogeneous computing architectures integrating specialized processors alongside general-purpose CPUs creates new resource management challenges that may catalyze even more structural innovation. Operating systems must increasingly orchestrate computation across diverse processing units with their own programming models, memory architectures, and performance characteristics, a need that tests the limits of traditional process and memory management abstractions designed for homogeneous systems. This trend might hasten the adoption of the more explicitly parallel and distributed architectural models even within single-machine operating systems. Security and reliability issues become more important as computing systems are more deeply integrated into critical infrastructure (power systems, transportation, etc.) and daily life. These priorities often favor architectural approaches that emphasize isolation, least privilege, and minimal trusted computing bases — principles long promoted by microkernel and capability-based designs. With hardware support for virtualization, memory protection and secure execution environments steadily improving, the performance penalty historically associated with these more formally secure architectures is diminishing, making it feasible for them to be used

widely in mainstream systems. The growth of edge computing the distribution of computation between everything from IoT devices to cloud data centers challenges traditional operating system boundaries and resource management models. By 2030, future operating systems may have to work efficiently across such distributed settings, coordinating resources, moving data, and determining where computation occurs across heterogeneous networks rather than on single machines. This may compel the fusion of traditional OS structures with distributed system paradigms, leading to emergent hybrid adjacently woven architectures spreading across device ecosystems, while exposing consistent interfaces to applications and end users. Virtualization is still remodeling how applications, operating systems and hardware interact. Applications downloading services provide for greater compensability, and can serve as the basis for a move toward more library-like systems, allowing applications to have only the system services they need. The distinctions between application and operating system become less and less clear in this transition. Domain like machine learning augmented reality, autonomous systems have emerged where the workloads exhibit different characteristics and have different requirements that can drive domain specific architectural innovations Specialized operating system structures that would be radically different from general-purpose ones optimised for traditional interactive and server workloads might be needed for real-time constraints, massive parallelism and probabilistic computing models. Enabling persistent memory technologies that weaken the traditional boundaries between volatile memory and persistent storage break common operating system abstractions and might drive architectural updates in file systems, memory management, and process models. Systems tailored to make the most of these new technologies might take on structures that differ substantially from those optimized for the strict hierarchy of memory-storage elements that's been the hallmark of computing for decades. These trends indicate that we have just scratched the surface, and operating system structures will only become more and more diverse rather than converging on the one true path. We believe different computing environments and workloads will continue to require specialized architectural approaches, although particular fundamental principles modularity, appropriate abstraction, separation of mechanism from

policy, and efficient resource utilization will remain applicable to many different implementations. Perhaps the best lesson regarding the history of operating system structures is one of pragmatism: the ability to adapt to changing requirements and capabilities rather than faithfully adhering to any given architectural paradigm is what best characterizes successful systems design. Operating system developers must therefore appreciate both the theoretical underpinnings of these varieties of structure and the engineering requirements that color their application in individual circumstances. The ideal system architect will combine principle and pragmatism, creating a new generation of operating systems which will effectively serve the needs of all users, applications and computing environments. But as we end this abstraction on operating system structures, and it is necessary to state that operating systems are a field of continuous development, full of new problems and innovations. (The architectures explored in this Unit are not just dead ends, but growing traditions that remain in the DNA of contemporary system design and will help guide future directions as computing continue to evolve into exciting new domains, form factors, and application spaces.)

1.3.38 Batch Operating Systems: Maximizing Throughput and Resource Utilization

Operating systems are the most critical link between computer hardware and the software applications that provide value to users. Operating systems are sophisticated software ecosystems meant to resolve computational resources for performance, giving core services to applications, and to offer interfaces that are human and machine accessible. The development of operating systems has been inextricably linked to the development of computer hardware, with each generation of operating systems reacting to and facilitating novel possibilities in computing hardware. The evolution of operating systems: from the first systems, this simply loaded programs up sequentially into memory, through to modern complex environments that manage distributed resources over global networks. Operating systems must balance competing objectives: isolation or controlled communication, security or accessibility, reliability or failure, high performance or fairness. The architecture of an operating system is ultimately a game of tradeoffs: architectural choices differ radically based on what the system is optimizing for. This fundamental trade-off

dynamic has spawned a diversity of OS types, each optimized for specific use cases and environments. Batch systems care more about throughput than interactivity, real-time systems care more about predictability than general performance, distributed systems care more about availability than simplicity, and desktop systems care more about user experience than raw performance. That makes these distinctions important for students of computer science, since the operating system one chooses has fundamental implications regarding the applications that can be built on top of a given operating system, the performance of the operating system, and what guarantees can be given to users of applications built on top of a given operating system. In this Unit we discuss the different types of operating systems that operated as the backbone of computer systems and analyzing their goals and architectures as time progressed in computer science innovation. These variations and the particular problems they solve give us a sense of both the depth of diversity in computing environments, and the wide principles that underlie all operating system design. Operating System Design Goals Operating systems are about more than abstract design goals they inform the features, shortcomings, and usability of our computing systems. While exploring these differing approaches, we will find overlapping themes in how the designers of these systems manage complex requirements, balance competing objectives, and address the timeless issues of resource management and coordination of processes. From the embedded systems controlling household appliances to the massive cloud infrastructures powering global services, operating systems form the fundamental layer upon which all applications run, thus making their study critical to understanding modern computing.

1.3.39 Interactive and Time-Sharing Systems: Prioritizing User Experience

Interactive operating systems were a radical paradigm shift that changed the way we related to computers, turning computing machinery from batch processing calculators into systems that could respond in something more like closer to human thought processes and work practices. Interactive systems are characterized by the presence of a loop that delivers timely responses to user commands, giving the appearance of having the machine dedicated to the user even when resources are being shared among many users or processes. Of

particular note in this cohort was time-sharing, which enabled multiple concurrent user interactions with a single system by multiplexing control through a rapid switching of attention on the system by the operating system in order to keep the computing environment feeling responsive. The initial rise of such systems in the 1960s as illustrated by groundbreaking projects such as MIT's Compatible Time-Sharing System (CTSS) and even MULTICS (Multiplexed Information and Computing Service) were not merely technical advancements but a philosophical reimagining of what the computing experience should be: that a computer was a utility that could be always on for many users (as opposed to a constrained resource that should be carefully docketed). This highlights the central design goals of interactive systems: maximizing response time to the user at the expense of raw throughput (the measurement of how much work a computer can do), leading to complex scheduling algorithms trading fairness for interactive performance. These systems provided preemptive multiprogramming, in which the operating system could interrupt running programs after very short time slices in order to make sure that no individual program hogged system resources to the detriment of interactivity. Another important result of time-sharing research was virtual memory, a mechanism that made it possible for programs to run as if they had access to more memory than what was physically present, paving the way in those days for more sophisticated applications and more efficient use of the memory among many users working at the same time. With interactive systems, user interfaces evolved significantly, moving from command-line interactions, through early graphical user interfaces to the rich multi-touch and voice driven interfaces we know today. This evolution is illustrative of the continuing effort to make computers usable for the non-expert while giving powerful abstractions to the more knowledgeable. The need for protection mechanisms in multi-user systems led to great strides in security since these systems were required to prevent users from interfering with each other's processes or data. Compounding this demand, architects further needed to ensure that different processes (types of applications) could not interfere with each other, so memory protection, file access controls, and user authentication systems were developed to meet these needs and laid the groundwork for modern computer security. This is perhaps the most recognizable form of interactive systems, with

personal computer operating systems including Microsoft Windows, Apple macOS and multiple distributions of Linux, representing the maturation of several decades of interactive system development, bringing time-sharing ideas that had previously been developed for mainframes into personal computing environments. Modern personal computers are optimized for a single user rather than multiple concurrent users, but the low-level mechanisms created for time-sharing (such as preemptive multitasking and virtual memory with the concept of protection rings) are absolutely essential for multiple concurrent applications and system stability. Besides the command interface itself, interactive systems also introduced concepts like the shell (command interpreter), hierarchical file systems, and graphical windowing systems, which remain critical to how users engage with computers today. The focus on human factors in system design has resulted in rich research in the human-computer interaction literature highlighting that technical performance metrics do not capture system quality well enough — perceived responsiveness, consistency, and usability translate directly to productivity and user satisfaction. We are taught that the transition from batch to interactive computing is one of the great paradigms shifts in all of computing and has had effects on how computers are designed, programmed and used. The change shows how operating system design goals directly impact technical architecture and the overall computing experience, including hardware design, programming languages and application capabilities. As computing moves ever forward toward more natural, context-aware interfaces, the lessons we learned in the formative days of interactive computing are still relevant guideposts to find balance between technical constraints and human needs.

1.3.40 Real-Time Operating Systems: Ensuring Predictable Timing and Reliability

A Real-Time Operating System (RTOS) is an operating system with a real-time application that processes data as it comes in, typically without buffering delays. Real-time systems differ from general-purpose operating systems in that most general-purpose operating systems optimize for average performance rather than guarantee someone meets deadlines, which is essential in applications like industrial automation, automotive control systems, aerospace, medical devices, telecommunication infrastructure and more. What sets real

time systems apart from others is predictability; that is, meeting constraints to response times within bounds, even at peak loads or under stress. Such determinism is enforced by custom scheduling algorithms, avoidance of stochastic mechanisms such as virtual memory, and careful attention to interrupt latencies and context switch overhead. Real-time systems fall generally into hard real time, in which failure to meet a deadline constitutes system failure (like aircraft flight controls or automotive anti-lock braking systems), or soft real time, where infrequent failures to meet a deadline degrade quality but don't cause catastrophic failure (like multimedia streaming or telecommunications). This distinction has a very strong repercussion on architectural choices, as hard real-time systems typically use static resource allocation and worst-case execution time analysis to deliver absolute guarantees. For instance, the real-time operating system employs a fundamentally different scheduler than that of a general-purpose system, using algorithms such as Rate Monotonic Scheduling (RMS) that allocate priority based on the frequency of tasks, or Earliest Deadline First (EDF) which dynamically determines priority based on which process has the impending deadline. These strategies help assure that the right resources get to important tasks in time to meet their limitations even if they need to be run in front of less time-critical operations. Paging and virtual memory techniques that allow for indeterminate timing behavior are usually avoided in memory management for real-time systems, in favor of static allocation or controlled dynamic memory allocation with bounded allocation times. I/O operations similarly make predictable timing characteristics by, e.g., using direct memory access (DMA) and dedicated hardware for the transfer of data between the I/O device and the processor without being the bottleneck. Some commercial RTOS implementations like VxWorks, FreeRTOS, QNX, and RTLinux are mature and cater to a wide range of industries with diverse requirements of certification, reliability, and performance. It is worth noting that these systems include functionality rarely found in general-purpose operating systems such as priority inversion prevention protocols, deterministic inter-process communication mechanisms, and timing and synchronization features. Verification techniques come into play specifically for the development of real-time systems because they cannot just be functionally correct, but they also need timing analysis;

in most cases, formal methods are used to prove that a real-time system meets its deadlines under all possible operating conditions. This level of rigor is crucial for safety-critical applications where timing failures may threaten human lives or cause immense economic loss. Embedded systems, a related category of systems often using real-time operating systems, impose an additional set of constraints with limited resources, power efficiency and specialized hardware interfaces. These devices, from basic microcontroller applications to complex multi-core systems, frequently require specialized OS environments designed to optimize resource utilization but still support real-time guarantees. Another modern trend in real-time systems is the implementation of time-sensitive networking protocols that can extend timing assurances across distributed systems. Hypervisors that run both real-time and non-real-time operating systems on one hardware are also trending. Finally, artificial intelligence techniques are continuously applied to real-time systems while providing timing productivity. The changing landscape of real-time systems Several applications in mainstream computing are now emerging requiring timing guarantees that were once only employed in specialized domains, such as virtual reality, autonomous vehicles, and industrial IoT hence the increasingly ubiquitous importance of the principles of real-time computing. The design of real-time operating systems is an example of how fundamentally divergent goals yield tightly divergent architectural choices even while performing the same basic sets of functions for process management, memory allocation, and I/O handling. We have been allowing our systems to become predictable instead of faster, and this shift has enabled important applications that become critical where failure is not an option, such as space exploration and medical devices that sustain human life. As computation becomes more tightly woven into physical systems that interact with the world in real time, the principles that drove the development of specialized real-time operating systems are finding wider applications throughout the computing stack.

Distributed Operating Systems and Network-Centric Approaches

They are a significant evolution from conventional single-node computer focused models to a distributed environment where many Linked computers operate as a single virtual computer. Whereas traditional operating systems control resources on a single computer, distributed systems coordinate across multiple machines that might be

spread through worldwide networks, working together for goals that the machines couldn't achieve individually. These systems developed due to the exponential growth of networked computing and the need for scalability, high availability, and resource sharing between organizations. Distributed systems are designed to achieve a set of common goals, including location transparency, allowing users and applications to access resources without the need to know the physical location, fault tolerance, where the system maintains the availability of services even in case of any component failure, scalability, where the system grows with the increase in the number of users and resources, and also geographic location of the resources, and consistent performance irrespective of the hardware heterogeneity of the system. To satisfy these objectives, more sophisticated mechanisms for communication, coordination, resource management and failure detection and handling are needed which go far beyond the needs of standalone systems. The architectural approaches to distributed operating systems vary widely, from completely decentralized peer-to-peer systems in which all nodes are functionally equivalent, to hierarchical architectures with specialized management nodes. Client-server models are another common approach which provide the benefits of both centralization and distribution by splitting the functionality of the system between service providers and consumers. More recently, micro service architecture has gained popularity as a paradigm for building distributed applications, decoupling functionality into small independently deployable services that communicate over a well-defined interface. Messages can be delayed, delivered out of order, or not delivered at all. Distributed operating systems facilitate different forms of communication (such as remote procedure calls (RPC), message passing, or distributed shared memory), and use advanced protocols to address those uncertainties. Clock Synchronization is another fundamental challenge, since every node has its own idea of time that may deviate with respect to others, complicating the ordering of events and carrying out time-dependent operations. In general, process management in distributed systems consists of traditional scheduling, process migration (which could be as simple as moving running processes to other nodes based on load balancing or resource access), detection of global deadlocks, and global resource allocation. This allows the system to make more efficient use

of available resources across the network as a whole while still delivering acceptable levels of performance to individual users and applications. Especially for distributed systems, one of the most challenging parts is data consistency and duplication, because if we keep more than one copy, it might lead to higher availability and performance, but when we update the data, it would cause inconsistency. Distributed operating systems support a range of consistency models from strong consistency that gives the illusion of single copy, to eventual consistency that allows for temporary divergence with corresponding tradeoffs in terms of performance, availability and programming complexity. Several distributed operating systems stand out, including: Amoeba (VrijeUniversiteit Amsterdam), Chorus (A microkernel-based OS that started the revolution for distributed systems), and more recently, Borg and Kubernetes from Google that schedule containerized applications on Beowulf cluster. Though only a few pure distributed operating systems have seen extensive commercial adoption, their principles have had an immense impact on modern computing ecosystems. Cloud computing platforms such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform use many ideas of distributed operating systems at a massive scale, providing users the illusion of an infinite amount of resources that are available on demand. Virtualization technologies, which allow multiple logical systems to share physical hardware, have become foundational techniques for deploying distributed systems, allowing resources to be abstracted, isolated and managed in heterogeneous environments. Finally, some models of Distributed Systems today have autonomous management capabilities, which allow them to learn the best configuration to use through machine learning and adjustment through AI. Security issues in distributed settings are when challenges become most disturbing when continuous attack surfaces grow with another type of node and every communication channel. They should also implement comprehensive security architecture with proper authentication, authorization, export risk management and intrusion detection across organizational and geographic boundaries. The gradual replacement of existing operating systems with distributed ones indicates a paradigm shift in the understanding of what computing means and what it can achieve, moving from snapshot- or image-based computation to holistic systems

that are defined more by their connectedness than by their individual components. As computer technology continues to progress towards more and more distributed models from edge computing at the network periphery through to global cloud infrastructures human-centered design principles as discovered through distributed operating system research remain vital in guiding system designers aiming to strike the right compromise of performance, reliability, security and manageability across diverse, complex networks.

1.3.41 Specialized Operating Systems: Tailoring Design to Unique Requirements

Outside of the general categories of languages there is a broader operating systems ecosystem which addresses niche requirements or constraints in specific domains. These specialized systems are instances of how the bare metal principles of operating systems can be repurposed and reshaped into something new that is uniform and guided by some base constraints, often resorting to extreme design choices that would be completely unthinkable in any sort of general-purpose computing, hence perfect for their dedicated environment. The most common subcategory here is embedded operating systems, which power the billions of dedicated computing devices baked into everything from cars and appliances to industrial equipment and consumer electronics. These systems are typically resource-constrained in terms of memory, processing power, energy consumption and reliability, many of them requiring deterministic operation over years of continuous operation without user intervention. Operating systems such as FreeRTOS, Zephyr, and Rethread all prove that embracing a more minimalist design approach can deliver a capable operating system with memory footprints in kilobytes rather than gigabytes, and are perfect for microcontrollers with limited resources. Operating systems for mobile devices, such as Android and iOS, have become a class of their own, combining the interactivity of desktop systems with the resource-depleted environment and alternate interaction model that mobile devices have. These systems are optimized for energy efficiency, touch based interfaces, connectivity, and security based on the personal nature of mobile devices. The design picks a number of the classic trade-offs on mobile systems, including higher application isolation, per-application permissions and complex power management that to be definitively increases battery utilization by doing usage

allocation and lessening background activity. High-performance computing (HPC) operating systems cater to the specific requirements of supercomputers and large computing clusters utilized in scientific simulations, weather forecasting, genomic analysis, and various other computationally demanding applications. Advanced job scheduling for batch workloads, support for extreme parallelism across hundreds or thousands of processors and optimized communication facilities on the hardware level for tight-coupled parallel programs are built into systems such as Cray Linux Environment and IBM Parallel Environment. Since these systems are designed for machine workloads, and not man-computers use, instruments are applications that prioritize established computation throughput and efficient resource utilization over the interactive responsiveness. Exadata and Oracle RAC, as systems which combine traditional operating systems functionalities as implemented with specialized functionalities for data processing, storage management, and transaction management, use special purpose systems for data management and data interaction. Such systems employ advanced buffer management, query optimization, and concurrency control techniques tuned for data-specific workloads, often eschewing general purpose operating system facilities altogether to ensure higher performance through direct access to the hardware device. Network operating systems (like Cisco IOS, Juniper JUNOS, and VyOS) are used for running the network infrastructure equipment (like routers, switches, and firewalls). Such systems are designed to process packets at an extremely high throughput, manage traffic, and remain highly available under very high loads, often with some form of a real-time scheduler to ensure the network functions well even during peak demand times. Again, systems such as VMware ESXi, Microsoft Hyper-V, and Xen are another more specialized category providing the abstraction and multiplexing of the physical hardware that offers support for several guest operating systems on the same infrastructure. Hypervisor-based systems have advanced resource management, isolation, and emulation capabilities that provide the ability to run multiple different operating environments together on the same hardware bases. Operating systems that fall into safety-critical categories — which cover aerospace, medical devices, nuclear facilities, and automotive applications — typically use formal verification, redundancy, and fault-tolerance mechanisms that exceed

those found in consumer devices. INTEGRITY, LynxOS and PikeOS, for example, are designed for meeting stringent certification requirements such as DO-178C (airborne) or ISO 26262 (automotive) where the correctness of critical system components can often be proven mathematically. For example, security-controlled OSs -- such as SEL4 (with its formally verified microkernel), Qubes OS (with its threat model that emphatically prioritizes isolation) and Open BSD (which approaches secure defaults, and process separation) -- favor the maximization of attack surface as opposed to features or performance, making architectural decisions that systematically discard whole classes of threats. They prioritize the correct drawing and playback of multimedia content in real time according to parameters like scale and type via specialized scheduling and resource management, all while ensuring the sound and visuals remain in sync regardless of system load. Over these 50 years operating systems concepts have proven extremely adaptable over even very different environments and requirements as evidenced by the great variety of these specialized operating systems. Though the basic functions of process management, memory allocation, and I/O handling are universal, their wildly divergent design goals for their ecosystems lead to unique architectures crafted for specific use cases. The specialization trend continues to accelerate as computing is seeping into all manner of new domains, from wearables to smart home systems to autonomous vehicles to industrial IoT applications, each with its own set of unique requirements that influences how operating systems are designed. By studying these specialized systems, we gain valuable insights into the flexibility of OS principles and the powerful effects that design goals can have on the architecture of a system, lessons that we can apply hopefully to innovation even with more general-purpose computing.

1.3.42 Future Directions and Emerging Paradigms in Operating System Design

So continues the evolution of operating systems as we seek to broaden the scope of computing and face increasing complexity and challenges that test the limits of the designs we have known. A dozen or so trends are revolutionizing operating system design, fueled by hardware advances, evolving usage patterns, and pressing needs for security, efficiency, and adaptability in an interconnected world. Cloud computing, along with edge devices, is driving a sea change in the

architectural distinction of operating systems and how that functionality is spread across computing environments. Edge computing is blurring traditional delineations between local and remote execution, and is giving rise to new operating system paradigms that enable the seamless relocation of processes, data, and state from edge devices to cloud infrastructure (and vice versa) in response to dynamic conditions, resource availability, and application requirements. Such a distributed execution model calls for operating systems able to operate across heterogeneous hardware while maintaining coherent application state and security across trust boundaries. Operating system functions are increasingly powered by artificial intelligence, which allows for adaptive resource management and predictive optimization with autonomous operations that exceed static policies or heuristics. Machine learning or AI-based operating systems can offer benefits in areas such as pre-fetching and scheduling based on system usage patterns (more on this in the next section), optimizing power consumption for anticipated workloads, discovering anomalies from baseline usage patterns that may correlate with potential security hazards, and automatically re-tuning system parameters to maximize application performance as requirements change. The transition to self-tuning systems encapsulates a radical move away from the deterministic, rules-based systems that have defined operating system design for decades, for systems that improve themselves over time through usage. Architectural innovations that fundamentally rethink traditional operating system models are being driven by security and privacy concerns. Increased threats are pushing techniques such as capability-based security, formal verification of critical components, and hardware-enforced isolation from the research realm into the real world. Perimeter-based security models are being supplanted by zero-trust architectures that require every access request to be validated irrespective of its origin, and privacy-preserving computation methods such as homomorphic encryption and secure enclaves are now being built into operating system services. Security as a Fundamental Design Principle that Shapes Core Operating System Architecture These developments make a departure from security as an add-on feature to security as a fundamental design principle. The booming world of Internet of Things (IoT) devices is forcing innovation in lightweight operating systems that can run on limited hardware and that participate

in distributed applications that potentially involve hundreds or thousands of devices. This trend is evident in the various operating systems (OS) for embedded devices, such as RIOT, TinyOS, and Amazon FreeRTOS, which provide sophisticated functionality that is also highly resource-efficient. This includes new network protocols designed for low-power, low-bandwidth wireless communication; discovery mechanisms allowing battery-powered devices to efficiently find services; and security models that are both lightweight and suitable for unattended operation in the face of possible attacks. It also hints how containerization and micro services architectures are transforming application deployment models, with operating systems adapted to this model. Some third-party operating systems built specifically to host containerized applications include those from CoreOS (now owned by Red Hat), RancherOS and Google (Container-Optimized OS). They are designed with the bare minimum components required for what they do. This specialization trend is a return to purpose-built OSs, following decades of convergence onto general-purpose platforms, driven by virtualization technologies that allow many highly specialized systems to co-exist on shared infrastructure. The use of quantum processors, based on a range of principles that differ fundamentally from classical designs, poses possibly the most dramatic challenge to traditional designs for an operating system. New quantum operating systems face unique challenges such as qubit allocation, quantum error correction and the fusion of quantum and classical processing. Although functional quantum computers are still being developed, the operating systems used by these devices are likely to need completely new abstraction and resource management paradigms more similar to nature than to classical operating systems. The increasing focus on sustainability and energy efficiency is driving the operating system design from the mobile devices being designed to maximize battery life to data centers being designed to lower their carbon footprints. Energy-aware scheduling, dynamic voltage and frequency scaling across multiple cores, workload consolidation, intelligent resource hibernation, and other techniques are being developed into fundamental components of the operating system, rather than merely optional power-saving features. This move shows that more and more people are starting to understand that energy efficiency is not just an operational issue but the core design constraint

that should inform system architecture from the ground up. Operating systems innovations that minimize latency and provide consistent performance guarantees in haselwareeug applications are driven by real-time analytics and event processing requirements. The well-known batch-oriented paradigms are replaced by the stream processing model able to manage continuous data flows with predictable processing times, supported by operating systems functionalities designed to achieve such a behavior. This historical bifurcation has blurred, and both workloads need to coexist on systems that efficiently support both whilst maintaining isolation where required. Together these emerging paradigms imply a new era of radical innovation in operating system design, rivaling the paradigm shift from batch to interactive computing or the rise of distributed systems. With the ubiquity of computing, its increasing complexity, and its integration into essential infrastructure, operating systems should move beyond acting merely as resource managers of stand-alone platforms and instead become orchestration systems for heterogeneous sets of distributed computational resources that can self-adapt to novel operating conditions and needs. The operating systems of tomorrow are likely to be based on a greater degree of specialization (largely thanks to specialization in hardware and firmware as well) running on tightly-coupled interoperation; continuous self-optimization based on AI; active security models rather than passive ones; and design paradigms treating sustainability directly as a design goal rather than as a side consideration to performance, or reliability, etc. We aren't simply going to add to the existing space of operating systems; What these changes will do is change the nature of what an OS is and what is an OS to applications, to us to the outer environment. Operating systems is one of the few aspects of computer science that has tangential implications on almost everything; they are foundational systems that either enable or constrain what can be accomplished in computing, so it should come as no surprise that this field remains centrally located to many of the most exciting problems and opportunities in computer science today.

Summary

An operating system (OS) serves as the core software that manages computer hardware and software resources, providing an essential interface between users and machines. It enables users to interact with a computer system without needing to understand hardware-level

details. The introduction to operating systems outlines their historical evolution from early batch processing systems to advanced multitasking and multiuser systems like real-time, networked, and distributed OS. These developments reflect the growing complexity and versatility of modern computing needs, emphasizing the role of the OS in simplifying and streamlining user interaction with computers.

The need for operating systems arises from the demand for efficient and organized management of computing resources such as the CPU, memory, input/output devices, and storage. Operating systems handle key functions including process scheduling, memory allocation, file system management, device control, and security enforcement. They ensure that multiple applications and users can work smoothly and concurrently without conflicts. In terms of system operations, the OS controls how the CPU executes instructions, manages system interrupts, initiates the boot process, and maintains synchronization among processes. It facilitates system calls, manages hardware communication, and handles errors and recovery. Together, these functions illustrate how operating systems are indispensable for ensuring performance, reliability, and usability in modern computing environments.

Multiple-Choice Questions (MCQs)

1. Which of the following best defines an Operating System?
    a) A collection of programs that manage hardware resources
    b) A software used for document processing
    c) A hardware component of the computer
    d) A program used to browse the internet

    (Answer: a)
2. Which is NOT a function of an Operating System?
    a) Process management
    b) Memory management
    c) Compiling programming languages
    d) File system management

    (Answer: c)
3. What is the main purpose of system calls?
    a) To provide an interface between user programs and the OS

b) To execute application software
c) To compile programs
d) To manage network devices

(Answer: a)

4. Which type of OS executes jobs one at a time without user interaction?
   a) Multi-programming OS
   b) Time-sharing OS
   c) Batch processing OS
   d) Real-time OS

   (Answer: c)
5. Which of the following is an example of an Operating System service?
   a) File creation and deletion
   b) Providing direct access to hardware
   c) Executing JavaScript in web browsers
   d) Playing multimedia files

   (Answer: a)
6. Time-sharing operating systems are designed for:
   a) Running a single program at a time
   b) Providing fast response time to multiple users
   c) Executing batch jobs sequentially
   d) Eliminating multitasking

   (Answer: b)
7. Which system call is used to create a new process in Unix/Linux?
   a) exec()
   b) fork()
   c) open()
   d) exit()

   (Answer: b)
8. Which OS structure follows a hierarchical design with layers?
   a) Monolithic OS
   b) Layered OS
   c) Distributed OS
   d) Network OS

   (Answer: b)

9. Which design goal focuses on ensuring an OS remains operational despite failures?
    a) Security
    b) Portability
    c) Reliability
    d) Efficiency

    (Answer: c)
10. Which of the following is NOT an OS design goal?
    a) User convenience
    b) System security
    c) Hardware development
    d) Efficient resource allocation

    (Answer: c)

Short Questions

1. What is an Operating System, and why is it needed?
2. List three primary functions of an OS.
3. Define batch processing operating system.
4. What is time-sharing OS, and where is it used?
5. Explain the purpose of system calls.
6. What is the difference between multi-programming and multitasking?
7. Describe two key services provided by an OS.
8. What is the role of the kernel in an OS?
9. Explain the concept of monolithic vs. layered OS structures.
10. Why is security an important OS design goal?

Long Questions

1. Explain the need and functions of an operating system in detail.
2. Compare and contrast batch processing, multi-programming, and time-sharing OS.
3. Discuss the main services provided by an operating system.
4. Explain system calls with examples and their role in OS functionality.
5. Describe different operating system structures and their advantages.
6. How does the design of an OS affect its performance and usability?

7. Explain the importance of OS reliability, efficiency, and security in modern computing.
8. Discuss the role of the kernel and user space in OS architecture.
9. How does an OS manage process scheduling and memory allocation?
10. Explain different types of operating systems and their real-world applications.

MODULE 2
PROCESS MANAGEMENT AND SYNCHRONIZATION

LEARNING OUTCOMES

- To understand process concepts and states.
- To explore process control and operations.
- To analyze process scheduling and CPU scheduling algorithms.
- To study inter-process communication and synchronization techniques.
- To examine deadlock characterization and handling mechanisms.

2.1.1 Process Concepts

Aforementioned processes. times and speeding the instruction processing. Control Unit; The control unit (CU) is in charge of managing the instructions (pipelining) fetch and execute different instructions at the same time (in different stage of the cycle) significantly improves instruction throughput. Additionally, the processor architecture uses caching techniques to temporarily store frequently needed data in fast-access memory that is physically close to the CPU, dramatically reducing memory read system performance, and CPU designers have invested in improving the performance of individual stages of this cycle. Overlap of fetch-execute of multiple perform some complicated calculation.

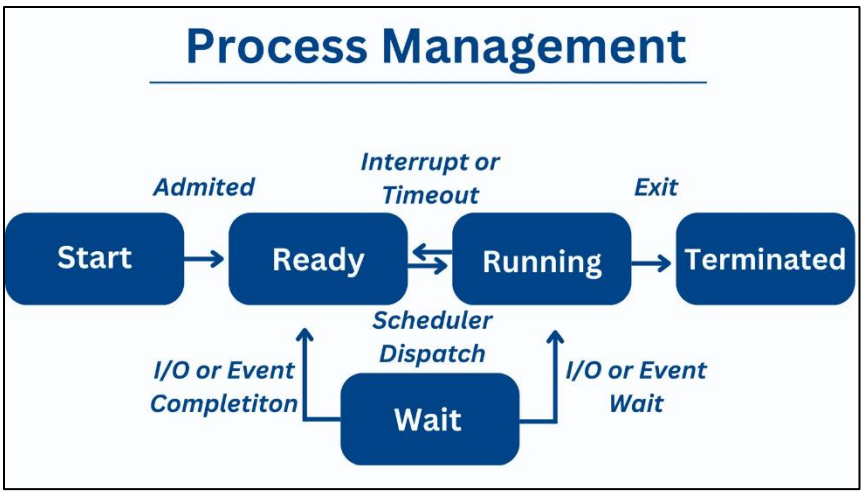


Figure 2.1.1: Process Management

Thus, this cycle efficiency directly dictates to is stored back to either memory or a register. And it continues infinitely in the same fashion, so the CPU can process a flow of instructions and an arithmetic/logic operation, data transfer, or change control flow. The final output of the execution the fetched instruction (opcode and operands). After decoding, the CPU executes the instruction – which could be (a register that keeps track of the instruction currently in execution). Once fetched, the instruction is decoded, in this step, the CPU identifies the operation to be done from loaded into memory of computer. The CPU starts by fetching an instruction from the memory; the address is determined by the program counter the fetch-decode-execute cycle, is the basic way CPUs do stuff. When a program begins execution, the

instructions it needs are that must be completed in order: fetch, decode, execute and write back. This cycle, called to understanding how computers operate at a fundamentally low level. At the core of how a CPU operates is the instruction execution process, which is a series of actions interplay of processes that allow it to carry out commands and handle information. Learning these processes are key The CPU's functionality is based on a complex Other running processes. not in RAM at the time. These mechanisms are essential to support multiple processes running concurrently without affecting the to process memory beyond the one physically accessible to it, extending the address space by treating secondary storage as an extension of RAM.

For Example: Imagine you open your music player app on your computer. The moment you double-click the app icon, the operating system creates a process for that program. This process includes the program's code, data, and resources like memory space and file handles. While the music is playing, the operating system schedules the process to run on the CPU, allowing it to read audio data from storage, decode it, and send the sound to your speakers. At the same time, other processes—such as your web browser or a document editor—are also running, each with its own isolated memory and resources. The OS manages these processes by rapidly switching between them, giving the illusion that everything is running simultaneously. If you minimize the music player or pause the song, the process doesn't end; it simply waits, ready to run again when needed. When you finally close the app, the operating system terminates the process, freeing up the memory and resources it was using.

As with execution of process, managing will be finite. These types of scheduling algorithms are key for fairness and responsiveness in a shortest execution time. With this, no single process can gain control of the CPU for too long as the CPU time allocated for a process to execute of resources and minimize wait times. This algorithm "First Come First Served" means that the processes are scheduled in the order they arrive, and the second is "Shortest Job Next" which decides based on the state, which is then loaded into the context of the next process that is going to be executed. The operating system scheduler uses algorithms such as First-Come, First-Served (FCFS), Shortest Job Next (SJN), and Round Robin to determine the order in which processes are executed in order to optimize the use by giving time slices to each process and doing

context switches extremely faster. This is the process of switching the context, which means the current process must save its of contemporary operating systems. The CPU does this involved in many concurrent processes with concurrent threads using scheduling algorithms and memory management. Multitasking, the ability to run many programs at once, is one of the pillars The core of the CPU is the instruction cycle but it is and exceptions are critical in their ability to allow systems to respond promptly to incoming events and exceptions. Exception handlers is not stored in memory, instead, the CPU accesses the interrupt descriptor table (IDT), which contains an entry representing the address of the handler for each interrupt or exception vector. Interrupts to an interrupt controller, which decides their priority and dispatches them to the CPU.

The location of interrupt and in keeping the system stable and preventing errors from cascading through the system. Interrupt requests are sent that takes proper actions to rectify the wrongdoing. Exceptions

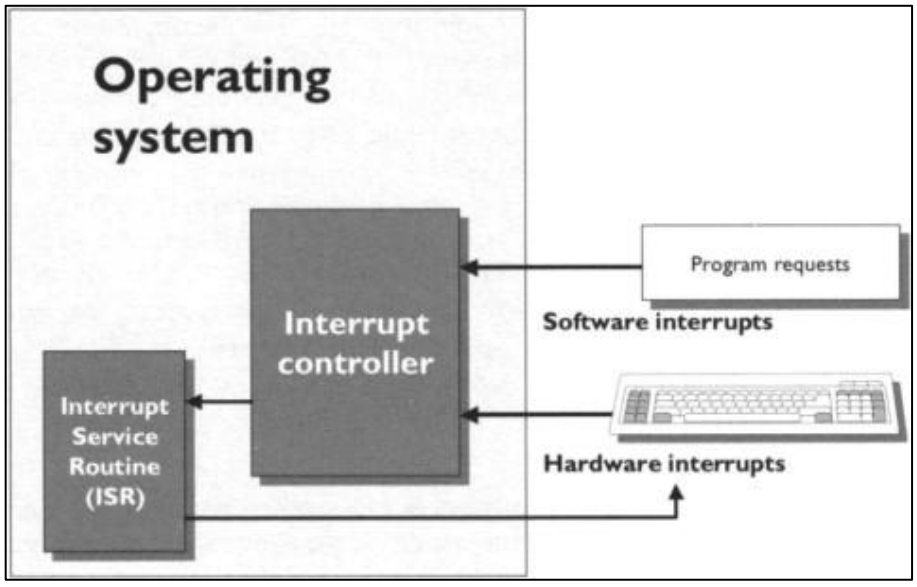


Figure 2.1.2: Interrupt In OS

play an important role event alerts emitted by the CPU itself when any errors or unusual conditions, such as division by zero, invalid memory access, or illegal instructions, occur. Similarly, when an exception gets raised, the CPU as well gives up on its current instructions and passes control to an exception handler happens, or, any hardware devices need to be read, cpu gets interrupted. Exceptions are the continues its previous execution after the interrupt has been handled. When data is needed from storage, or, a network event handler, a special routine set

up to deal with the interrupt. The CPU restores the saved state and the CPU. When an interrupt happens, the CPU pauses its current operations, saves the current state, and branches off to an interrupt can react to external events and error conditions. Hardware devices like keyboards, mice, and network interfaces generate signals to initiate an interrupt request to these are vital mechanisms any CPU must have in place so it and multicore design, enabling better performance and more complex task processing. In addition, CPU architecture has evolved to incorporate features such as parallelism tasks can be executed simultaneously through the use of multicore processors, which is crucial to meet the growing performance requirements of modern applications. and throughput. These topics allow you to understand how computational a common address space and resource pool, making them lighter than processes. Multithreading enables applications to execute multiple threads simultaneously (including background computations while responding), which improves responsiveness create multiple threads of execution. Threads have systems and allow performance gains for applications that can use parallel processing. Another technique that improves Parallelism is threading, allowing a process to have multiple cores, each of those cores can do things autonomously executing its own specific instructions and managing its own resources. Multicore processors are everywhere nowadays in computer threads/processes.

If you branch (conditional branch instruction) will go, in order to reduce the amount of penalty cycles that occur from branch instructions. This feature of Multicore processing integrates many on-chip CPU cores together, enabling simultaneous execution of numerous units in the CPU so the CPU can execute multiple instructions at the same time. Branch prediction is the process of guessing which way a executed at various stages simultaneously. This means that there is multiple execution a CPU core, which makes it possible for the processor to carry out several instructions at a time. Pipelining splits the instruction execution cycle into stages, so you can have several instructions being instructions (monads) simultaneously using one core or even across multi-cores. Techniques like pipelining, superscalar execution, and branch prediction enable instruction-level parallelism (ILP) within Parallelization means executing the multiple process is basically a program in execution, which includes the program code, the

current activity represented by the value of the program counter, and the contents of the processor's registers. an operating system schedules computational task. A active entities (representing running programs) that are owned, scheduled, and managed by the OS. Learning about the process state and CPU utilization is essential to understand how A modern operating system's core functionality is process management, where processes are processes. and it is taken out of the system. The process scheduler in the operating system takes care of this complexity by ensuring efficient usage of the CPU and equitable distribution of resources to competing to the ready state as well. Finally, when it completes its work or if it is terminated by a user or system, the process enters the 'terminated' or 'exit' state, during which its resources are freed the process returns to the 'ready' state until it gets a chance to use the CPU.

A process can also be preempted by the operating system, usually because it has run out of its time slice, or because a higher-priority process needs to run, sending the process back does not execute until

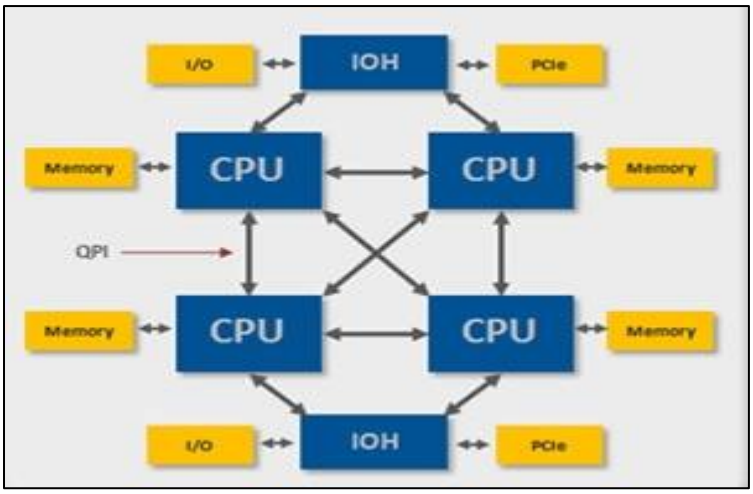


Figure 2.1.3: CPU in Multiprocessing Interface

the needed resource is released or the event it is waiting for gets finished. Next, once the condition is satisfied, process may give up the CPU voluntarily, for example, when it needs some input from user or to read from a file, it moves into the waiting or blocked state. While in this state the process not permanent. A this state, the queue executes its operations with an active CPU. But this running state is 'running'. In the CPU to become free. So the scheduler is an important part of an operating system, which picks one of the processes from the ready queue and allocates it the CPU which passes the state of the process

from 'ready' to process starts in the 'new' state, adopting a process life-cycle when it's creating or loading itself into memory. On successful creation, it moves into the ready state, indicating that it is ready to execute and is waiting for and other system resources.

A By evolving through a series of states throughout its lifecycle, this abstract entity indicates its relationship with the CPU Issues. and

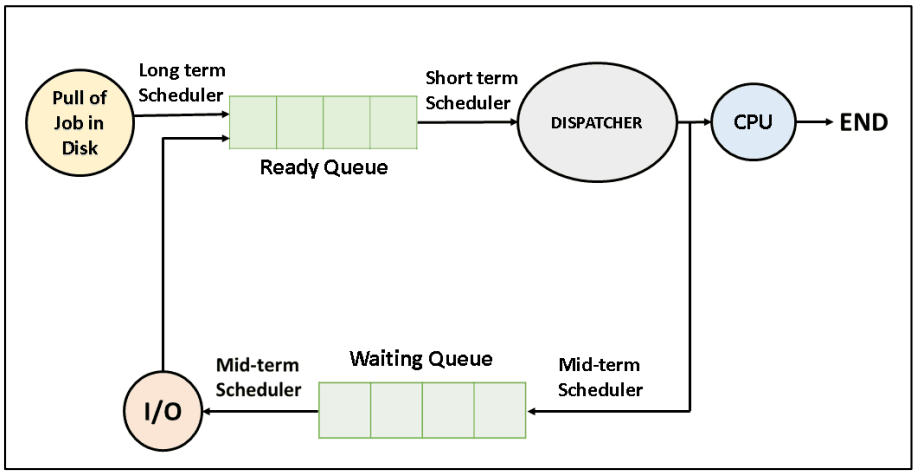


Figure 2.1.4: CPU Scheduling Mechanism

responsiveness. As an administrator, tools like your task manager and performance monitoring utilities give real-time insight into your CPU utilization, enabling rapid assessment and adjustment for performance low CPU usage indicates that the CPU is not being fully utilized, which might mean that certain resources are left idle, or that scheduling is inefficient. CPU Usage Basics CPU utilization is one of the most essential metrics for both system performance slowly spinning, may raises response time, indicates the CPU is greatly loaded. On the other hand, system loads and potential bottlenecks. Very high CPU usage, is by assigning them equally sized time slices. The operating system keeps track of CPU utilization using hardware timers and performance counters to provide information about Number) can be starvation. It ensures fairness among processes next (SJN): It is concerned with processes that have the shortest execution time. In Priority Scheduling, every process is assigned a priority number, and the CPU is allocated to the process with the highest priority number (Lowest for short processes. Shortest job Job Next (SJN), Priority Scheduling, Round Robin, etc. The simplest CPU scheduling algorithm is First-Come, First-Served (FCFS) which simply assigns the CPU to the processes

arriving first but can cause long wait times algorithms designed to maximise CPU usage with fairness and responsiveness. The common types of scheduling algorithms are First-Come, First-Served (FCFS), Shortest types of scheduling algorithms in the operating system to decide which process will get the CPU at a particular time. They are different gets CPU cycles. We use different utilization is the fraction of time the CPU is active doing non-idle work. CPU utilization refers to the percentage of time that a CPU is busy checking the state of processes, and when a process is in the running state, it figure for applications monitoring. CPU the central processing unit and the brain of your computer; it runs instructions and processes calculations. Its usage is an important The CPU is Fairness and responsiveness to make sure that every process gets a fairly different share of the CPU time. that is holding back for low priority processes for too long and it would be not executed in a crude manner." The scheduling algorithm should be designed such that it can balance between that important tasks are run quickly. But, if all the processes get assigned priority at all time, then that could lead to starvation; the scheduler. Generally, high-priority processes get the CPU a lot more than low-priority processes, so context switching is a key to reducing overhead while maximizing CPU usage. The OS also manages process priorities which can affect decisions made by data. Now, fast saves the state of one process and load the state of the second process. This process also comes with overhead in terms of the time taken to store and retrieve all register values, memory mappings, and process-specific and thus controls the selection process. But switching is a vital job of the scheduler and we are the one who CPU. The scheduling algorithm determines in what order processes are run, the state of becoming ready. When a process is forced to wait or made to relinquish the CPU, the scheduler takes another process from the ready queue to share the SJN are examples of non-preemptive scheduling algorithms, where process can run to completion w/o interruption and thus long wait times for other process. It also maintains the ready queue that connects all the processes in the utmost importance. FCFS and process, adding it to the in-wait queue and giving the CPU to another process so no single process owns the CPU for a long time. This is especially critical in interactive systems, where responsiveness is of long it has been executing for, and what resources the process needs, in order to make educated guesses on what

process should be allocated the CPU. For example, a preemptive scheduling algorithm like Round Robin or Priority Scheduling allows the operating system to suspend the execution of a an involved endeavor that must consider numerous conflicting goals regarding CPU utilization, waiting time, and fairness among competing processes.

Unit 2.2: Process State

2.2.1 Process State

The scheduler needs to also consider factors like process priority, how Process scheduling is in order to optimize performance and remain responsive. Through process system calls. It helps to understand the process state and CPU utilization and communication. When a program needs to interact with the operating system (e.g. to perform file I/O, memory allocation, or create a new process), it uses manage shared resources. You are using it to process management features followed by the Linux operating system, including process creation, termination, processes, ensuring data consistency and eliminating race conditions. There are such as semaphores, mutexes, and monitors to synchronize processes and process at previous executions. In addition, the scheduler is responsible for inter-process communication and synchronization: the coordination between several balancing is an important part of multi-processor scheduling when processes are evenly distributed among the available cores so that no core becomes a bottleneck. To improve performance and minimize cache misses, the scheduler must account for cache affinity, or the relative caching similarity of a processors so that the maximum parallelism can be achieved and consequently, the performance. Load process scheduling gets a little harder. The operating system has to spread processes over multiple cores or So with the modern-day multi-core and multi-processor systems, the CPU resource allocation, the operating system allows applications to run efficiently, ensuring a stable and reliable computing environment. Management and resulting in chaos and instability in the system. Switch In the absence of the PCB, the OS would not be able to distinguish between processes, processes; which resources are being used? This data structure is crucial for smooth multitasking as it allows the OS to switch between processes efficiently, a procedure called a context the OS's dossier on every running program, containing critical information that informs the operating system so it can correctly allocate and coordinate their execution. When a program is start, OS make matching PCB, which contains information about the state of the delicate art of multitasking, where multiple programs compete for the attention of the CPU, the PCB

serves as an individual identity card for each process, containing and preserving intrinsic and extrinsic data about the state of a process.

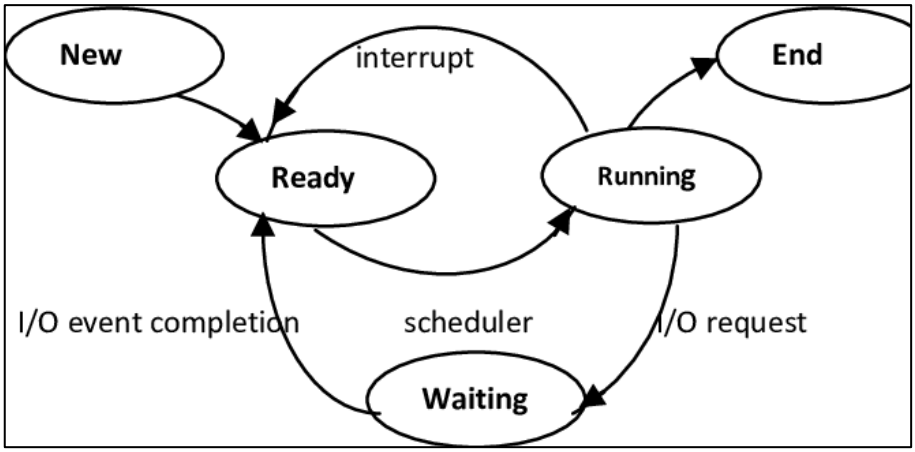


Figure 2.2.1: Process State Model
[Source - https://www.researchgate.net/]

Key Process States

A process typically moves through a series of states from its creation to its termination. The five primary states are:

- **New**: The process is being created and has not yet been loaded into the main memory. It's a nascent process that the OS is preparing.
- **Ready**: The process has been loaded into main memory and is ready to run. It's waiting for the CPU scheduler to allocate the CPU to it.
- **Running**: The process is currently being executed by the CPU. This is the only state where the process can actually perform its instructions.
- **Waiting** (or Blocked): The process has temporarily stopped its execution because it is waiting for some event to occur. This could be waiting for user input, waiting to access a file, or waiting for an I/O operation to complete.
- **Terminated**: The process has finished its execution. It is no longer active, but its process control block (PCB) might still exist to allow the operating system to collect its final status.

State Transitions

A process's state changes as it progresses and interacts with the operating system and hardware.

1. **New → Ready**: When a process is created, it moves from the **New** state to the **Ready** state, where it awaits its turn on the CPU.
2. **Ready → Running**: The CPU scheduler selects a process from the **Ready** queue and dispatches it to the CPU.

3. **Running → Waiting**: A process can voluntarily move from the **Running** state to the **Waiting** state if it needs to perform an I/O operation or wait for an event.
4. **Waiting → Ready**: Once the event the process was waiting for (e.g., I/O completion) occurs, it is moved back to the **Ready** state.
5. **Running → Ready**: This transition happens due to an interrupt. For example, a timer interrupt may occur, indicating that the process has exhausted its allocated time slice. The process is then preempted and put back into the **Ready** queue.
6. **Running → Terminated**: When a process completes its execution or is explicitly terminated by the operating system, it enters the **Terminated** state

Unit 2.3: Process Control Block

2.3.1 Process Control Block

In other words, its key structures used by an operating system is the Process Control Block (PCB), which serves as the main repository of information about the running processes. In the One of the processes is currently running, in a ready state, or waiting, or has been terminated. Multifaceted concept. The process state is a key component that signifies if a A PCB itself can do many, and like process management itself, is a of each process. can separate it from others. These data points are aggregated within the PCB, which allows the operating system full control and accounting info provides resource usage like CPU time and memory usage. Lastly, the process ID uniquely identifies itself among all the processes so the OS allocated CPU time. Status info tracks the allocated input/output devices for the process, tables and segment tables, control how the process accesses memory. Moreover, the process control block (PCB) stores scheduling information such as process priority and scheduling queues the OS uses to decide which process should be registers, along with the PCB, and preserves the computational state. Information about memory management facts, like page the context switch. Temporary data that is utilized by the process is stored in CPU A context switch involves saving the state of the currently running process, and restoring the state of the next scheduled process, the program counter is a critical part of that state that tells the CPU where to resume execution of the process after place and allowing it to quickly switch from one task to the next. of performance, and the PCB serves to minimize the overhead associated with this operation. PCB becomes the building block that makes the context switching process possible and less complicated by allowing the OS to store all the needed information in one allows the new process to continue where it left off the other process. For multitasking operating systems, the efficiency of switching contexts is a key determinant PCB into memory, which allows the OS to restore its saved state by copying the data stored in the PCB back into the CPU registers. These process other relevant data in the PCB of the process. Then, the OS loads the next process's running process. It does this by storing the CPU registers, program counter, and context switching. When the OS wants to switch from one process to another, it first has

to save the state of the currently The PCB is especially useful during from destabilizing the rest of the system. PCB's are stored in a protected area of memory that normal users cannot access. This prevents malicious (or inadvertent) data corruption importance. That's why it has a smooth and efficient computing environment. DONOTSPEC in PCB is of utmost share data and synchronize their actions. In short, the PCB acts as the OS's main mechanism for process management and control, which helps that higher-priority processes get more CPU time. In addition, it provides intercrosses communication as a means for processes to like memory, input/output devices to the Processes. The PCB also enforces process priorities and scheduling policies, ensuring for resource management and process synchronization. PCB contains all the information, which is used by OS to allocate and deallocate resources In addition to context switching, the PCB is also crucial

Figure2.3.1: Process Control Block
[Source - https://techspace.co.th]

2.3.2 Operations on Processes

We also explore how the operating system, as orchestrator, manages the processes (the unit of executing code) in the system, including the method of scheduling those processes. Helpful for tuning resource allocation and ensuring responsive UX. Processes are created and executed, suspended, resumed, and terminated over their lifecycle. Process creation is typically triggered by user input or software events and involves allocating the necessary resources, such as memory and file descriptors, and establishing the context in which the process will execute. The CPU needs to save relevant data so it can resume

execution from where it left off, such as the program counter, registers etc. The operating system keeps a data structure called the process table, which holds the information about every process, which allows it to manage state and a much greater detail about processes. The process states are — new, ready, running, waiting, terminated and they represent the various stages a process undergoes in its lifecycle. Processes transit between these states due to events like I/O requests, time-slice expirations, and process termination. That is, from the running state, the process maybe move to the waiting state whenever it asks for I/O, the when it finishes with the I/O, process get back to the ready state. Explanation: The IPC, or also known as inter-process communicational low communication and data transmission between multiple processes PIPE; In computing, a pipe is a mechanism for connecting the output of one process to the input of another. Graphically represented as a message queue is a queue of messages that can be read and written by different processes or threads. These are essential for cooperation, where one process needs to wait for another to finish a task or share data, and so on. The OS exposes system calls which the processes use to leverage these IPC mechanisms for controlled and secure communications. The last phase is Process termination, where all allocated resources are released and the process is removed from the process table. This returns system resources, making them available for use by other processes. The OS also needs to deal with unexpected terminations, like crashes or users force-quitting, to ensure system integrity. This means that the operating system needs to handle all processes, allowing them to run concurrently and ensuring a seamless user experience. CPU Scheduling is an OS function that chooses one of the ready processes to be allocated CPU at a given time.

Goal of CPU scheduling:

1. CPU utilization should be high
2. Throughput should be high
3. Turnaround time should be low
4. Waiting time should be low
5. Response time should be low
6. Fairness.

There are two types of scheduling algorithms Preemptive and Non-preemptive. In non-preemptive scheduling, when a process gets the

CPU, it holds it until it terminates or relinquishes the CPU by its own accord. First-Come, First-Served (FCFS) is a simple non-preemptive algorithm that handles the CPU to the process that arrives first. Additionally, FCFS is straightforward to implement but can suffer from the convoy effect, as a long process can block multiple smaller ones, producing a poor average waiting time. Shortest-Job-Next (SJN) ~ SJN is non-preemptive which selects the process with a minimize burst time. Shortest job next is an optimal algorithm for minimizing average waiting time, but it needs knowledge about future burst times, which is often impractical. On the other hand, preemptive scheduling permits the operating system to suspend a currently executing process and pass the CPU to another process. RR is a preemptive algorithm that gives each process a fixed time slice, or quantum. If a process fails to finish within its time quantum it is preempted and placed at the end of the ready queue. When RR can ensure a fair share of the CPU to all processes, small time slice can cause excessive context switching, leading to lower CPU efficiency. Shortest-Remaining-Time (SRT) is a preemptive implementation of SJN, which, at any point in time, chooses the process with the shortest remaining burst time. The Shortest Job First (SJF) algorithm, though has minimum average waiting time, it hinges on accurately predicting burst times and may lead to starvation for longer processes. We assign priority to each process in priority scheduling and allocate CPU to the process that reaches with the highest priority. Static or dynamic priority, preemptive or non-preemptive. Therefore, while preemptive priorities scheduling can preferentially run higher-priority processes, it cannot starve low-priority processes. This can be mitigated by using aging techniques wherein the priority of a process increases overtime. Multi-level queue scheduling It splits the ready queue into multiple queues. Processes are queued according to their properties, like whether they are foreground or background processes. The multi-level feedback queue scheduler is considered one of the most flexible and responsive process scheduling algorithms, as individual processes are able to be moved between queues based on their behavior.

In process management and CPU scheduling, context switching is an important operation performed by the operating system. Simply put, this is the process of saving the state of the currently executing process, and loading the state of the next process that needs to execute. This

includes information such as the program counter, registers, and memory management. When a process gets preempted, blocked, or terminated, or a new process is chosen to run, the operating system must switch context. Context switching is an essential function in operating systems, enabling multitasking by switching between processes, but it comes with an overhead. Both the time slices and scheduling algorithms affect how often context switching happens. In Round Robin Scheduling, a time slice that is too small would cause lots of context switches. Operating systems make use of context switching optimization techniques like keeping the context switching routine fast and utilizing hardware level support by having special registers that stores process states. Context switching must be efficient so that CPU resource usage stays high and the system responds quickly. The underlying OS must balance between the overhead of context switching to ensure fairness and responsiveness, and minimizing overhead to maximize CPU throughput. Scenarios such as real-time systems that must prioritize the timely execution of critical tasks are another case where efficient context switching is a necessity. Reducing the context switching latency can be more critical for these types of system to maintain the deadlines and improve the system stability. Modern operating systems utilize advanced techniques to enhance task-switching efficiency such as lazy context switching (only saving or restoring the context that is actually needed) or hardware-accelerated context switching that uses specialized hardware to speed up the process. The relationship between process operations and CPU scheduling forms the basis for the efficient operation of a computer system. You need to be able to do so with one of multiple processes using multiple operating systems. It is essential to utilize CPU scheduling algorithms that will enhance the efficiency of the system to meet the demands of the workload being processed. You also learn about the trade-off between different scheduling algorithms, and the effect of context switching overhead. Advancements in operating systems have resulted in advanced scheduling algorithms and process management approaches that can accommodate a wide range of workloads and system needs. Machine learning and artificial intelligence techniques will likely become an integral part of future operating systems for advanced process management and CPU scheduling. Hardware and software co-designs would keep

complementing each other leading to better performance and responsiveness and energy efficient systems. This cycle continues as operating systems strive to optimize both process operations and CPU utilization for efficiency and responsiveness, creating a robust environment for application execution and user engagement. Inter-Process Communication (IPC) is a fundamental concept in operating systems and is especially important in modern computing environments with concurrent processes. It ensures these processes can communicate and synchronize with each other, allowing them to exchange data, coordinate actions, and work together toward a common goal. Such inter-process interaction forms the foundation for developing sophisticated applications that tap into the strengths of multi-tasking and parallel processing. When it comes to CPU utilization, effective IPC mechanisms play a vital role in boosting system performance.

2.3.3 Inter-Process Communications

The idea of inter-process communication (IPC) is quite standard practice, but it can go wrong, and you will end up with bottlenecks, context switches, and overhead, which can slow down CPU performing tasks. On the other hand, well-designed IPC mechanisms support efficient data transfer and synchronization between processes, helping to keep the CPU busy and minimize inefficiency.

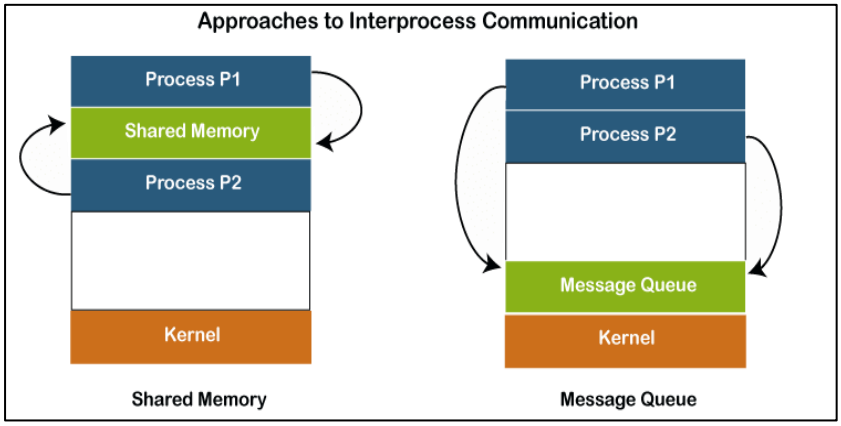


Figure2.3.2: Inter Process Communication

IPC covers various approaches, each with pros and cons, to cater to different communication requirements and system architectures. Some of these include shared memory, message passing, pipes, sockets, semaphores, etc. The ideal approach depends on the specific requirements of a program and the characteristics of its workload;

knowing the difference between these methods is important for maximizing a CPU's performance and maintaining the efficiency of multi-process systems. Shared memory, for example, provides fast communication by allowing processes to each access that same region of memory. But it requires synchronization, lest you corrupt your data. In contrast, message passing is a more structured form of communication, where processes send messages to one another via a communication channel. This approach comes in handy for distributed systems or when processes are running on different machines. Certain IPC mechanisms are chosen based on the communication latency, data size, and the complexity of synchronization needed. Through effective IPC, normal applications become more practical and system reliability and responsiveness are improved, all of which lead to higher CPU usage IPC is facilitated at a lower level by the CPU, which handles the hardware resources and executes the communication primitives associated with IPC. When processes communicate, whether via IPC, the CPU is responsible for transferring the data, synchronizing operations and ensuring that the communication protocol is addressed. For instance, in shared memory intercrosses communication, it is the CPU that must coordinate access to the shared memory region, enforce memory protection, etc. In Message Passing, the CPU is responsible for buffering messages and directing them to the destination process. The number of such operations is directly proportional to the performance of the IPC mechanism and hence utilization of the CPU. Context switching is a very important operation in a multi-processing environment that is invoked during IPC. When one process issues a communication request, such as a message sender or a shared memory access, it is possible that it will have to wait for another process to respond or release the resource. The CPU can then switch to another process in this waiting time so that it can do some usefully work. On the other hand, frequent context switching can introduce a performance overhead, as the CPU must save the state of the interrupted process and restore the state of the next process. Some effective IPC mechanisms improve the communication latency and minimize the synchronizing efforts that eventually results in less number of context switches during the message transfer. CPU also participates in IPC security and integrity enforcing. For example, memory protection mechanisms prevent unauthorized access to the shared memory regions, and thus

ensure that all active processes can access the data they have permission to access. For example, message authentication and encryption may be used to secure the confidentiality and integrity of messages exchanged between processes. These measures rely heavily on the security capabilities of the CPU to help build strong, secure IPCs. The CPU is responsible for a large part of the IPC process, as it provides the necessary hardware and software infrastructure that allows processes to efficiently and effectively communicate. This article explains how the CPU can be optimized by optimizing IPC mechanisms and minimizing overhead to improve multi-process

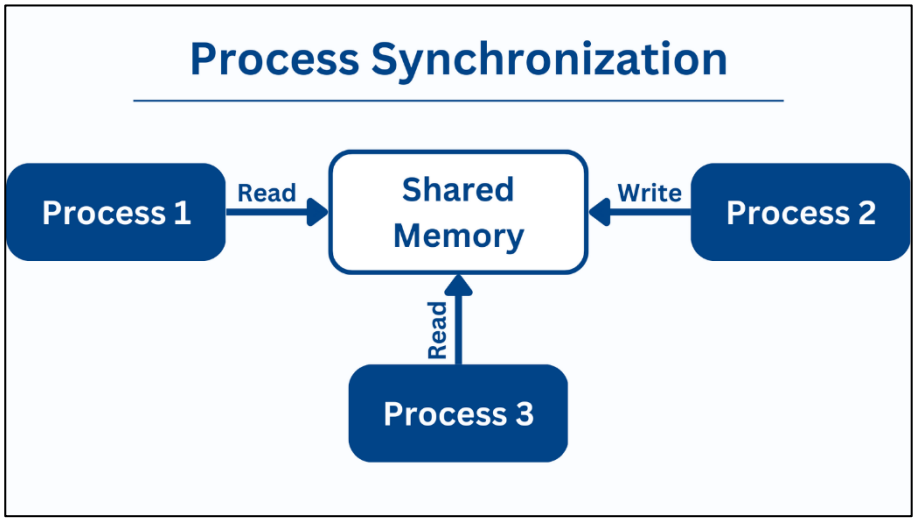


Figure 2.3.3: Process Synchronization in OS

applications.

Synchronization plays a crucial role in IPC, as it prevents conflicts between accessing shared resources which can lead to race conditions or data corruption. The CPU includes various synchronization mechanisms, including semaphores, mutexes, and condition variables, that processes can use to coordinate access to shared resources. For example, semaphores are commonly used to manage access to a limited number of resources, ensuring that multiple processes do not access the same resource at the same time. In contrast, mutexes offer mutual exclusion, preventing more than one process from entering a critical code section simultaneously. Data that can be used to signal that the shared resources state has changed. Q2: Why does the CPU need to execute these operations? If synchronization is too inefficient, deadlocks, livelocks, and other concurrency related problems would limit performance of the system. For example, when two or more

processes wait indefinitely for each other to release resources, it is called Deadlock. Livelocks happen when the processes in execution are constantly changing their state in response to each other, causing them to make no progress. It is using limited data and observing to detect these system stability problems and trigger the proper correction mechanism. The CPU architecture, beyond primitive synchronization, can directly impact synchronization performance. Hardware-level support enables lock-free synchronization techniques (e.g., atomic instructions such as compare-and-swap), which can greatly decrease contention (when multiple threads are competing for the same resource, causing some to wait for access) and reduce overhead under heavy contention in comparison to software-based locks. Modern CPUs have specialized instructions and cache coherence protocols to improve the performance of these atomic operations. This fusion between hardware and software is key to building highly perform ant and scalable concurrent applications. The CPU has a lot to do with synchronizing and handling interrupts, and you'd want to build up the ability for some signals as well. For example, interrupts might be used to inform processes about events or changes in the system, where signals could be used to facilitate inter-process communication. More advanced synchronization patterns, like event-driven programming or asynchronous communication, can be implemented using these mechanisms. Modern computing scenarios have introduced new CPU architectures and operating systems that have altered IPC considerably. These inter-process communication facilities have been fundamentally impacted by factors such as multi-core processors, distributed systems, and cloud computing environments. It becomes an even more urgent requirement with the increased number of cores per CPU. If you have a multi-core processor even that you have also multi-processes can run in parallel on different cores. Nevertheless, this level of parallelism brings its own problems relating to cache coherence, memory consistency, and synchronization. And, seeking to fill a gap, operating systems have delivered new IPC mechanisms that are tailor-made for the multi-core world: lock-free data structures, message queues that can be efficiently built on the underlying shared-memory architecture, etc. Distributed systems, where processes are on different machines connected over a network, would depend on IPC mechanisms that can handle network communication. Inter-process communication

across network barriers is often achieved in distributed environments using something like remote procedure calls (RPC) or message queuing systems. Dynamic resource allocation and the nature of virtualized infrastructure in cloud computing environments introduce unique challenges for IPC. Virtual machines (VMs) and containers, for example, add another layer of abstraction, which potentially affects communication latency and performance. Micro services Architecture: Cloud-native applications can be developed using micro services architecture, which means the applications are composed of small, independent services that communicate with each other using lightweight IPC (inter-process communication) mechanisms, such as REST APIs or message brokers. It is important for the CPU to be able to manage these different IPC types efficiently in order to create cloud applications which can scale and remain resilient. The advent of specialized hardware accelerators, e.g., GPUs, TPUs, etc., has further introduced new paradigms for parallel processing and IPC. These accelerators likely have different memory hierarchies as well as communication protocols, which creates a need for specific inter-process communication (IPC) strategies to effectively transmit data from CPU to the accelerator.

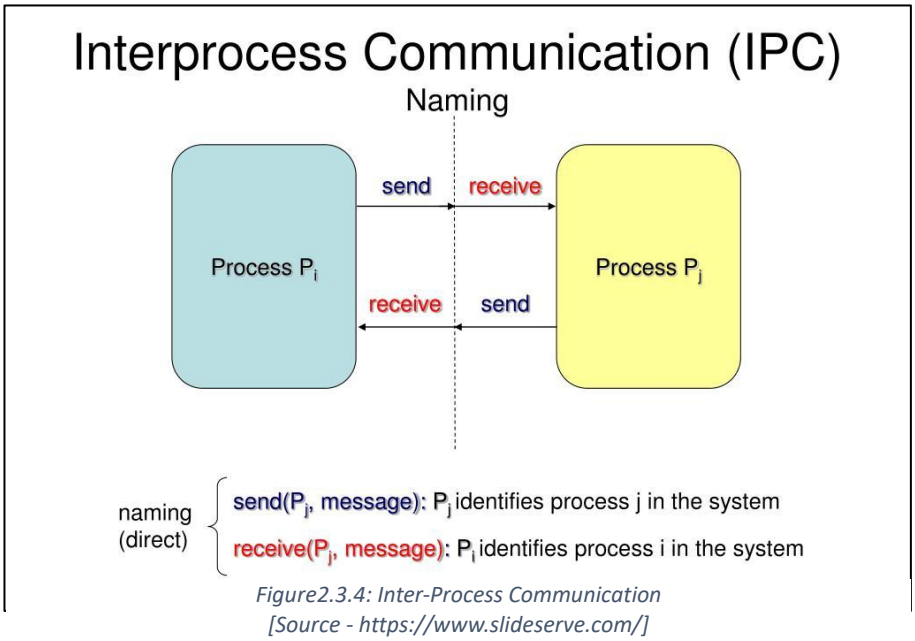


Figure2.3.4: Inter-Process Communication
[Source - https://www.slideserve.com/]

2.3.4 Foundations of Process Management and Communication

In the delicate ballet of an operating system, processes serve as the smallest entities of execution, an isolated instance of a program

contending for the computer resources. The efficient scheduling of these processes and the coordination of their interaction is the cornerstone for a working operating system. Now, at the core of this management is this thing called process scheduling, which is basically a mechanism that determines the order in which the processes are given access to the CPU. One CPU can only run one process at any instance, but many processes may be ready or waiting to run, and this indicates the need for process scheduling. New processes that need CPU time will need to be queued up with deciding algorithms that balance effective allocation of CPU time while preventing starvation for other processes. But before we dive into these algorithms, it is important to first understand how processes communicate with one another and coordinate their activities. IPC (Inter-Process Communication) enables processes to exchange data and synchronize their actions. This is especially important for complex applications where multiple processes handle various tasks to save resources and improve modularity. IPC methods are used to allow processes to work together and share resources in order to accomplish common tasks (such as shared memory, message passage and pipes). Had these communication pathways not been established, various processes would have been functioning in isolation, which would have prevented the evolution of complex and cooperative software systems. Sharing information and synchronizing execution is crucial for not only application functionality but also the efficient use of system resources. Examples include a print spooler process that communicates with application processes to receive print jobs, or a database server that coordinates with multiple client processes to handle data requests. And that's why IPC is a vital part of contemporary operating systems, allowing for the development of resilient and highly networked applications. Furthermore, the paradigm is also extended with the concept of process threads where a single process can run several threads at the same time. A thread is a small unit of process that may be addressed, which shares the same address space and resources of its parent process, allowing for more fine-grained parallelism and higher performance As such, this threading model is widely useful for any application that can be broken down into independent, smaller subtasks, such as web servers that can concurrently handle multiple client requests or multimedia applications that can process audio and video streams on separate threads at the

same time. However, the addition of threads brings new problems significantly around shared resources and preventing race conditions which takes us to the critical section problem.

2.3.5 Process Scheduling and CPU Scheduling Algorithms

In multitasking operating systems, process scheduling is the keystone of the system, making sure that the CPU is effectively utilized, and making sure that processes are run in a timely manner. The scheduler is part of the operating system that determines the next process that gets to run from the ready queue. The scheduling algorithm we choose has a great impact on the performance of the system such as throughput, turnaround time, waiting time and response time, etc. As such, different scheduling objectives and system requirements have led to the development of various CPU scheduling algorithms. First-Come, First-Served (FCFS) is the most basic algorithm you can have it executes processes in the order in which they enter the ready queue. FCFS is easy to implement but can cause variants of the convoy effect: a long process can block other, shorter processes, leading to large average waiting time. Selecting the process with the shortest burst time attempts to minimize average waiting time that is the goal of Shortest-Job-Next (SJN). However, knowing future burst times as SJN requires is often not feasible. Shortest-Remaining-Time (SRT) is a preemptive version of SJN where a shorter process can preempt the currently running process if its remaining burst time duration is less. In priority scheduling each process is assigned a priority, the scheduler selects the process with the highest priority. This algorithm can also be classified as preemptive or non-preemptive, and it enables the use of various scheduling policies based on process priority. However, the priority inversion problem when a low priority process blocks a high priority process can add more time as it cooks up counterproductive wait states. Round-Robin (RR) is a time-sharing algorithm in which, every process is assigned a fixed time quantum. In case a process has not completed its quantum, it will be pausing (or will be preempt) and the process that is at the front of the ready queue will start. RR is a little bit fairer in assigning CPU time but with relatively poor averages compared to the previous sorting algorithms, it is most appropriate for Interactive Systems and depends heavily on the values for the Time Quantum. Ready Queue Scheduling: Multilevel Queue Scheduling Multilevel

queue scheduled the ready queue into several individual queues. Processes are queued into these queues according to certain properties such as types of processes or based on priority. A slightly more complex scheduling algorithm is multilevel feedback queue scheduling, in which processes can move between the various queues based on their behavior (e.g. length of CPU burst or frequency of I/O burst). This allows for a highly adaptable and efficient scheduling system. Each of these algorithms has its own advantages and disadvantages, and the selection of algorithm is based on the different operating system requirement and corresponding workload. Therefore, you must grasp these algorithms to design and optimize operating systems capable of managing and executing numerous processes efficiently.

2.3.6 Process Threads and Their Significance

Process threads can be seen as a radical departure from the traditional model of process management, where separate processes operated in isolation from one another; they allowed for much greater parallelism and more efficient use of resources. A thread (or lightweight process) is the basic unit of CPU utilization. At the same time, unlike processes that have their own address space and resources, threads in the same process share the same code section, data section, and operating-system resources, such as open files and signals. Better yet, the model where threads share the same resource means they can communicate and collaborate more easily than separate process, since they just need to read/write directly a shared data instead of using any IPC mechanisms. User-Level and Kernel-Level Threads can be implemented either at user level or kernel level. User-level threads: are managed without kernel support by a thread library at the user level. This is a lightweight solution, but not useful when there are blocking system calls or usage of multiple CPUs. In contrast, kernel-level threads are handled by the OS kernel, which offers improved parallelism and blockage operations. But kernel level threads have relatively higher overhead as the kernel is also in charge of managing the threads. Multithreading is the concurrent execution of more than one sequence of instructions, or thread. It increases application responsiveness by allowing multiple threads to perform work in parallel, so the whole program isn't stuck doing one task. It enhances resource utilization by enabling threads to

share resources and run simultaneously on multiple CPUs. It simplifies the development of complex applications by enabling tasks to be broken down into smaller, independent threads. For example, a web server might spawn a separate thread to service each client request, allowing it to service multiple clients simultaneously. A multimedia application can decode audio and video streams in separate threads for smooth playback. But multithreading also comes with its own set of challenges, including shared resource management and data consistency. Race conditions which lead to unpredictable and erroneous results result when the outcome of a computation depends on the relative timing of threads executing in parallel. Synchronization: You are built with the ability to synchronize yourself. Mutexes, semaphores, and monitors are some of the synchronization methods used to control automatically synchronized thread execution and to safeguard shared resources. These mechanisms allow threads to safely access shared resources without causing issues such as data corruption or unexpected behavior. Has threads always been a part of Operating System design and implementation?

2.3.7 The Critical Section Problem and Synchronization

The critical section problem is the challenge faced by multiple processes/threads regarding the sharing of resources. Note that a critical section is a piece of code that accesses and modifies shared resources. If several processes or threads execute their critical sections at the same time, data inconsistency and race conditions may arise, resulting in incorrect and varying results. To avoid these problems, synchronization mechanisms are implemented to make sure that only one process/thread can access its critical section at a time. Protocols that solve the critical section problem must meet three requirements, namely mutual exclusion, progress, and bounded waiting. This means that only one process or thread is allowed to access the critical section at a time, a concept known as mutual exclusion. Note that if no process is in its critical section and some processes need to enter their critical sections, only those processes that are not in their remainder sections can take part in deciding which will enter its critical section next, and such selection cannot take place indefinitely. Bounded waiting makes sure that there is bound 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. To overcome the critical section problem several synchronization mechanisms have been introduced. Mutex (Mutual exclusion) locks are simple 2 state locks that can be acquired or released by a thread or a process. The mutex ensures that a single process (or thread) holds the mutex lock at a time, providing mutual exclusion. Semaphores are more general-purpose synchronization mechanisms that can be used to limit access to a given number of resources. Semaphores are integer variables that can only be accessed through two atomic operations: wait and signal. The wait operation: it decrements the semaphore value, and it blocks the process or thread when the value goes negative. The signal operation increases the value of the semaphore, and if the value is greater than or equal to 0, a blocked process or thread is released. The high-level synchronization constructs that encapsulate shared data and the operations that can be applied to that shared data are called monitors. It provides mutual exclusion by allowing only one process (or thread) into the monitor at any time. Condition Variables Condition variables are used to make a process or a thread wait until a specific condition occurs. You can access shared data between threads using constructs like Mutex, Atomic Int and other synchronization mechanisms. It is essential to correctly implement these mechanisms to avoid race conditions and ensure the correctness and reliability of concurrent systems.

2.3.8 Semaphores and Classical Problems of Synchronization

Synchronization, the coordination of multiple processes to ensure orderly execution and data integrity, is one of the fundamental challenges in operating systems. Semaphores are a classic synchronization data type in computer science, introduced by Edsger W. Dijkstra, and are an incredibly useful mechanism for regulating access to shared resources. A semaphore is an integer variable, the value of which is never negative, that, during initialization, is only accessed through two standard atomic operations: wait and signal. The wait operation, also known as P (it comes from a Dutch word "proberen", which means "to test"), is used to decrement the semaphore value. If the value is negative then the process that is executing wait is blocked until the semaphore value is non-negative. On the other hand, the signal operation (also referred to as V, from the

Dutch word "verhogen" which means "to increment") increases the semaphore value. If any processes were blocked on the semaphore, one is unblocked. There are two types of semaphores: binary semaphores, which may only have values 0 or 1, and counting semaphores, which allow any non-negative integer value. Mutual exclusion is commonly implemented using binary semaphores, so only one process has access to a critical section at a time. Counting semaphores, in contrast, control access to a limited number of resources. The original value gives you the total amount of available resources for this instance of your counting semaphore. Semaphores offer a general solution to different synchronization problems, in fact, the classical synchronization problems. The bounded-buffer problem, also referred to as the producer-consumer problem, describes a work environment with a fixed-size shared buffer, where producers make the items that are put in the buffer, and consumers take items from the buffer. The Semaphores make sure the producers don't insert an item into the full buffer and the consumers don't remove an item from the empty buffer. The readers-writers problem is a common synchronization problem that deals with concurrent access to a shared data set in which there are multiple readers and only one writer. Semaphores can also be implemented to ensure writers have exclusive access to the data set, and that readers do not access the data set while a writer is modifying the data set. The dining-philosophers problem consists of five philosophers seated around a circular table, each with a plate of spaghetti and two chopsticks. It takes both chopsticks to eat in a philosopher way. One way not to have a deadlock, which is where everyone is holding a chopstick and is waiting for the other, is to use semaphores. These are classical problems that illustrate the challenges of synchronization and the necessity of using the proper mechanisms, such as semaphores, to correctly and effectively operate concurrent systems. Though semaphores are powerful, they need to be used cautiously to prevent synchronization errors which could lead to deadlock and starvation, when processes are unable to proceed indefinitely.

2.3.9 Deadlock Characterization

Deadlock in concurrent systems is a scheduling problem that occurs when two or more process are blocked forever, each holding a resource and waiting for another resource held by another process in the cycle.

Before designing a deadlock handling mechanism, it is important to know the features of deadlock. A deadlock can only occur under four necessary conditions, which must hold (at the same time): mutual exclusion, hold and wait, no preemption, circular wait. Mutual exclusion means that resources are non-shareable i.e. only one process can use a resource at a time. Hold and Wait: A process holding at least one resource is requesting additional resources held by other processes. In a no preemption scenario, resources cannot be forcefully taken away from a process; they need to be released voluntarily by the process holding them. Circular wait → We are having a set of waiting process {P0, P1,..., Pn} such that P0 is waiting for a resource hold by P1, P1 is waiting for a resource hold by P2,..., Pn is waiting for a resource hold by P0. All four of these conditions cause the processes to hang and leave a wait, where the processes never move forward, leaving the whole system as a standstill. Resource-allocation graphs: These are very useful to both visualize and to analyze deadlock. A resource-allocation graph G is defined by a set of verticesV, and a set of edges E. The vertices are partitioned into two types, P = { P1, P2,..n }, the set of processes currently active in the system, and R = { R1, R2,…m }, the set of resource types in the system. If we say that there is a directed edge from process Pi to resource Rj, written Pi → Rj, this means that process Pi has requested one instance of resource type Rj. Here, an edge from resource Rj to process Pi, Rj → Pi, indicates that a resource of type Rj was allocated to process Pi. If a cycle exists in the resource-allocation graph, there is a possibility of deadlock. If there is only one instance of each resource type, then a cycle indicates that a deadlock has occurred. If there are multiple instances of each resource type, then a cycle does not necessarily indicate a deadlock. This means you have to do additional work to see if there is a deadlock, in this case. The deadlock characterization gives a technique to reason about the scenarios that can lead to deadlock, and how to prevent, avoid, detect, and recover from deadlock. By acknowledging the required conditions and applying mechanisms such as resource-allocation graphs, system architects can develop resilient strategies to avoid the threat of deadlock and preserve the reliability and responsiveness of concurrent processor systems.

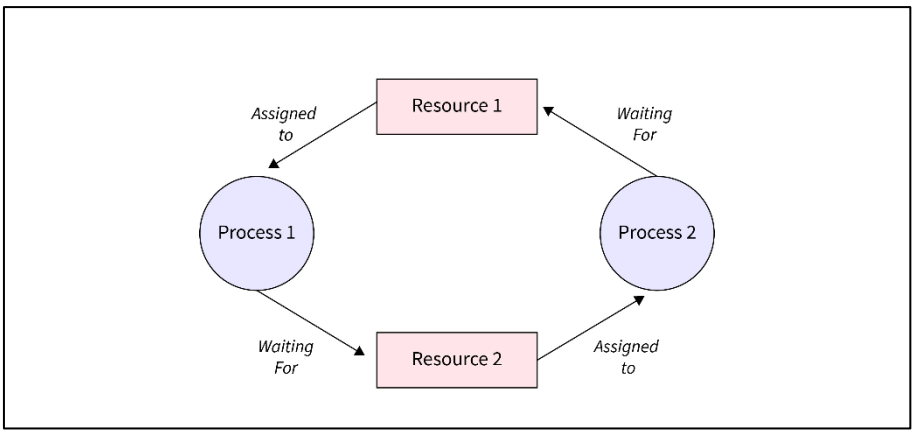


Figure 2.3.5: Deadlock
[Source - https://www.scaler.com]

2.3.10 Deadlock Handling: Avoidance

Deadlock avoidance is the appropriate technique of eliminating deadlock when the program executes, which ensures that the system will not enter a deadlock state. Ithence requires the operating system to know upfront the maximal resource needs of each process. It then checks request on resources to see if doing so will cause deadlock. The Banker's algorithm is a well-known deadlock avoidance algorithm, which is inspired by a banker who grants loans to customers. The Banker's algorithm needs every process to specify its maximum needs in advance. In the operating system, information about available resources, resources allocated to processes and maximum resources required by processes are maintained. When a process requests a resource, the system simulates the allocation to check if the resulting state is safe. If there exists some order in which the remaining resources can be allocated to each process then that state is called as safe state. The resource is allocated if the resulting state is safe; else the process has to wait. It is the Banker's algorithm which makes sure that system always remains in safe state and there is no deadlock. But it also has its limitations. It requires providing a declaration of maximum resource request size in advance for each process, which is not always possible. Computation can also be expensive since it requires the operating system to run complex calculations to determine if each state is safe. A different method of avoiding deadlock is the resource-allocation graph algorithm. This algorithm is when there's only one instance of each resource type. The system uses a resource-allocation graph and checks it for cycles before allocating resources. The resource is not allocated if allocating a resource will create a cycle.

This algorithm is simpler than the Banker's algorithm, but it can only be applied on single-instance resource types. Deadlock avoidance methods are helpful to prevent deadlock but they incur an overhead and not all resources can grow as per the demand. Deadlock avoidance mechanisms in systems require careful consideration of the trade-offs between deadlock prevention and resource usage.

Deadlock Handling:

Is a reactive approach for deadlock management, allows the system to enter a deadlock state and detects and recovers from it. The second approach is used by systems where it's not possible to avoid deadlocks due to the overhead in maintaining information about the resource needs and the lack of any advance information about the resource needs. Periodic Checking for Deadlock; in this scheme, we check the system for deadlock periodically. A popular technique is to utilize a resource-allocation graph and look for cycles. A deadlock is detected if a cycle is found. Another approach is the wait-for graph, a modification of the resource-allocation graph that focuses on the waiting relationships between processes. A wait-for graph has vertices as processes and edges as waiting relationships. An edge from process Pi to process Pj indicates that Pi is waiting for a resource that is being held by Pj. The cycles in the wait-for graph are a deadlock. After the deadlock is detected, the system needs to get out of that state. Many recovery methods can be applied. One approach is to kill all processes involved in the deadlock. While this is a very straightforward way to do this, it can lead to a lot of work being lost. Another approach is to kill one process at a time until the deadlock is broken. Based on like priority, resource consumption, and the amount of work completed, have a process chosen which will be aborted. A second recovery strategy is preempting resources. This means stealing resources from one process and giving them to another. You need to be careful not to starve in this approach when a process is being preempted so many times and it never reaches completion with its execution. The selection of recovery mechanism depends on various aspects of the system and the trade-off between performance and resource consumption. Deadlock detection and recovery are flexible methods for managing deadlocks, but they can incur overhead and cause work to be lost. Designers of systems that need to support semantics like deadlock

detection and recovery must evaluate these trade-offs against the rest of their system requirements.

Deadlock Handling: Prevention

Deadlock prevention is a prevention-based scheme, this scheme tries to remove one or more of the four necessary conditions for deadlock. If the system can prevent those conditions from ever occurring, then deadlock will never happen. To prevent deadlock that happens, mutual exclusion should be removed. One way to do this is to make resources shareable. Some resources, like printers and tape drives, are inherently non-shareable, however. An alternative is to remove the hold and wait. This is done by requiring processes to request all of their resources at once before they begin execution, or by requiring that processes release all of their resources before requesting more.

Summary

A process in an operating system is an active instance of a program in execution, representing a fundamental unit of work within a system. Unlike a program, which is a passive set of instructions, a process includes the program code and its current activity, such as the value of the program counter, contents of the processor's registers, and the variables in use. Each process operates within its own context and requires system resources like CPU time, memory, files, and I/O devices. Processes are created, scheduled, executed, and terminated by the operating system, which ensures proper synchronization and coordination among multiple processes running concurrently.

The state of a process reflects its current activity and can change as the process executes. Typical process states include new (being created), ready (waiting to be assigned to a processor), running (currently being executed), waiting (waiting for some event like I/O completion), and terminated (finished execution). These transitions are managed by the operating system using a data structure known as the Process Control Block (PCB). The PCB contains crucial information about each process, such as process ID, current state, CPU registers, memory limits, accounting information, and I/O status. It acts as a snapshot of the process at any given time and is essential for process switching, as it allows the OS to save the state of one process and load the state of another seamlessly, enabling multitasking and efficient resource utilization.

Multiple-Choice Questions (MCQs)

1. Which of the following is NOT a valid process state?
    a) New
    b) Running
    c) Terminated
    d) Scheduled

    (Answer: d)
2. The Process Control Block (PCB) contains which of the following information?
    a) Process state
    b) Program counter
    c) CPU scheduling information
    d) All of the above

    (Answer: d)
3. Which operation creates a new process in an operating system?
    a) Terminate
    b) Fork
    c) Kill
    d) Swap

    (Answer: b)
4. Inter-process communication (IPC) allows:
    a) Processes to share data and synchronize actions
    b) A single process to run multiple times
    c) The CPU to execute only one process at a time
    d) A process to execute in kernel mode only

    (Answer: a)
5. Which CPU scheduling algorithm selects the process with the shortest burst time first?
    a) First-Come, First-Served (FCFS)
    b) Shortest Job Next (SJN)
    c) Round Robin (RR)
    d) Priority Scheduling

    (Answer: b)
6. Which of the following is NOT a characteristic of a thread?
    a) Shares the same address space with other threads in the same process
    b) Requires more resources than a process
    c) Can run independently within a process
    d) Improves program efficiency and responsiveness

(Answer: b)

7. Which of the following synchronization problems occurs when multiple processes access shared resources incorrectly?
    a) Thrashing
    b) Critical Section Problem
    c) Page Fault
    d) Fragmentation

    (Answer: b)
8. What is the role of semaphores in process synchronization?
    a) They eliminate the need for process scheduling
    b) They prevent deadlock conditions completely
    c) They help control access to shared resources
    d) They replace CPU scheduling algorithms

    (Answer: c)
9. Which of the following is NOT a classical problem of synchronization?
    a) Producer-Consumer Problem
    b) Readers-Writers Problem
    c) Dining Philosophers Problem
    d) Page Replacement Problem

    (Answer: d)
10. Deadlock occurs when:
    a) A process is forced to terminate by the OS
    b) Multiple processes are waiting indefinitely for resources held by each other
    c) CPU scheduling fails to work
    d) All processes finish execution successfully

    (Answer: b)

Short Questions

1. Define a process in an operating system.
2. List the different process states and explain any two.
3. What is a Process Control Block (PCB)?
4. Name two operations on processes and explain their purpose.
5. What is Inter-Process Communication (IPC), and why is it important?
6. List and briefly explain any two CPU scheduling algorithms.
7. What is a thread, and how does it differ from a process?
8. Define the Critical Section Problem in process synchronization.

9. What is a semaphore, and how does it help in synchronization?
10. Explain the concept of deadlock avoidance in process management.

Long Questions

1. Explain the concept of a process and describe the different process states with a state transition diagram.
2. What is a Process Control Block (PCB)? Discuss its components and significance in OS.
3. Discuss the different operations on processes, including process creation and termination.
4. What is Inter-Process Communication (IPC)? Explain message passing and shared memory as IPC mechanisms.
5. Compare and contrast different CPU scheduling algorithms with their advantages and disadvantages.
6. Explain the concept of process threads and the benefits of using multithreading in an OS.
7. Discuss the Critical Section Problem and the different solutions used to resolve it.
8. What is a semaphore, and how does it help in process synchronization? Provide an example.
9. Explain the different strategies for handling deadlocks, including avoidance, detection, and prevention.
10. Describe the Dining Philosophers Problem and propose a solution using semaphores.

MODULE 3
STORAGE MANAGEMENT

LEARNING OUTCOMES

- To understand memory allocation techniques and paging.
- To study virtual memory concepts and page replacement algorithms.
- To analyze file systems, access methods, and their implementations.
- To explore free space management in file systems.

Unit 3.1: Contiguous Memory Allocation

3.3.1 Contiguous Memory Allocation

One key aspect of operating system function is memory management, which begins with the simplest option, contiguous memory allocation. Though deceptively simple, this technique sets the stage for understanding more complex ones. Contiguous memory allocation = All the data of a process is allocated in a single block. A process that is executed must also have a memory laid out for its code and data. In some ways it simplifies memory management for the operating system because it only needs to track one starting address and a size for each process's memory section. While this leads to benefits, it also presents some major challenges; especially as far as memory fragmentation goes. Consider a system with a fixed partition scheme (with pre-defined number of fixed partition sizes). When a process arrives, it is assigned to the smallest available partition that is large enough to hold it. Hence proved external fragmentation while allocating memory using this algorithm; where allocation takes little time and hits on memory. External fragmentation when total free memory is enough for a process's request but is not contiguous; So for example, after many processes have been loaded and exited, free memory may hold many small isolated blocks. Large process cannot get loaded even though the total free memory is large enough, as no single free chunk is big enough. & Variable partition schemes (try to address this by allowing partitions to be created dynamically as per process size. When a process loads, a current partition of the exact size is assigned. This is because it minimizes internal fragmentation, which is created when the allocated partition for a process is larger than the actual size, thereby wasting space that belongs to that partition. But variable partitions add to external fragmentation. When processes are loaded and ended, memory gradually becomes more fragmented and memory usage is less efficient. This compaction is a solution for external fragmentation; it moves various processes in memory, to make the free space in memory become a continuous block. Although effective, compaction is neither cheap operation as it involves relocating processes and updating their memory addresses. The cost of compaction could greatly affect the throughput of the system, although it potentially only occurs at large objects in systems where processes arrive and leave frequently

as described in thin provisioning. Although mathematically straightforward, successive contiguity is burdened with issues of fragmentation that more pliant
and effective strategies for memory management would tackle.

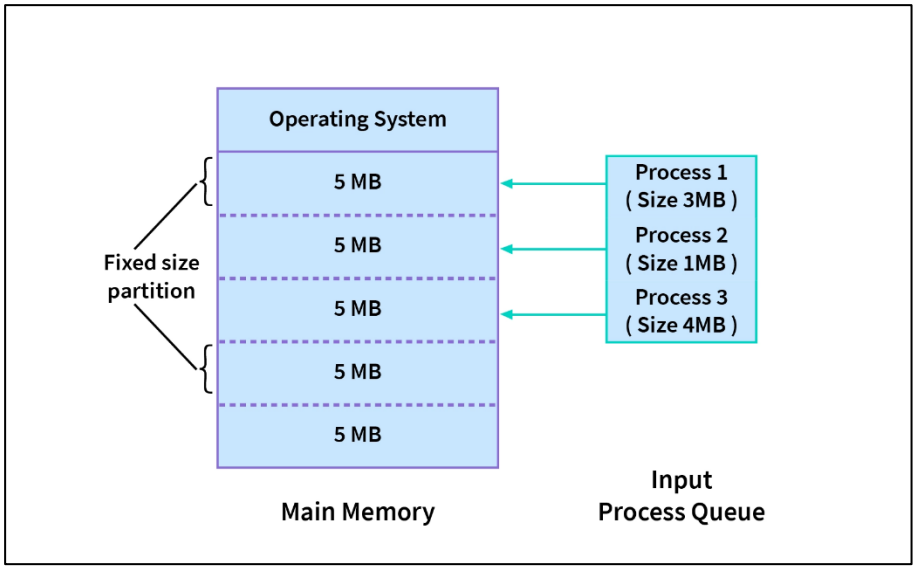


Figure 3.1.13:Contiguous Memory Allocation

[Source - https://www.scaler.com]

Key Concepts and Challenges

- **Fixed Partition Scheme**: Memory is divided into a set number of fixed-size partitions. A process is placed in the smallest partition that can fit it. This can lead to **internal fragmentation**, where the allocated partition is larger than the process, wasting space within the partition. More importantly, it can also cause **external fragmentation** because the total free space might be large enough for a process, but it's scattered in small, non-contiguous blocks.
- **Variable Partition Scheme**: This approach tries to reduce internal fragmentation by dynamically creating partitions that are exactly the size of the process. However, this method worsens **external fragmentation**. As processes are loaded and unloaded, the free memory becomes broken into many small, unusable chunks.
- **External Fragmentation**: This occurs when there is enough total free memory to satisfy a request, but it's not in one contiguous block. This problem is a major drawback of contiguous allocation schemes.

- **Compaction**: This is a solution to external fragmentation. It involves moving all allocated processes in memory to consolidate the free space into a single, large, continuous block. While effective, compaction is a resource-intensive operation that can impact system performance and throughput.

Unit 3.2: Paging Techniques

3.2.1 Paging Techniques: Swapping, Paging, Segmentation, Fragmentation

In order to improve when it comes to contiguous memory allocation, operating systems started to use more complex methods, such as swapping, paging, and segmentation. Swapping refers to memory management process in which a process is moved from main memory (RAM) to secondary storage (disk) and vice versa. When the principles of working are full then one of the inactive processes or processes with a low priority are transferred to the disk using Operating System and it will free memory for other processes. When it is again needed, the swapped-out process is brought back into main memory. Swapping is when the memory used by a running process is written to the disk, to free up RAM and reduce overall memory consumption, if the total memory requirements of the running processes exceed the available memory in RAM. This, however, incurs considerable overhead, as moving processes back and forth from memory to disk takes a non-negligible amount of time when compared with switching between processes that are in memory. Paging, a more complex technique, solves the problems of fragmentation that contiguous allocation has many. The paging mechanism divides physical memory (Ram) as well as logical memory (process address space) into fixed-size blocks, namely frames (for physical memory) and pages (for logical memory). The size of a frame is called the page size, which is usually from 4-8KB. The pages of a process are placed into the free frames in memory when the process is loaded. More specifically, the OS maintains a page table for every process, which translates the logical pages used by the process to the physical frames in which those logical pages are stored. This enables a process's pages to be not consecutive in physical memory, thus avoiding the issue of external fragmentation. Yet, paging complicates internal fragmentation because the last page of a process may not be fully used. Again, segmentation is another memory management technique by which the logical address space of a process is divided into a number of segments. Paging divides memory into fixed-size pages, while segmentation allows variable-length segments. Each process has a segment table maintained by the OS that maps base address and limit (size) of the segment. Segmentation has the benefit of

storing memory in a logical structure since the segments are related to logical units of the program. They do suffer from external fragmentation though, as segments can be of different length, leading to gaps in physical memory. Fragmentation is the general term for wasting memory, and it is a common problem in managing memory. Paging has another drawback named as Internal Fragmentation since allocated memory is greater than the required memory. In contiguous allocation and segmentation, external fragmentation refers to the condition of having enough total free memory, but it is spread throughout the system in small blocks. Solving fragmentation is an important aspect of memory management optimization and advancement of the system performance. Many contemporary operating systems implement a combination of paging and segmentation in order to gain the benefits of both techniques while minimizing their disadvantages. And, for instance, segmented paging combines logical segmentation with fixed-size allocation (which of course gives the best of both worlds).

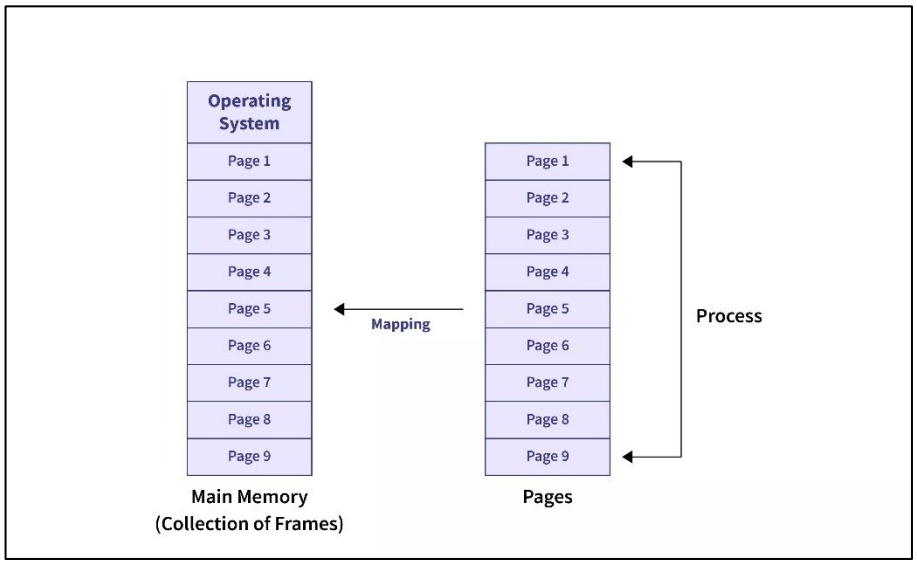


Figure 3.2.1: Paging
[Source - https://www.scaler.com]

1. Swapping

Swapping is a memory management technique that temporarily moves a process from main memory to secondary storage (like a hard disk) and then brings it back into memory later. This is done to free up RAM for other processes when the main memory is full.

How it Works:

1. The OS identifies a process to be "swapped out." This is typically an inactive or low-priority process.

2. The entire process's memory image is written to the disk.
3. The freed memory can now be used by another process.
4. When the swapped-out process needs to run again, its memory image is read back from the disk and loaded into a free block of RAM.

Drawback: This method can be slow due to the high latency of disk I/O, which is much slower than RAM access.

2. Paging

Paging is a non-contiguous memory allocation technique that solves the problem of external fragmentation. It divides a process's logical address space into fixed-size blocks called pages, and the physical memory into equally sized blocks called frames.

How it Works:

1. When a process is loaded into memory, its pages can be placed into any available frames in physical memory.
2. The pages do not need to be contiguous (next to each other) in RAM.
3. The operating system maintains a page table for each process, which maps its logical pages to their corresponding physical frames.

Benefit: This eliminates external fragmentation, as a process's memory can be spread across multiple free frames.

Drawback: It can lead to internal fragmentation, where the last page of a process may not be completely filled, leaving some wasted space within the allocated frame.

3. Segmentation

Segmentation is a memory management technique that divides a process's logical address space into variable-sized blocks called segments. Unlike paging, segments are not fixed in size and are often used to represent logical units of a program, such as the code segment, data segment, or stack.

How it Works:

1. The OS maintains a segment table for each process. This table stores the base address and the length of each segment in physical memory.
2. When a program is executed, its segments are loaded into non-contiguous blocks of physical memory.

Benefit: It provides a more logical view of memory from a programmer's perspective.

Drawback: Because segments are of variable sizes, this technique can lead to external fragmentation, as free memory can become broken into small, unusable chunks.

4. Fragmentation is a general term for the inefficient use or waste of memory. There are two main types:

- **Internal Fragmentation**: This occurs when an allocated block of memory is larger than the size of the data or process that needs to be stored. The unused space within the allocated block is wasted. Paging is a technique that can lead to internal fragmentation.
- **External Fragmentation**: This happens when the total available free memory is sufficient to satisfy a memory request, but it is not in a single, contiguous block. Instead, it is scattered in many small, non-contiguous chunks. Contiguous memory allocation and segmentation are techniques that can suffer from external fragmentation.

Unit 3.3: Demand Paging

3.3.1 Demand Paging

It is a virtual memory management concept that allows a page to be loaded into a virtual memory only when the page is needed. In classical paging, all the pages of a process are brought into memory immediately after the process is first invoked, even if some of these pages were not used. Demand paging, however, loads pages on demand, i.e., a page is loaded only when the process tries to access it. In this way, the memory that the process only actively uses is in RAM, thus drastically cutting down the required memory for a process to run. A page fault happens when a process tries to access a page that is not currently in memory. The operating system responds to the page fault by bringing in the missing page from secondary storage (disk) and placing it into a free frame in physical memory. The operating system keeps track of which pages are valid (in memory) and which are invalid (not in memory) by using a valid/invalid bit for each page in the page table. When a page is loaded, valid bit is set to 1 else its set to 0. The operating system will select one page currently in memory and evict it in order to bring in the page that caused the page fault. In this case, whichever page replacement algorithm decides to replace a certain page. Page replacement algorithms include Common Page Replacement Algorithms (First In, First Out · Least Recently Used · Optimal). FIFO: Replace the oldest page in memory; LRU: Replace the page not used for the longest time. Optimal replaces the page not going to be used for the farthest future time, but it is not practical to implement because it requires future knowledge. The performance of demand paging is greatly influenced by the page replacement algorithm used. A good algorithm should try to minimize page faults, so as to reduce the overhead involved with disk I/O. Thrashing happens when a process is executing so fast that it spends more time paging than running. A page fault occurs when the number of pages that are kept in memory at a time is less than the working set for a process (i.e. the set of pages that a process is actively using). This poses a serious problem, though; when thrashing occurs, the system is busy thrashing pages and the CPU is waiting for page loads from disk more time than it is spending in user space. The only solution left to avoid thrashing is for the operating system to provide each process with

enough frames to hold its working set. That was where working set models come in handy to figure out the working set of a process and the correct amount of frames to assign. In summary, demand paging is an efficient memory management technique that allows for greater flexibility in program execution and optimal memory usage. It is a key feature of contemporary virtual memory architectures, allowing optimal utilization of hardware resources and improving overall system efficiency.

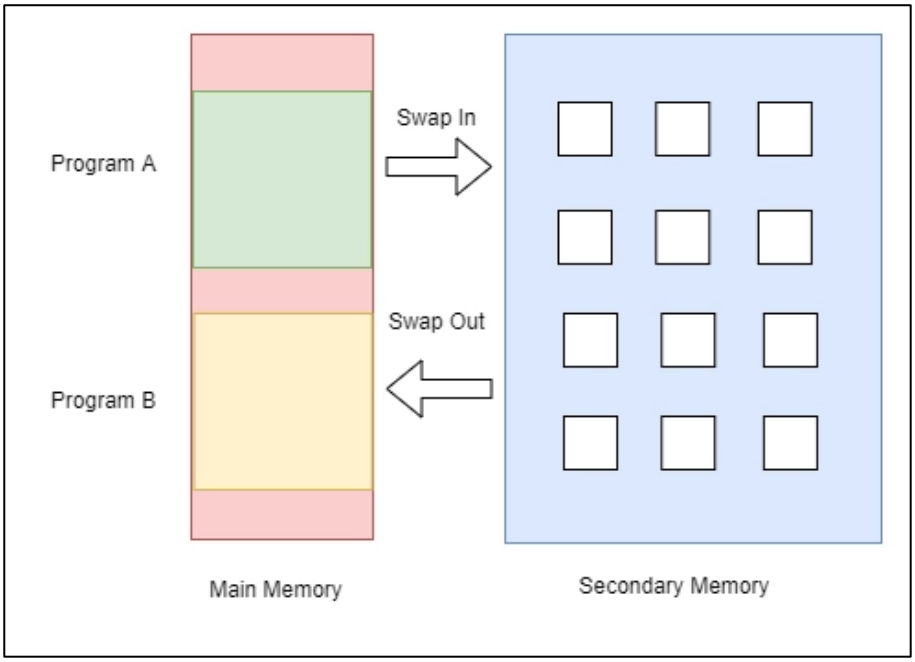


Figure 3.3.1: Demand Paging

[Source - https://www.naukri.com]

Advanced Demand Paging Considerations

There are a few more advanced topics concerning demand paging beyond the scope of basic implementation. A key element of this is managing changed pages, so-called dirty pages. If an operation modifies a page in memory, the OS must write back the changes to disk before replacing the page. This is generally done by keeping a dirty bit in the page table, which is set when a page is modified. When it is determined that a page needs to be replaced, the operating system checks the dirty bit. If the bit is set, the page will be written back to disk, if not, it gets discarded. However, keeping consistency across the data has an implication that adds an overhead to the paging replacement process which is an impact of the write-back operation. Another factor is the use of shared pages. This reduces memory usage

and can lead to performance improvements because shared pages allow the same physical page to be used by multiple processes in-memory. There are system-wide caches as well, for example, if there are multiple processes running the same program, these processes can share the code pages of the program. The pages are usually sharing implemented with a reference count, which counts the how many processes are sharing the page right now. If a process doesn't need the shared page anymore, the reference count is decremented. The page can be reclaimed once the reference count drops to zero. In fact, you are trained on shared pages although, copy-on-write (COW) pages are given higher efficiency. The fork system call creates a new process, and initially the pages are shared between the parent and the child. On the other hand, as soon as one of the processes tries to modify a shared page, a copy of the page gets created and the modification is applied to the copy. Such system call results Page Fault on page level; So very minimal pages are copied on process creation. You can also do things like page buffering, which works to improve the performance of demand paging by keeping a pool of free frames. When a page fault happens, the operating system can easily take a free frame from the pool, thus improving the latency of retrieving a page. So, before they are really used, they put it in a memory page buffer, that is called page buffering.

3.3.2 Page Replacement Algorithms and Virtual Memory

With the shift to modern OS, virtual memory is the backbone that allows processes to run without needing to load their entire memory into physical RAM. The trick is enabled by a subtle combination of hardware and software working behind the scenes: pages — the discrete units of virtual memory are transferred between the limited main memory and more commodious secondary storage as needed. The process of swapping pages in and out of memory is allowed, but requires implementations of effective algorithms to practice a certain strategy for when to evict a page to load in a new one to make it more efficient. This approach is complicated, however, because the behavior of a process is difficult to predict, and it is not easy to say which pages are the least likely to be needed immediately. The earliest and one of the conceptually simplest algorithms is called the First-In, First-Out (FIFO) algorithm. It works on the principle of replacing the oldest page

in memory. Although simple to implement, FIFO is subject to Belady's anomaly, in which increasing the number of page frames may sometimes result in an increase in the number of page faults, which is counterintuitive and undesirable. In contrast, the Least Recently Used (LRU) algorithm looks to evict a page that has not been accessed for the longest time. The hint behind this algorithm is localization which explains that recently accessed memory addresses are likely to be accessed again in the near future. LRU is often more efficient than FIFO, but it requires keeping track of a history of page usage, which can have a non-negligible cost. Access time journals (aka workless access time journal devices) (or seemingly all-in-ones (with pagetable cache ontop of workless ram) computelemens or what have you) back ends usually rely on hardware (counters getting set to zero (max delay) on page access) to track access times. Page Replacement Algorithms Belady's optimal (OPT) algorithm is a theoretical but unattainable

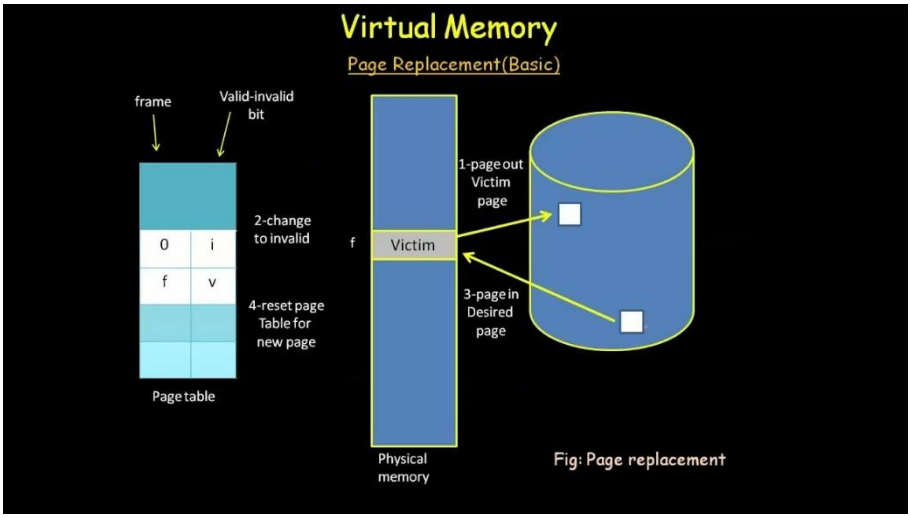


Figure 3.3.2: Page Replacement Concept

optimum for page replacement.

The Optimal (OPT) algorithm. The page replacing Algorithm is the best theoretical optimum scope for page replacement. It replaces the page that will not be used for the longest period of time in the future. Please note: OPT offers the optimal page fault rate; this is how it is defined; however, in practice, it requires knowledge of future memory accesses, which are impossible to have in real life, and hence, impractical. Nevertheless, OPT is a good baseline to use for estimating the performance of other page replacement algorithms. The clock replacement algorithm, also called the Second Chance algorithm,

provides a compromise between the simplicity of FIFO and the efficiency of LRU. It has a circular queue of pages, and a use bit for each page. When it's time to replace a page, the algorithm walks down the queue, resetting the use bit for every page it sees. If a page with a cleared use bit is found, it is replaced. When all pages have their use bits set, the algorithm resets use bits and keeps traversing the queue until a page with a cleared use bit is encountered. Because it is efficient and does not suffer from Belady's anomaly, the Clock algorithm is widely used in many operating systems. The Clock algorithm and its variants (e.g. Not Recently Used algorithm) refine the algorithm by taking the use bit and modified bit (whether or not the page has been modified since loading into memory) into account. This allows the algorithm to focus on replacing clean pages before dirty pages, which can save on the cost of writing existing modified values back to secondary storage. Working Sets Working sets also help us understand page replacement. A working set is the collection of pages being actively used by a process over a time interval. Working set model: The working set model tries to keep the working set of the process in the memory to reduce page faults and improve performance. The model requires estimating the size of the working set, which is difficult to do. Thus, it does not give the complete image but it serves its purpose by being a useful guide to prevent the memory from being shredded into million pieces in a Virtual memory system. Another algorithm is called the page fault frequency (PFF), which dynamically allocates page frames to a process depending on its page fault rate. In performing this algorithm, if the rate of page fault is high, the number of page frames gets increased by the algorithm, and if the page fault rate is low, the number of page frames gets decreased. By adapting in this manner, memory usage is kept optimized, and the system's performance benefits. One of the major concerns in the context of virtual memory is the idea of thrashing, which is when a process spends more time swapping pages in and out than executing instructions. These working sets are what's stored in the system memory, including RAM, which is why thrashing happens. Operating systems, to successfully eliminate thrashing, can use load control, which is the adjustment of the level of multiprogramming (the number of processes that can be in execution at a given time), and working set. A study of an interface between hardware and software is essential for understanding the

implementation aspects of virtual memory. MMU (Memory Management Unit) is a hardware unit that translates a virtual address to a physical address from the physical address to a page table mapping virtual memory to physical memory In contrast, the operating system is responsible for maintaining the page table--the data structure that keeps track of the mapping between virtual and physical pages--and executing the page replacement algorithm. The performance of virtual memory relies on how well this teamwork works. Though for the simplest sense, the modern operating systems are still reliant on the base concepts of page replacement but they also have integrated the concepts of demand paging where the pages are loaded from disk into physical memory only when they are needed and page clustering where similar pages are clustered together in such a way that the number of page faults could be less. The use of these approaches, in combination with smart page replacement algorithms, allows virtual memory to operate smoothly and effectively, facilitating the proper performance of processes, regardless of the limited number of physical memory resources that the system possesses.

3.3.3 File Concepts

Files are fundamental abstractions in operating systems, providing a structured and persistent mechanism for storing and retrieving data. They serve as the primary means for users and applications to interact with data, whether it be documents, images, executables, or system configuration files. A file, at its core, is a named collection of related information that is recorded on secondary storage, such as hard disks, solid-state drives, or optical media. The concept of a file encompasses not only the data itself but also metadata, which includes information about the file's attributes, such as its name, size, creation date, and access permissions. The file system, a crucial component of the operating system, is responsible for organizing and managing files and directories. It provides a hierarchical structure that allows users to organize files into directories, creating a logical and intuitive file organization. Directories, also known as folders, can contain both files and other directories, forming a tree-like structure that facilitates efficient file management. The file system also manages the allocation of storage space, ensuring that files are stored and retrieved efficiently. Different file systems employ various data structures and algorithms to

manage storage space, such as linked lists, bitmaps, and inodes. The choice of file system can significantly impact performance, reliability, and security. File naming conventions vary across operating systems, but they generally adhere to certain rules and guidelines. File names typically consist of a base name and an optional extension, separated by a period. The extension indicates the file type, such as .txt for text files, .jpg for image files, and .exe for executable files. Operating systems impose restrictions on the length and characters allowed in file names to ensure compatibility and avoid conflicts. File types are essential for identifying the format and structure of a file. Operating systems recognize various file types and associate them with specific applications. This allows users to open and manipulate files using the appropriate software. File types can be classified into several categories, such as text files, binary files, executable files, and directory files. Text files contain human-readable characters and are typically used for storing documents, source code, and configuration files. Binary files contain non-text data, such as images, audio, and video, and are typically processed by specialized applications. Executable files contain machine code that can be executed by the operating system. Directory files contain information about other files and directories, forming the hierarchical structure of the file system. File access methods determine how data is accessed and manipulated within a file. Sequential access is the simplest access method, where data is accessed in a linear order, from the beginning to the end of the file. This method is efficient for processing large files that are accessed sequentially, such as log files and backup files. Direct access, also known as random access, allows data to be accessed in any order, regardless of its position in the file. This method is efficient for accessing specific records or data elements within a file, such as database files and index files. Indexed sequential access combines the advantages of sequential and direct access. It uses an index to locate specific records within a file, allowing for both sequential and direct access. This method is commonly used in database management systems and file systems that require efficient access to large amounts of data. File attributes provide information about the characteristics of a file, such as its name, size, creation date, modification date, and access permissions. File attributes are stored in the file's metadata and can be accessed and modified by users and applications. Access

permissions control who can access and manipulate a file. They typically include read, write, and execute permissions, which determine whether a user can read, modify, or execute a file. Access permissions can be set for different user groups, such as the file owner, group members, and other users, ensuring that files are protected from unauthorized access. File operations are the actions that can be performed on files, such as creating, deleting, opening, closing, reading, writing, and renaming. These operations are typically provided by the operating system through system calls, which allow applications to interact with the file system. File systems employ various techniques to ensure file integrity and reliability, such as journaling, which logs file system changes before they are applied, and checksums, which detect data corruption. These techniques help to prevent data loss and ensure that files are stored and retrieved correctly. File caching is another technique used to improve file system performance. It involves storing frequently accessed file data in memory, reducing the need to access secondary storage. File caching can significantly improve performance, especially for applications

3.3.4 File System Structures and Implementation

The file system structures and implementation is what underlines any operating system's capability to manage persistent data. Why is there a file system? At the most basic level, a file system is a natural way of organizing data when stored, enabling users and applications to access, modify and share data. It abstracts away the intricacies of physical storage devices, providing a straightforward interface for data management. From raw storage blocks to a coherent file system is a long and complex journey involving a myriad of design decisions and

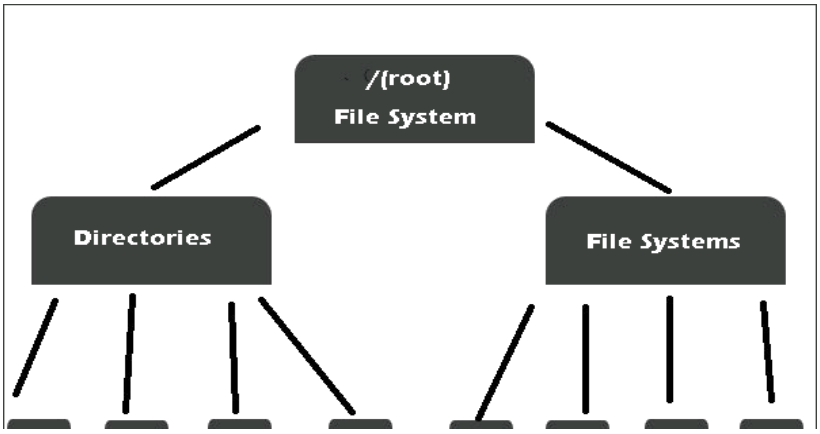


Figure 3.3.3: File System in OS

implementation details, each of which greatly impact the end product's performance, reliability, and security. At the core, file system is built upon a hierarchical structure usually represented as a tree, where directories (or folders) act as containers for files and other directories. Such a hierarchical structure encourages a logical organization of related files, making it more navigable and manageable. At the very top of the hierarchy is the root directory, which serves as the entry point for the entire file system. In this arrangement, files are located by their pathnames lists of directories to navigate through until the desired file is located. Must also store metadata (e.g. names, size, timestamps, permissions, owner, etc.) in addition to the actual data content. This metadata is important in regards to file management operations and is often stored in data structures such as inodes or file allocation tables. A file system must manage both data and metadata, and the efficient organizing and accessing of this information is key to performance. Different allocation policies are used by file systems, which have effects from fragmentation to access speed to storage utilization. Here are some of the most commonly used methods of allocation: contiguous allocation, linked allocation, and indexed allocation. Discontinuously, on the other hand, is efficient with access but may leave behind external fragmentation. In linked allocation, the blocks are connected using pointers, which can reduce fragmentation but uses more memory for random access. In contrast, indexed allocation creates an index block that points to data blocks, allowing random access but also requiring more storage for the index. The allocation strategies are chosen based on the expected usage patterns and performance requirements for the file system. Beyond merely managing data and metadata, file systems must grapple with concurrency control, crash recovery, and security. Concurrency control mechanisms, including locks and transactions, provide the ability for multiple processes in a system to read and write files without corrupting data. Crash recovery mechanisms such as journaling, logging, etc., allow the file system to restore its consistency post a system crash. Such as access control lists (ACLs) and encryption help to secure sensitive data by preventing unauthorized access. How these features are designed and implemented have a major effect on the file system's reliability and robustness.

The implementation of a file system involves a complex interplay of data structures, algorithms, and system calls. The operating system kernel plays a central role in managing the file system, providing an interface between user applications and the underlying storage devices. The kernel maintains data structures that represent the file system hierarchy, metadata, and allocation information. These structures are often stored in memory to facilitate fast access and manipulation. When a user application requests a file operation, such as opening, reading, writing, or deleting a file, the kernel translates the request into a series of operations on the storage device. This involves locating the file's data and metadata, allocating or deal locating storage blocks, and updating the relevant data structures. The kernel provides system calls, such as open(), read(), write(), close(), mkdir(), and rmdir(), which serve as the interface between user applications and the file system. These system calls encapsulate the low-level details of file operations, allowing applications to interact with the file system in a standardized and platform-independent manner. The implementation of these system calls involves intricate algorithms for navigating the file system hierarchy, managing metadata, and accessing storage devices. For instance, the open() system call typically involves searching the directory structure for the specified file, verifying access permissions, and allocating a file descriptor to represent the opened file. The read() and write() system calls involve locating the file's data blocks, transferring data between the storage device and the application's memory, and updating the file's metadata. The close() system call releases the file descriptor and updates the file's metadata, such as the last access time. The kernel also manages the buffer cache, a region of memory used to cache frequently accessed file data and metadata. The buffer cache improves file system performance by reducing the number of disk accesses, which are significantly slower than memory accesses. When an application requests data from a file, the kernel first checks the buffer cache. If the data is present in the cache, it is retrieved directly from memory, avoiding a disk access. If the data is not in the cache, the kernel reads it from the disk and stores it in the cache for future use. The buffer cache employs various replacement algorithms, such as least recently used (LRU), to manage the cached data and ensure that frequently accessed data remains in the cache. The implementation of the buffer cache is critical for file system

performance, as it directly impacts the speed at which applications can access and manipulate files.

The choice of file system implementation significantly impacts the overall performance and reliability of the operating system. Different file systems employ varying data structures, algorithms, and techniques to manage data and metadata, each with its own set of trade-offs. For example, the FAT (File Allocation Table) file system, commonly used in older versions of Windows, uses a simple linked allocation scheme and a flat directory structure. While FAT is relatively simple to implement and understand, it suffers from performance limitations, especially with large files and fragmented disks. The NTFS (New Technology File System), used in modern versions of Windows, employs a more sophisticated B-tree structure for managing metadata and supports advanced features such as journaling, access control lists, and encryption.NTFS offers better performance and reliability than FAT, but it is more complex to implement and manage. The ext4 (Fourth Extended File system), commonly used in Linux distributions, also employs a B-tree structure for metadata management and supports features such as extents, which improve performance for large files, and delayed allocation, which reduces fragmentation.Ext4 is known for its performance and scalability, making it suitable for a wide range of applications. The implementation of a file system also involves considerations for portability and interoperability. Operating systems may support multiple file systems, allowing users to access data stored on different devices or partitions. The kernel must provide a common interface for accessing these file systems, abstracting the differences in their underlying implementations. This involves the use of virtual file system (VFS) layers, which provide a uniform interface for file system operations, regardless of the specific file system being used. The VFS layer translates generic file system operations into specific operations for the underlying file system, enabling applications to interact with different file systems in a consistent manner. The implementation of the VFS layer is crucial for supporting multiple file systems and ensuring interoperability between different operating systems. Furthermore, the implementation of distributed file systems, such as NFS (Network File System) and AFS (Andrew File System), involves additional complexities related to network communication, data consistency, and fault tolerance. Distributed file systems allow multiple

computers to access and share files over a network, enabling collaborative work and resource sharing. The implementation of these file systems requires careful consideration of network protocols, caching strategies, and security mechanisms to ensure efficient and reliable data access. The design and implementation of file systems continue to evolve, driven by advancements in storage technology, changing user requirements, and the need for improved performance, reliability, and security. In essence, the file system implementation constitutes a critical component of the operating system, bridging the gap between user applications and physical storage devices. It involves intricate algorithms, data structures, and system calls to manage data and metadata effectively. The kernel plays a pivotal role in orchestrating file system operations, providing an interface for user applications and managing the buffer cache to enhance performance. The choice of file system implementation significantly impacts the overall performance, reliability, and security of the operating system. Different file systems offer varying trade-offs, and the selection depends on the specific requirements of the system and its intended usage. The implementation of the VFS layer enables interoperability between different file systems, while distributed file systems facilitate network-based file sharing. As storage technology advances and user demands evolve, file system implementations continue to adapt and innovate, ensuring efficient and reliable data management. The efficiency of a file system is judged by its speed of access, its reliability in the face of system failures, and its ability to manage storage space effectively. The speed of access is determined by factors such as the allocation strategy, the buffer cache size, and the disk access time. The reliability is ensured through mechanisms such as journaling, logging, and redundant storage. The ability to manage storage space is influenced by the file system's ability to minimize fragmentation and utilize available space efficiently. Modern file systems also incorporate features such as data compression and encryption to enhance performance and security. Data compression reduces the amount of storage space required for files, while encryption protects sensitive data from unauthorized access. The implementation of these features requires careful consideration of performance trade-offs and security implications. The future of file system implementation lies in addressing the challenges of managing increasingly large and complex

data sets, supporting diverse storage technologies, and ensuring security and reliability in distributed and cloud-based environments. As data continues to grow exponentially, file systems must evolve to handle the demands of modern computing and data management. Finally, the intricacies of file system implementation extend beyond the core functionalities of data storage and retrieval. The modern computing landscape demands sophisticated features that cater to diverse user needs and evolving technological paradigms. Features such as snapshots, which allow for point-in-time recovery of file system states, are increasingly vital for data protection and disaster recovery. Similarly, copy-on-write (COW) techniques optimize storage usage and enhance performance by delaying physical data copying until modifications are made. These advancements underscore the continuous innovation within file system design, driven by the need for efficiency and resilience. Furthermore, the rise of cloud computing has necessitated the development of scalable and distributed

3.3.5 Free Space Management: Principles, Techniques, and Implementation

It is the core of a congruous operating system to expose persistent data which lives inside its own file system structures and implementation. Essentially, file system is a computing method known as the logical organization of data being stored, so the user and any application can read, edit or share information easily. It abstracts the chaff of physical devices into a form that is much more useful, which allows you to deal with data, rather than devices. Creating a complex file system on top of ordinary storage blocks requires careful thought and systematic execution: every decision at the design and implementation stages of the project can have a tremendous impact on speed, dependability, or even safety of data. Essentially, a file system is based on a hierarchy one that is normally represented as a tree in which directory (folder) nodes are used to contain files and other directory nodes. This Top-Down Organization Makes for Naturally Related Files That Are More Effortlessly Navigable and Manageable. The highest node in the hierarchy is called the root directory, which provides the entry point to the whole file system. Under this structure, files are referenced by their pathnames, which define the path through the directory hierarchy to the file. The file system has to keep track of metadata that describes file

names, sizes, creation and modification timestamps, permissions, and ownership data in addition to the actual data itself. This metadata is essential for file management operations and is usually stored in data structures such as inodes or file allocation tables. Efficient organization and retrieval of data and metdata is paramount to the performance of the file system. There are a number of different allocation strategies that a file system can use to allocate the real physical space, which has implications for fragmentation, speed of access, and overall utilization of storage space. The most common methods include contiguous allocation, linked allocation, and indexed allocation. In contiguous allocation a file gets a sequence of blocks, providing faster sequential access but causing external fragmentation. Linked allocation links blocks using pointers which avoids fragmentation but adds random access cost. Reloading of pointers from data block example in Indexed allocation Indexed allocation uses index block that contains pointers to data blocks which allows random access, but comes at the price of using more space to store the index. Which allocation strategy to chose depends on the access patterns and performance of the file system. File systems also deal with concurrency control, crash recovery, and security, among other things, in addition to data and metadata management. Mechanisms for concurrency control, such as locks and transactions, help ensure that multiple processes can simultaneously access and modify files without corrupting the contents. Crash recovery procedures (such as journaling and logging) allow the file system to return to a consistent state after a system crash. Sensitive data is safeguarded by security measures like access control lists (ACLs) and encryption. What and how these features are designed and implemented has a great influence on the reliability and robustness of file system.

Theoretical Foundations and Fundamental Algorithms

The figure (left) shows how the operating system kernel manages the file system, acting as an interface between the user applications and the underlying storage devices. Data structures that represent the file system hierarchy, metadata and allocation information are maintained by the kernel. These are usually kept in the memory to enable them to be accessed and modified quickly. When a user application needs to perform a file operation like opening, reading, writing, and deleting a file, the kernel converts that request to a series of operations on the

storage device. This includes finding the file's data and metadata, allocating or deal locating storage blocks, and updating the corresponding data structures. User applications interact with the system calls provided by the kernel, including open, read, write, close, mkdir, and rmdir; these function calls act as the interface to the file system. These system calls abstract the underlying complexities of file manipulation, enabling applications to communicate with the file system in a uniform and OS-agnostic approach. Basic file system functionality: File systems provide a set of system calls for operations like opening, reading, writing, and closing files. The open system call usually requires traversing the directory structure to find the requested file, checking access rights, and allocating a file descriptor to represent the opened file. The read and write system calls go through finding the file's data blocks, moving blocks of data around from the storage device to the memory of the application and updating the file's metadata, etc. assert close fd The close system call closes a file descriptor and updates the file's metadata (e.g. last access update). The kernel also controls a method called a buffer cache, which is a portion of memory that is used to store file and metadata that is frequently accessed. The post cache improves file system performance by reducing the number of necessary accesses to disk, which are orders of magnitude slower than memory accesses. It works by checking a cache that sits between the application and the file itself. When there are a huge number of records, this greatly speeds up data retrieval since the data is only fetched from the memory, not from the disk. If the data is no longer on the cache, the kernel fetches it from the disk and places it in the cache for subsequent access. It uses various replacement algorithms, including least recently used (LRU) as examples, to efficiently manage the cached data (more frequently accessed data should remain in the cache). The buffer cache is responsible for file system performance, which is what makes every application read and write files faster.

3.3.6 Memory Allocation Strategies and Fragmentation Management

In other words, memory allocation strategies are tactical implementations of free space management principles, moving us from theoretical constructs to systems that balance competing objectives.

Choosing the right allocation strategy is largely influenced by workload characteristics, hardware architecture, and application needs. The consecutive-fit strategies first-fit, next-fit, best-fit, and worst-fit strategies only differ in the search policy for free lists. Unlike first-fit, which always starts from the beginning of the free list, next-fit continues from the last allocated location, which likely improves locality but may fragment hot sections in memory over time. The best-fit and the worst-fit strategies optimize for different goals by the former minimizing the short-term waste of memory at the cost of creating completely unallocatable small fragments while the latter preserves the large continuous regions of memory at the cost of short-term wastage. One example of a power-of-two strategy is the buddy system, which

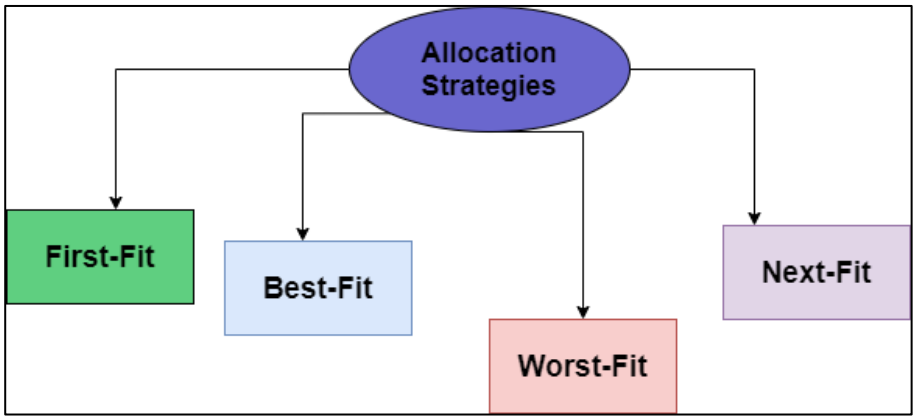


Figure3.3.4: Memory Allocation strategies

only allows for allocations to power-of-two sizes, making bookkeeping easier; the drawback is internal fragmentation. If a block is freed in this system, it can be combined with its buddy (the adjacent block of the same size) to make a bigger block, which may make it possible to mitigate external fragmentation.

Slab allocation, introduced in solaris, is a technique where memory blocks are pre allocated (called slabs) for certain types of objects, making it because each allocated object knows its size and freeing the memory blocks for more easy insertion of allocated objects. Segregated free lists keep a separate pool for each size class, allowing for fast allocation of common sizes and improved locality, but at the cost of increased overall memory consumption due to potential fragmentation, since a smaller allocation won't fit into the pool of a larger allocation. These kinds of allocators are typically bitmap-based, meaning that they store the state of each byte of memory in a bit vector very compact but slightly slower allocation than with list-based allocators. The main

problem with free space management is fragmentation, which occurs when the free memory is broken into non-contiguous nodes so blocks cannot be fully utilized, which has two kinds external fragmentation (inaccessible gaps between allocated blocks) and internal fragmentation (the space allocated but not actually used). Many different techniques used in the implementation of allocation strategies to avoid fragmentation such as split (to break down the larger blocks to satisfy smaller requests), coalescing (the merger of two adjacent free blocks), compaction (the process of moving allocated objects to give larger spaces of free in a contiguous manner), and size rounding (the practice of standardizing the size of allocations, therefore, avoiding very small ones) These techniques hit different performance notes depending on the allocation profile of the application: programs that allocate lots of short-lived objects benefit from allocation-time optimizations like generational schemes, whereas long-running systems are characterized by their small but more unpredictable lifetimes and require balanced techniques that avoid the accumulation of fragmentation over extended periods of time. Adaptive allocation strategies monitor their workload and adapt their behavior according to observed workload properties, changing policies when needed, for example, depending on memory pressure or allocation patterns. Adapting request optimization strategies dynamically based on data regarding allocation requests, memory usage over time, and fragmentation statistics. How freed memory is retained (or not) also affects fragmentation and performance through memory reservation policies, which dictate how memory is provisioned for use beyond immediate needs: over-reserving memory decreases fragmentation but wastes memory resources, while under-reserving memory (to keep it less fragmented) means that you have to do increasingly frequent resizing operations that involve costly system calls or reorganization of memory.

Operating System Memory Management and Virtual Memory Integration

The most obvious and important implementation of free space management principles is in operating system memory management, which gets instrumental as the bridge between hardware resources and application needs. Most modern operating systems implement a layered architecture on the topic of memory management, utilizing a virtual

memory space that provides an additional layer of abstraction separating the viewpoint of the application from the physical structure of the storage devices. Giving this sort of abstraction allows advanced free space management techniques that would simply not be possible in systems with only physical memory. The virtual memory I/O sub-system will divide the address space to fixed size pages (usually 4KB

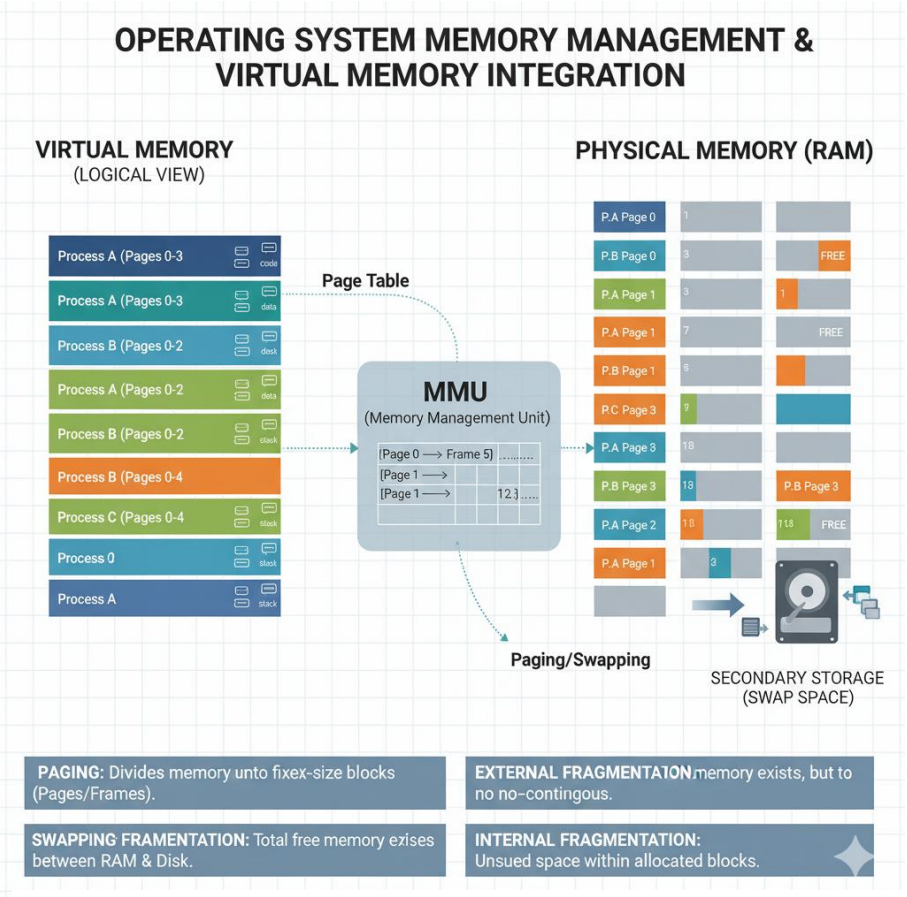


Figure3.3.5: Operating System Memory Management and Virtual Memory Integration

up to 64KB) which will be mapped to physical frame when using page tables, and use Translation Lookaside Buffers (TLBs) to reduce the address translation process. This paging mechanism adds a distinct type of free space management at several levels: free virtual address ranges within each process's address space, free physical memory frames, and pages moving between main memory and secondary storage via page replacement algorithms. During such periods the operating system free space manager needs to balance conflicting requests from different processes and ensure that the system remains responsive and stable at different load levels. Demand paging, which is bringing pages into memory and still keeping them on disk until they are accessed stands as a more specialized version of the lazy allocation and seeks to

improve memory resource utilization as it postpones the physical resource commitment to pages until the very moment where such pages are required. Low-level page replacement policies like LRU, Clock, Working Set, and ARC carry out advanced free space management techniques that follow past fault experiences to anticipate future access patterns. Many operating systems have a mechanism called memory over commit that adds a level of abstraction to memory management the total of all virtual memory allocated can exceed the amount of physical memory available; it allows the free space manager to act as if it had all the resources it doesn't currently have at its disposal, based on statistical multiplexing, keeping in mind that requesting all allocated memory in the same time is rare. Operating systems provide multi-level free space management at diverse granularities with distinct strategies: coarse-grained management of large contiguous regions needed for memory-mapped files or shared memory segments, medium-grained management for process heap allocations and fine-grained management for kernel-internal data structures. Most kernel physical frame allocators use the buddy system, zones, or some hybrid which tries to optimize performance and memory used by balancing the requirements of both. Memory compaction methods are used to periodically defragment physical memory into larger contiguous ranges, resulting in larger contiguous physical memory to service enormous pages (megabytes to gigabytes sized pages) on with less TLB pressure on applications with sizable working collections. Operating system free space manager and processor Memory Management Unit (MMU) integration, in particular under NUMA (Non-Uniform Memory Architecture), where memory access time varies as a function of the distance between processor and memory location, adds more complexity. Modern OS uses page migration and allocation policies to favor local memory allocation while balancing the load amongst the memory nodes. Free space management is furthermore complicated when the kernel needs to deal with hardware prefacers, cache hierarchies, and memory controllers, as decisions about where to place memory affect not only how well you are packing the boxes but also the latency of access to boxes and the use of bandwidth. Specialized memory types such as persistent memory (PMEM) or high-bandwidth memory (HBM) add even more complexity to free space management by creating heterogeneous memory pools with varying performance

characteristics, cost profiles and persistence guarantees, necessitating sophisticated tiring and placement algorithms.

User-space Allocators: Design, Implementation, and Optimization

As the primary interface from applications to the operating system memory management facilities—with each user-space memory allocator utilizing their own sophisticated free space management techniques tuned to application-specific workloads while abstracting away system calls and virtual memory operations. Some allocators allocate memory in bulk from the OS using sbrk or mmap or some other syscall, and implement free space management through return stacks and per-thread caches as optimizations on top of the more granular allocations. General-purpose allocators like malloc/free implementations must strike a balance between performance across a variety of workloads with unpredictable allocation patterns, object lifetimes and size distributions. These allocators are designed with attention to thread safety, cache locality, fragmentation, and allocation speed. There are a variety of production-quality implementations such as ptmalloc (the allocator used by GNU libc), jemalloc, tcmalloc, and mimalloc which in combination cover many points in the design space each with a different focus on different aspects of the allocation problem. Ptmalloc uses per-thread arenas to avoid contention, where each of these arenas implements a hybrid best-fit and segregated fit algorithm. Jemalloc, on the other hand, focuses on reducing fragmentation by using a carefully chosen set of size classes and regularly purging unused memory, making it especially well-suited for long-lived applications. The primary focus of tcmalloc is scalability in multi-threaded environments using thread-local caches and a central heap for pages, while mimalloc emphasizes security and performance by techniques like eager coalescing and secure free lists. Specialized allocators are optimized for certain workload characteristics: pool allocators preallocate memory for objects of a single size, which allows for very rapid allocation and deal location in returns for absolute flexibility; region-based allocators (aka arena allocators) allow only bulk deal location, simplifying memory management for phases of a computation with well-defined lifetimes; and object-specific allocators will implement custom strategies that suit particular data structures or usage patterns. In garbage-collected environments, free space management encompasses memory reclamation by means of automated

compaction, and allocators that are tailored to work with collector algorithms. These allocators commonly reflected fast paths for allocation, object contiguity to allow efficient collection, and management of metadata for efficient reference tracking. Mark-sweep collectors want allocators that can effectively reuse variably-sized free blocks, and copying collectors capitalize on bump-pointer allocation strategies over if contiguous memory regions. Thread-safe memory management functions provide behavioral guarantees that help ensure safe usage in multi-threaded environments. Modern allocators utilize strategies to reduce contention, such as thread-local caches, lock-free data structures for common operations, and fine-grained locking approaches that improve parallelism at the cost of more complex memory handling. Optimizing the performance of user-space allocators requires clever engineering set of practices like size classes where the sizes of various classes were designed to find the trade-off between internal fragmentation and management overhead, hot/cold splitting, perfecting, and alignment of each slab on the heap to help utilization of the hardware. State-of-the-art allocators utilize hardware features, e.g., transparent huge pages, non-temporal instructions, and cache control primitives, to achieve high performance. Security has become an important consideration in the design of allocators, and many modern allocations have incorporated additional features such as guard pages, canaries, randomization of object addresses, and even separation of the metadata of objects from the objects themselves, to mitigate issues such as buffer overflows, use-after-free vulnerabilities, and double-free attacks. Production allocators are common with cross-system memory management tricks like madvise calls, decommitting of unused pages, memory compaction and so on, which reduce physical pages and vastly enhance overall performance. Another important aspect of modern allocators is their debug ability and introspection capabilities, with support leak detection, heap validation, allocation tracking, detailed statistics gathering, etc. to aid development and debugging efforts.

3.3.7 Specialized Free Space Management Systems

Most memory allocators deal with general-purpose usage, but requires for more specialized free space management systems that can fit the needs of specific domains also are common an impressive demonstration of how the core ideas of memory management can be

customized to specific restrictions and optimization possibilities. While database management systems (DBMS) employ a buffer pool manager responsible for many optimizations triggered by the integration of a specialized free space manager and the in-memory database page cache, the forced-page-layering policies can go beyond simple regency used within memory even to page dirtyness, and I/O scheduling opportunities, and query execution plans. It is common for these systems to include their own application-level memory allocators that are attuned to database workloads, with features such as block-oriented allocation, specialized structures for index nodes, and separate pools for different object types. Another area is they can record information about free space availability, which file systems usually do with a bitmap, extent trees, free lists, etc. Copy-on-write file systems such as ZFS and Btrfs use novel strategies for file space that is free but that also adheres to transactional semantics, while log-structured file systems like F2FS organize free space around sequential writes. Free space management techniques for real-time systems trade off memory utilization efficiency for bounded allocation and deal location times, which is often more critical than memory utilization efficiency in these systems. These systems often seldom use variable latency techniques like global coalescing or complex search algorithms where the time complexity can rapidly increase, in favor of a combination of pre-allocated pools, static partitioning, or scope-based memory allocation. High-performance computing (HPC) environments use specialized allocators that are tuned for extreme levels of parallelism, NUMA awareness, and dedicated computation patterns. This includes topology-aware allocators, custom alignment for vectorized access, and integration with job scheduling systems for whole-node memory usage. Graphics Pipelines use domain-specific memory management for resources such as textures, frame buffers, and geometry data, with custom allocators that understand the 2D or 3D nature of the resources and hardware-specific alignment and padding requirements. Modern GPU compute frameworks offer unified memory models with sophisticated free space management that crosses host and device memory and automatically migrates data based on access patterns, hiding the complexity of explicit transfers. However, embedded systems have limited memory resources and rely on specialized free space management techniques that are applied based on specialized

constraints such as statically allocated objects that require determinism, or objects of fixed size requiring a pool-based allocation as well as custom fragmentation mitigation techniques that exploit application-specific information about allocation patterns and lifetimes. High-throughput, low-latency network stacks employ zero-copy buffer management and must use specialized memory pools for common packet sizes, present in systems such as packet processing systems. Garbage collection systems are perhaps the most specialized type of free space management, containing techniques such as generational collection, concurrent marking, incremental compaction, and region-based collection that take advantage of specific properties and information from the managed languages and runtime environments. Just-in-time (JIT) compilers manage code memory according to their unique needs such as executable memory, alignment requirements in addition to constant caches to invalidate instructions. For example, hypervisors and virtual machine monitors maintain multi-level free space management that must account for physical memory allocation to virtual machines and be able to support features such as memory ballooning, page sharing by using deduplication, and live migration between physical hosts. Container runtimes use specialized memory management techniques that work with cgroup limits, accelerate page cache pressure and enable efficient copy-on-write for container images. Big data frameworks also have custom memory management systems understanding the lifecycle of distributed computations, with specialized techniques for spilling to disk, managing data from shuffles, or leveraging the memory of heterogeneous nodes. When it comes to in-memory databases and caching systems, free space management is typically optimized for key-value storage using techniques such as log-structured memory allocation or slab allocation to minimize fragmentation and maximize throughput.

Future Directions and Emerging Research in Free Space Management

The state of device-free space management is constantly equipped to navigate these shifts driven by technologies, workload characteristics, and computing paradigms that are also evolving. Non-volatile memory technologies (NVM) like Intel Optane, Samsung Z-NAND, and multiple flavors of resistance RAM are obfuscating the classic boundary dividing memory and storage, prompting novel Layers of

Indirection for managing free space that take into consideration persistence, wear-leveling, and hybrid memory hierarchies. The existence of these technologies brings new floors for multi-tenant read / writes performance, write endurance, and failure atomicity that lead to research into special-purpose allocators minimizing writes, batching updates, and recovering from power failures to keep metadata consistent. This heterogeneous memory architectures interoperability of DRAM, HBM, NVM, and traditional storage introduces intricate memory hierarchies, necessitating sophisticated tiering algorithms, placement policies, and migration strategies to cost-efficiently accommodate diverse access patterns and performance characteristics. Increasing popularity of multi-tenant environments in cloud computing has motivated research into isolation-minded free space management techniques that avoid performance interference while maximizing resource utilization through techniques such as page coloring, NUMA-aware allocation, and quality-of-service guarantees for memory bandwidth. As free space management research has turned to security considerations, new techniques such as address space layout randomization (ASLR), fine-grained object protection, guard regions, and memory tagging have emerged to help mitigate vulnerabilities that spring from mistakes in memory management. By combining machine learning and systems programming, new horizons emerge learning-based free space management with allocation strategies adapting to seen distributions, predicting future memory usage patterns with predictive models, and employing reinforcement learning to optimize long-term memory usage on varied workloads. These allocations of resources are made under the influence of energy efficiency, which has become a key design constraint in contemporary computing systems and motivates research on power-aware memory management techniques that factor in the energy cost of allocation decisions, placement decisions and data movement operations. To keep up with the never-ending memory size race, ignoring the properties of the order-of-magnitude difference in the size of memory addressed and the used datasets, research of techniques that keep optimal memory behavior at extreme scales such as hierarchical metadata, probabilistic (and shrinking) data structures for free space management or approximate allocation techniques that absolutely do trade perfect allocation size for allocation algorithm range - have popped up as points

of interest. Rust, Web Assembly, and other memory-safe models are inspiring research works on ownership-based memory systems that use compile-time knowledge about the lifetimes and access patterns of objects to make smarter allocation choices and eradicate entire categories of memory errors. With the increasing significance of domain-specific workloads such as machine learning, genomics, and cryptography — there is a rising interest in domain-specific memory allocation strategies that go beyond the traditional abstraction of a memory page to capture the access patterns/characteristics/life-times/performance requirements of such workloads. New concurrency models beyond classical threading (e.g., asynchronous programming, actor-based and dataflow models) are challenging traditional assumptions of free space management regarding thread-local caching, allocation ordering and synchronization strategies. Although the field of quantum computing is in its early days, it presents many new free space management challenges that are opportunities for terrestrial systems, stemming from the probabilistic nature of quantum states, finite coherence time of qubits, and different demands of quantum algorithms. Besides driving technology, methodological innovations in research on free space management include enhanced analysis and modeling techniques, systematic benchmarking approaches, and formal verification techniques that yield stronger guarantees regarding correctness, performance characteristics and security properties of allocators. Going forward, we will see free space management become even more specialization and adaptive, with systems dynamically choosing between many different strategies based on workload characteristics and the utilization of hardware resources, as well as application-specific needs.

Summary

Contiguous memory allocation is a traditional memory management method in which each process is allocated a single, continuous block of memory. This approach is simple and allows for quick access since memory blocks are adjacent, but it leads to problems such as external fragmentation. As processes are created and terminated, free memory gets scattered in small chunks, which may not be usable even if the total free memory is sufficient, thereby reducing the efficiency of memory usage. To overcome these issues, non-contiguous memory allocation

techniques like paging were introduced, allowing more flexible and efficient memory management.

Paging eliminates the need for contiguous blocks by dividing the process's memory into equal-sized pages and physical memory into frames of the same size. Pages from a process can be loaded into any available memory frames, which removes the problem of external fragmentation and makes better use of available memory. The operating system keeps track of page-to-frame mappings using a page table. Building on this, demand paging is a more advanced concept where pages are loaded into memory only when they are needed during program execution, rather than all at once. This reduces the amount of memory used and speeds up the process load time. However, it introduces the possibility of page faults—when a requested page is not in memory—requiring data to be fetched from secondary storage, which can slow down execution if not optimized properly. Demand paging is a crucial feature in virtual memory systems, helping modern operating systems manage memory more efficiently.

Multiple-Choice Questions (MCQs)

1. Which memory allocation method assigns a single contiguous block to a process?
    a) Paging
    b) Segmentation
    c) Contiguous Memory Allocation
    d) Virtual Memory

    (Answer: c)
2. What is the main drawback of contiguous memory allocation?
    a) High efficiency
    b) Internal fragmentation
    c) Increased system security
    d) Low overhead

    (Answer: b)
3. Which memory management technique allows processes to be swapped in and out of memory?
    a) Paging
    b) Swapping
    c) Segmentation
    d) Virtual Memory

(Answer: b)

4. In paging, what is a page?
    a) A fixed-size block of data stored in main memory
    b) A dynamic memory allocation technique
    c) A method for organizing files
    d) A replacement algorithm

    (Answer: a)
5. Which type of fragmentation occurs in paging?
    a) External fragmentation
    b) Internal fragmentation
    c) Logical fragmentation
    d) No fragmentation

    (Answer: b)
6. Which page replacement algorithm replaces the page that has not been used for the longest time?
    a) FIFO (First In First Out)
    b) LRU (Least Recently Used)
    c) Optimal Page Replacement
    d) MRU (Most Recently Used)

    (Answer: b)
7. Virtual memory allows:
    a) More processes to be executed than the available physical memory
    b) Only real-time execution of processes
    c) Immediate swapping of processes without demand paging
    d) Elimination of the need for secondary storage

    (Answer: a)
8. Which file access method reads data in the same order in which it is stored?
    a) Sequential access
    b) Direct access
    c) Indexed access
    d) Random access

    (Answer: a)
9. What is the purpose of free space management in file systems?
    a) To increase file security
    b) To track unused storage blocks

c) To reduce file sizes
d) To prevent user access to certain files

(Answer: b)

10. Which of the following is NOT a common file system structure?
    a) Single-level directory
    b) Two-level directory
    c) Hierarchical directory
    d) Random directory

(Answer: d)

Short Questions

1. What is contiguous memory allocation, and what are its limitations?
2. Explain the difference between paging and segmentation.
3. What is swapping, and how does it work in memory management?
4. Define internal and external fragmentation.
5. What is demand paging, and how does it improve memory utilization?
6. Name and briefly describe two page replacement algorithms.
7. Define virtual memory, and why is it important in modern operating systems?
8. What are the different file access methods?
9. Describe the structure of a file system in an operating system.
10. What are the different techniques used for free space management in file systems?

Long Questions

1. Explain contiguous memory allocation, its advantages, and its disadvantages.
2. Compare and contrast paging and segmentation, highlighting their advantages and disadvantages.
3. Discuss the concept of demand paging, including the steps involved and its advantages.
4. Explain the different page replacement algorithms (FIFO, LRU, Optimal) and compare their efficiency.
5. What is virtual memory? Discuss its role in memory management and how it is implemented.
6. Describe file system structures and explain the different types of file organizations.

7. How are file systems implemented in an operating system? Discuss various implementation techniques.
8. Explain different file access methods, with examples of where they are used.
9. Discuss the challenges of free space management and describe the various strategies used to manage free space in file systems.
10. How does file system security impact file management, and what are the methods used to ensure data protection?

MODULE 4
DISK SCHEDULING AND DISTRIBUTED SYSTEMS

LEARNING OUTCOMES

- To explore disk structures and scheduling techniques.
- To understand RAID structures and disk management.
- To study distributed system structures and file systems.
- To analyze remote file access, naming, and transparency.

Unit 4.1: Disk Scheduling and Distributed Systems

4.1.1 Disk Scheduling and Distributed Systems

Data management in modern computing systems is a complex dance between data requests, commonly controlled by disk scheduling, and distributed systems, necessitating effective coordination between distributed systems. This article would cover Disk scheduling part part of Operating system which is a huge topic and addresses an important challenge of minimizing the seek time and maximizing the disk throughput. The order in which requests are serviced has a major impact on performance, especially when there are multiple processes simultaneously requesting access to disk blocks. Many disk scheduling algorithms have been designed to solve this optimization problem, including First-Come, First-Served, Shortest Seek Time First (SSTF), SCAN, C-SCAN, and LOOK. FCFS is a straightforward but inefficient disk scheduling algorithm that services requests in the order of their arrival; it results in excessive head movement. SSTF (Shortest Seek Time First) selects the request with minimum seek time from the current head position, optimizing seek time in total but may cause starvation to other requests if far from the current head position. The elevator algorithm, also known as SCAN, moves the disk head either way and services the requests along the way until it reaches one end of the disk, at which point it reverses direction. To counteract this uneven distribution of service, C-SCAN (Circular SCAN) is an option, which moves the head in one direction and begins serving requests back at the beginning of the disk instead of servicing requests on the back trip. LOOK and C-LOOK are optimized versions of SCAN and C-SCAN algorithms, respectively, which do not go to the end of the disk if there are no requests in that direction. This is single-disk, but distributed systems add even more complexity. At first sight, distributed storage and network storage do not sound like the same thing. Distributed file systems, like Hadoop Distributed File System (HDFS) and Google File System (GFS), use data replication and distributed caching to increase fault tolerance and approach high performance. These systems also have to manage network latency, data partitioning and consistency models (e.g., eventual consistency versus strong consistency). In such distributed databases, two-phase commit and Paxos are examples of methods to ensure the atomicity of transactions and consensus in

execution among the nodes. Additionally, incorporating derivative models through cloud computing and edge computing has revolutionized both disk scheduling and distributed systems, leading to virtualized storage and widespread distributed data processing. To address these challenges, organizations increasingly leverage cloud storage services, such as Amazon S3 or Azure Blob Storage for scalable and durable storage, and edge computing platforms that allow for distributed data processing closer to end-users to minimize latency and

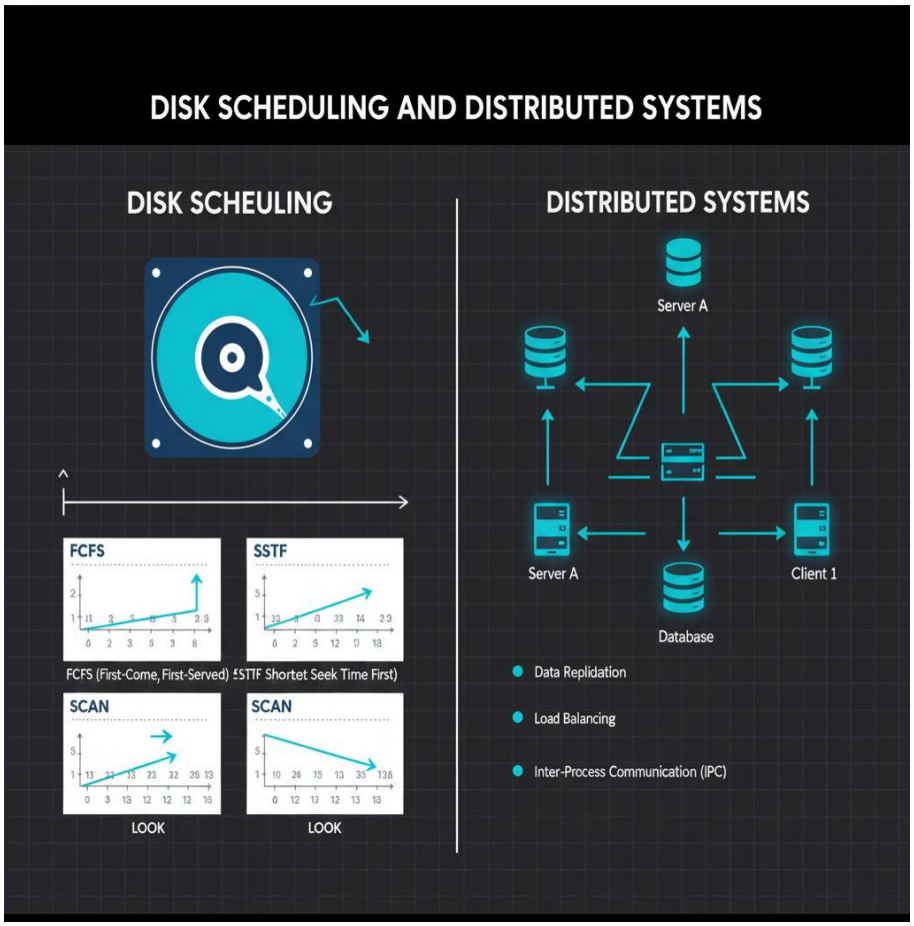


Figure 4.1.1: Disk Scheduling and Distributed Systems

bandwidth usage. From these past trends, newer storage technologies such as SSDs and NVMe have begun entering the market, with SSDs containing orders of magnitude faster access times followed by no seek time at all. For such situations, scheduling algorithms usually target load balancing/ wear leveling for SSD lifetime support. The interaction between disk scheduling and distributed systems remains dynamic, as emerging trends in big data analytics, machine learning, and latency-sensitive workloads push the boundaries of existing architectures,

highlighting the need for dedicated research on the intersection of storage and distributed domain.

Architectures of Distributed Systems

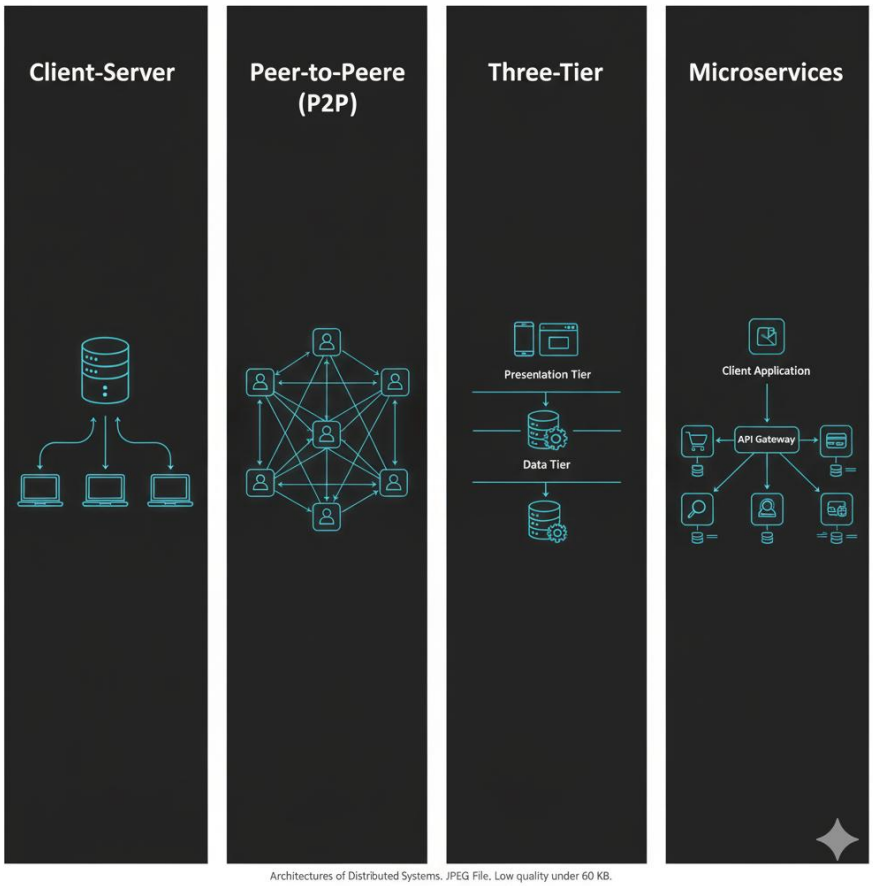


Figure 4.1.2: Architectures of Distributed Systems

- **Client-Server**: Clients request services or resources from a central server. While a simple model, it can become a bottleneck if the server is not redundant.
- **Peer-to-Peer (P2P)**: Each node acts as both a client and a server, sharing resources directly with other nodes. This offers high redundancy and no single point of failure.
- **Three-Tier / Multi-Tier**: This expands on the client-server model by adding separate layers for presentation, application logic, and data management. It's common in web applications.
- **Microservices**: An application is broken down into small, independent services that communicate over a network. This provides flexibility and scalability.

Key Characteristics:

- Resource Sharing: Nodes can share hardware, software, and data.
- Scalability: The system's capacity can be increased by simply adding more nodes.
- Fault Tolerance: If one node fails, the rest of the system can continue to operate, ensuring high availability.
- Concurrency: Multiple components can operate simultaneously.
- Transparency: The user is not aware of the underlying architecture and interacts with the system as a single unit.

Common Algorithms

- First-Come, First-Served (FCFS): The simplest algorithm, it processes requests in the order they arrive. It's easy to implement and fair, but can be very inefficient due to long seek times if requests are scattered across the disk.
- Shortest Seek Time First (SSTF): This algorithm services the request that is closest to the current position of the disk arm. It significantly reduces total seek time but can lead to starvation, where requests far from the head may never be serviced.
- SCAN (Elevator Algorithm): The disk arm moves in a single direction, servicing all requests in its path. When it reaches the end of the disk, it reverses direction and services the remaining requests. This provides a good balance between performance and fairness.
- C-SCAN (Circular SCAN): Similar to SCAN, but the disk arm only services requests in one direction. When it reaches the end, it quickly returns to the beginning of the disk without servicing any requests on the return trip. This provides a more uniform waiting time.
- LOOK and C-LOOK: These are optimized versions of SCAN and C-SCAN. The disk arm only moves as far as the last request in a given direction, instead of going all the way to the end of the disk.

Unit 4.2: I/O Hardware

4.2.1 I/O Hardware

The I/O hardware is the glue that allows seamless interaction between a computer and its external environment and is a key component of any computing system. Wide variety of I/O hardware, including keyboard, mouse, monitor, printer, scanner, network interface, storage devices. Each of these devices are communicating through physical hardware interfaces and protocols to send and receive data and control state. Introduction The fundamental operation of I/O hardware is the communication between the CPU and peripheral devices. These communications are usually controlled by I/O controllers, dedicated pieces of hardware that manage data transfers and interactions with external devices. I/O controllers serve as bridges, converting high-level instructions given by the CPU into low-level signals that the peripheral devices can interpret. One such device might be a disk controller, which will handle the positioning of the disk head, along with actually transferring the data between the disk and the system memory. Just like a network interface controller (NIC) is responsible for sending and receiving data packets on a network. Further training on I/O hardware I/O hardware's continued advancement in speed, key, and connectivity older interfaces have been superseded by high-speed variants, such as PCI Express (PCIe) or Thunderbolt, which provide vastly improved throughput rates compared to their predecessors (PCI & ISA). PCIe Domination One notable aspect of the evolution of the computer motherboard is the widespread adoption of the PCIe standard. The evolution of USB (Universal Serial Bus) has transformed the way we connect peripheral devices, offering a standardized plug-and-play interface for everything from keyboards and mice when was this difference of devices to external hard drives and cameras. USB has gone through many generations, and USB 3.0 and USB 3.1 deliver far greater data transfer speeds than previous iterations. These I/O operations can be leveraged as Shared Network resources or shared disk network, while the development of wireless technology, includes Wi-Fi, Bluetooth that enhance the connection of I/O devices and also allows wireless data transfer. The industry standard for wireless network connection is Wi-Fi, and Bluetooth is used for connecting devices like headsets, speakers, and mobile devices for short-range

wireless connectivity. As a result, there has been a growing need for multimedia applications, and therefore the design of specialized I/O hardware, like a graphics processing unit (GPU), a digital signal processor (DSP). For instance, GPUs are specialized in accelerating graphics rendering and parallel processing and DSPs are designed explicitly to process audio and video. As I/O hardware gets integrated into embedded systems and the Internet of Things (IoT), specialized interfaces and protocols have been developed. IoT devices commonly use low-power wireless technologies (such as Zigbee and LoRaWAN) to connect to the network. All in all, the future of I/O hardware is expected to be influenced by ongoing developments in high-speed connectivity, wireless technologies, and dedicated processors, leading to more immersive and interactive computing experiences.

4.2.2 Application of I/O Interface

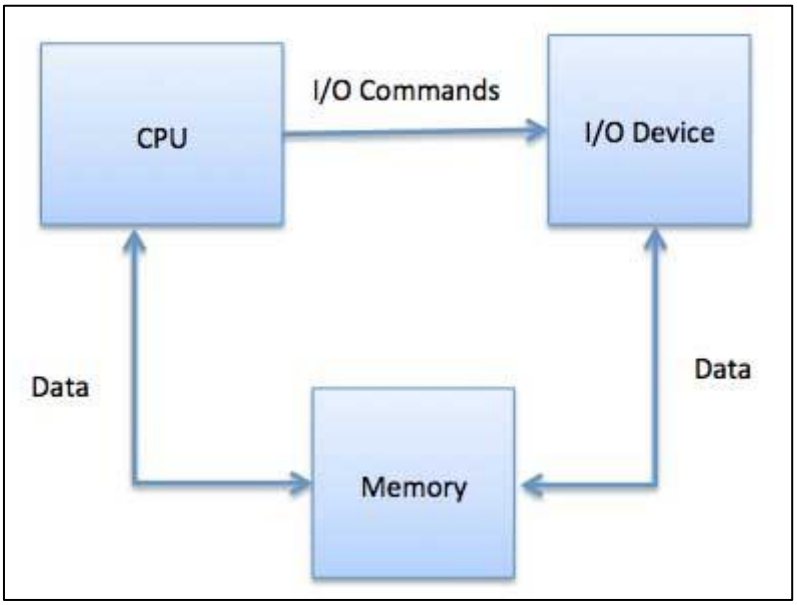


Figure 4.2.1 I/O Hardware
[Source - https://www.tutorialspoint.com]

Due to the versatility of I/O interfaces, they have been utilized in a wide range of fields to facilitate communication between computing systems and the rest of the world. One key component in the architecture of a personal computer is the I/O subsystem that handles input and output interactions between the user and multimedia. They interact with software applications through the use of input devices such as keyboards, mice, and touch screens. Visual output comes from monitors and projectors, whereas audio output comes from speakers and headphones. Which printers and scanners for digitizing/uploading documents and printing output? Multimedia applications have spawned

specialized I/O interfaces like HDMI and Display Port that render high-definition video (HDMI) and audio (HDMI, Display Port) output. USB also comes with a variant of formats various from regular A/B shaped USB plug and cable connection for mobility or as compact as for on the motion like portable solid-state driver, USB interface is anyhow most widely interfaced connector amongst all peripheral devises from external storage devices to cameras has now also become for connectivity in mobiles and tablet. Networking I/O interfaces connect computers and devices to both local area networks (LANs) and wide area networks (WANs). Ethernet interface to translate the packet on a local area network (LAN) and modems and/or routers to bridge between the user and the internet. Examples of network applications especially web browsing, email, and video conferencing depend heavily on the I/O interfaces to transmit and receive data. You are specialized in IoT and embedded systems ·Complete Input/ Output interfaces Typically, however, serial interfaces like UART and SPI are used for communication between embedded devices. Wireless interfaces, including Bluetooth and Wi-Fi, provide wireless communication capabilities for IoT devices, empowering them to connect to the Internet. I/O interfaces play a critical role in industrial automation systems to control and monitor machinery and processes. I/O interfaces are used for data acquisition and control in PLCs and DCS. I/O Interfaces: I/O interfaces play a vital role in the communication between storage devices and computers/servers. Common interfaces for hard drives and SSDs include SATA and NVMe, while Fibre Channel and iSCSI are used for SANs. Storage: A bottleneck at scale in Cloud Computing and Data Centers, which use I/O interfaces for data transfer and storage management Cloud apps generate enormous amounts of data, making high-speed network interfaces and storage interfaces critical. I/O interfaces are also used in specialized areas, including medical imaging, scientific research, and virtual reality. I/O Interfaces are Required in Medical Imaging Devices Instrument and scientific devices, famous are Spectrometers, Microscopes, etc. Motion tracking and haptic feedback in virtual reality systems is managed through I/O interfaces. The automobile and the smarting of everything are modes of I/O interfaces that have piqued my interest beyond abstraction in a display or monitor.

Security and Virtualization in I/O Operations

Such developments, alongside the traditionally more complicated computing systems and the growing pervasiveness of virtualization technologies, have created an environment of critical concerns of security in I/O. The importance of securing input, output, and other I/O operations cannot be overstated—from data breaches to system compromises and even denial-of-service attacks, vulnerabilities found in such interfaces could lead to disastrous consequences. Such unauthorized access can lead to theft/corruption of sensitive data from I/O devices. I/O drivers and firmware vulnerabilities can be exploited by malware to control the system or attack. Secure I/O operations: the procedures for protecting I/O devices and data from unauthorized access and attacks This entails the use of robust authentication and authorization mechanisms, encryption of all data in transit and at rest, regular updating of I/O drivers and firmware to rectify security vulnerabilities, etc. Implementing additional hardware-based security elements, including Trusted Platform Modules (TPMs) and secure boot, to further secure I/Os, the field of virtualization is already well established, particularly for running multiple often disparate operating systems on a single physical machine to maximize utilization.

Unit 4.3: Disk Structures

4.3.1 Fundamentals of Disk Structures

Secondary storage, namely hard disk drives (HDDs) and solid-state drives (SSDs), is the bedrock of virtually all computer systems today, providing permanent data storage. Now, understanding the structure of these disks becomes fundamental to comprehend how data is organized, accessed and managed. Data organization HDD Hierarchical structure Traditional HDDs use magnetic platters to store data. Each platter is divided into concentric circles called tracks and tracks are further divided into sectors. Sectors, which are usually 512 bytes or 4 kilobytes long, are the smallest amounts of data that can easily be read or written. The read/write head is mounted on top of the actuator arm and on the surface of the platters to access certain tracks and sectors. A several platters stacked on an spindle, creating a cylinder, which contains tracks at an equal radial distance on all platters. The next method of addressing data is by means of cylinder, head, and sector (CHS), though this has now been mostly replaced by logical block addressing (LBA). LBA abstracts away these physical details and presents the operating system with a linear sequence of blocks. It abstracts over the disk management and enables faster data access. In terms of components, HDD performance is impacted by the seek time (the time it takes to move the read/write head to the correct track), rotational latency (the time it takes for the target sector to come into position by rotating under the head), and data transfer rate (the rate at which data can be read or recorded to and from the disk). HDDs use spinning disks to write data; SSDs use flash memory which removes HDD mechanical components. Every solid-state drive (SSD) stores data in the form of blocks and pages, where a page is the smallest unit of a read/write operation and a block is a collection of such pages. This is because SSDs don't suffer from seek time or rotate latency like HDDs do, resulting in much faster access times. But SSDs can write only a certain number of times (limited write cycles), which is why wear-leveling techniques are used to ensure write operations are spread across the memory cells evenly. On the disk, the file system handles how files and directories are stored and retrieved. It stores metadata with file names, sizes and timestamps, and allocates disk space to files. The file system, including boot sector and file allocation table (FAT),

and directory structure, can be defined differently based on the operating system and file system type (e.g. FAT32, NTFS, ext4). The file system and the layout of various files and directories on disk, for example.

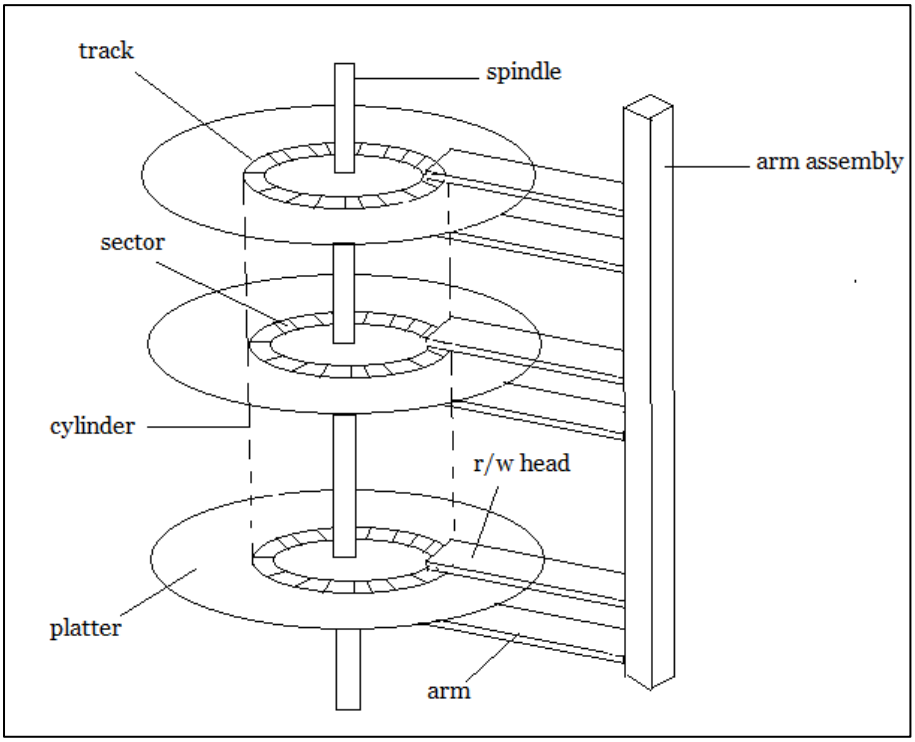


Figure 4.3.1: Disk Structure

[Source - https://www.computersciencejunction.in]

4.4.1 The Importance and Nuances of Disk Scheduling Algorithms

In a multitasking environment, typically multiple processes request access to the disk at the same time, resulting in a queue of pending I/O requests. Various disk scheduling algorithms are used to manage the serving sequence of these requests, with the goal of reducing seek time and thus enhancing overall disk performance. The floating-point purposes of these algorithms have a significant impact on system responsiveness and throughput. There are different algorithms to detect and decrypt a given cipher text, with their own merits and demerits. The first in, first out (FIFO) algorithm is the simplest it services requests in the order they arrive. FCFS is fair, but seek times can be large if requests are scattered all over the disk. Shortest Seek Time First: For each incoming request, SSTF finds the one that has the shortest distance from the current head position and fulfills that. One drawback of SSTF is starvation, where requests too far from the head get stalled indefinitely. Another simple method is the SCAN algorithm (for "elevator"), in which the head moves in one direction, servicing requests as it finds them, until it reaches the end of the disk and then reverses direction. However, while SCAN is favorable for fairness, it may still become detrimental to requests at the far side of the disk, resulting in these requests having very long waiting times. C-SCAN (Circular SCAN); A variant of SCAN where the disk arm services requests in one direction only. C-SCAN offers more consistent wait times than SCAN. Continued algorithm of SCAN and C-SCAN are LOOK and C-LOOK respectively. They do not move to the end of the disk but instead only the farthest request in the current direction. Decreasing unnecessary head movement all the while enhances performance. Depending on the workload and performance requirement, different disk scheduling algorithms can be chosen. The SSTF or LOOK algorithms may be used for applications with large amounts of random workload. Instead, SCAN or C-SCAN algorithms may be better for workloads that have sequential requests. One such advanced disk scheduling algorithm is the Deadline algorithm, which supports real-time guarantees based on the request deadlines. In some cases, the OS may also employ hybrid strategies, blending various algorithms to achieve the best performance over a range of scenarios.

Knowledge of the trade-offs between these algorithms is critical in devising optimal disk management strategies.

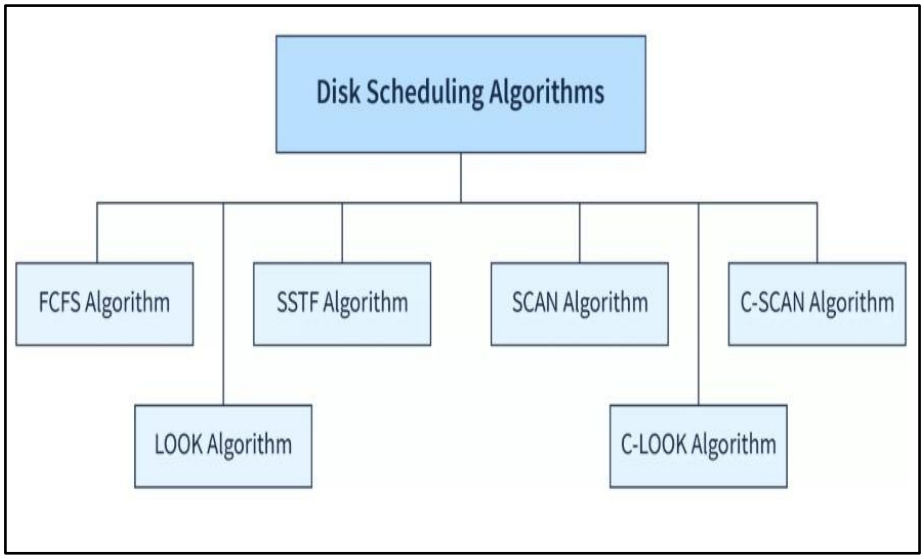


Figure 4.4.1: Disk Scheduling Algorithm

Source - https://www.scaler.com/topics/disk-scheduling

4.4.2 Comprehensive Disk Management Techniques

Proper disk management is essential for keeping systems functioning efficiently, ensuring that data remains intact, and that resources are used in an optimal manner. It involves various methods such as disk formatting, partitioning, file system management, and disk defragmentation. Formatting a disk sets up a file system structure that's required to use the disk. This relies on writing metadata to the disk, like the boot sector, the file allocation table and the directory structure. Disk partitioning is a technique by which we divide the physical disk into logical partitions and use these partitions to run multiple operating systems or file systems on a single disk. Partitions are like separate disk drives and they help keep everything organized and flexible. File systems manage how files are stored and retrieved in storage systems. The file system maintains data structures (such as inodes and file allocation tables) to track the location and metadata of files. In this article, we will learn about disk cleanup, disk defragmentation, how to disk defragmentation and why we need to perform disk defragmentation? How fragmentation happens over time, files can get fragmented. Defragmentation restacks these fragments into cluster blocks minimizing seek time and thus file retrieval. In GNU/Linux, Disk quotas are used to limits, or restrict the amount of disk space that

users or groups can consume, preventing disk space exhaustion, and thus ensuring fair resource distribution. RAID 1; Disk Mirroring RAID 1, or Disk Mirroring, creates a mirror copy of the data across multiple disks for redundancy and flock tolerance. In the event of failure of one disk, the system can still run with the mirrored copy. With disk striping (RAID 0), data is spread across many disks to accelerate read/write speeds. But RAID 0 does not offer redundancy. RAID 5 and RAID 10, for instance, offer both striping and mirroring to provide a balance between performance and redundancy. Disk caching can save the data that is used very frequently in the memory and avoids the excess work over the disk. When the cache is full, you use some cache replacement algorithm like LRU (least recently used) or LFU (least frequently used) to decide which data to remove from the cache. Disk scheduling algorithms are also an important aspect of disk management and they are discussed before where a step is taken for how to optimize the order of I/O requests. When merged, all of them form a potent and operational disk management system.

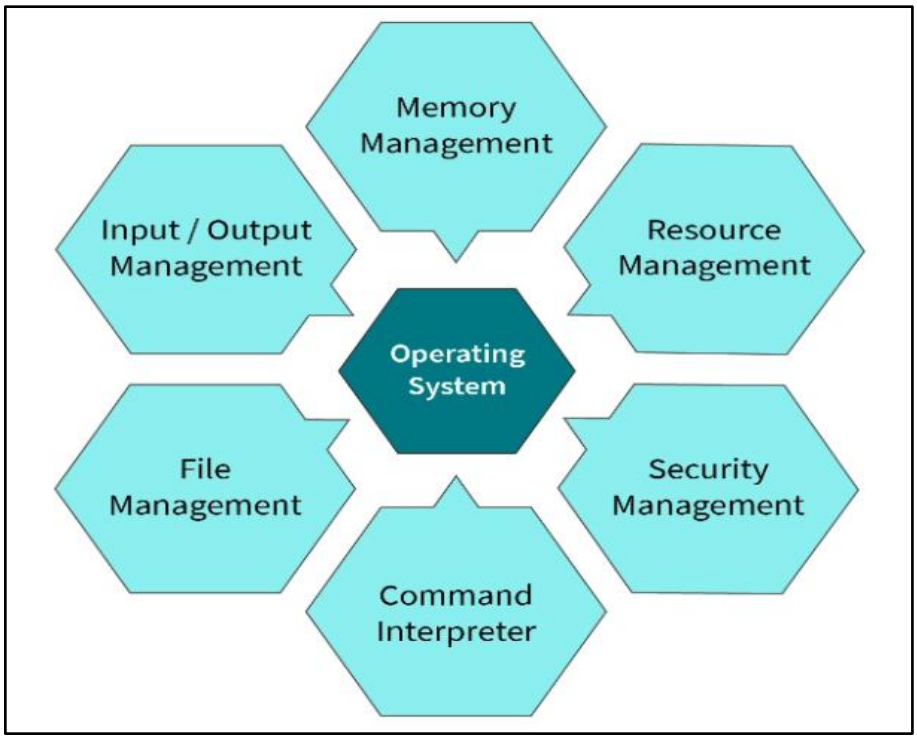


Figure 4.4.2: Disk Management Techniques

Source - https: //www.scaler.com/topics/operating-system/disk-management

Advanced Disk Management and Optimization

Also, advanced disk management encompasses more technical strategies that enhance performance, reliability, and security. The introduction of SSDs has brought both challenges and opportunities for disk management. Wear leveling, garbage collection, and TRIM commands are all functions designed to improve the performance and longevity of the SSD. In order to avoid this premature wear the flash controller needs to implement what is called wear leveling which distributes the write operations among the memory cells. When blocks are no longer needed, they will be freed up through garbage collection, with this process helping to make writes faster. TRIM Command Which Helps SSD to Recover Deleted Data It is accomplished through disk encryption which secures sensitive data by encrypting it prior to writing it to a disk. While full-disk encryption (FDE) encrypts the entire disk, file-level encryption encrypts individual files. When files get compressed, the amount of disk space required to store data storage gets lessened. So that means you turn in the no compression, -- no bzip2, -- and you run through the lossless compression algorithms (gzip, zip). Disk snapshots → create point-in-time versions of the disk. They are implemented using copy-on-write and redirect-on-write and other techniques. Centralized storage solutions, such as storage area networks (SANs) and network-attached storage (NAS), support large environments. SANs use high-speed fiber channel or iSCSI connections for block-level access to storage, and NAS uses Ethernet connections for file-level access. Logical Units (LUNs) for storage are created by these storage virtualization solutions, which abstract their resources. and can create virtual pools of storage to efficiently resize dynamically. Thin provisioned - storage allocated on demand to avoid wasting space. Using storage tiering, commonly used data is automatically placed to faster tiers, like SSDs, while data that is accessed less frequently can remain in slower tiers, like HDDs. AI/ML together is becoming as a powerful method to focus on performance, reliability, and cost of storage infrastructure in organizations.

Emerging Trends and Future Directions in Disk Storage and Management

Constantly innovating itself and evolving with technologies and the data storage needs. New paradigms, such as the adoption of NVMe (Non-Volatile Memory Express) and NVMe-oF (NVMe over Fabrics), persistent memory technology, and the increasing use of cloud-based storage systems, represent the future of storage, Branham added. What is NVMe: NVMe is an interface protocol focused on high-performance SSDs and can achieve much higher data transfer rates than other interfaces such as SATA and SAS. NVMe-oF adds a layer of abstraction to NVMe, allowing NVMe traffic to be sent over network fabrics like Ethernet and Fiber Channel, facilitating high-speed remote storage access.

4.4.3 RAID Structure

RAID (Redundant Array of Independent Disks) is a technology that uses multiple hard disk drives to achieve redundancy and/or performance improvements. Essentially, RAID is designed to increase the reliability and speed of data storage by spreading the data across multiple disks in such a manner that the impact of a single disk failure is minimized. It was first introduced during the late 1980s in an effort to satisfy the demand for both fault-tolerant and high-performance storage in increasingly complex computing environments. RAID levels differ in terms of data distribution and protection. At its most basic level, RAID 0 (striping) splits evenly or by segments of data across two or more disks, allowing simultaneous access that maximizes read and write speeds. That said, RAID 0 provides no redundancy, so the failure of a single disk results in loss of all data. RAID 1 (mirroring): Data is stored on two (or more) disks as a copy for 100% redundancy. Whether you lose one disk, data is still accessible from the other. RAID 1 is known for great fault tolerance, but it halves the available storage capacity because every piece of data is written twice. RAID 5, known as striping with distributed parity, is a balance between RAID 0's speed and the redundancy of having parity information spread across all of the disks. The parity information can be used to reconstruct data in the event of failure of any one disk, which gives a compromise between performance and fault tolerance. RAID 6 (striping with double parity) is like RAID 5, but includes two sets of parity data, meaning it can

recover from two simultaneous disk failures. RAID 10 (also known as RAID 1+0) takes the mirroring and striping approach to combine both high performance and high redundancy. Requires at least four disks, with data mirrored across each of pairs of (2) disks and then striped across the mirrored pairs. RAID 01 (or RAID 0+1) combines striping and mirroring by striping data across disks and mirroring it to another group of striped disks. The one significant difference between RAID 10 and RAID 01 is the order of operations: RAID 10 mirrors then stripes, but RAID 01 stripes then mirrors. Different RAID levels cater to varying applications based on the requirements of performance, redundancy, and the cost. For instance, where database servers use RAID 10 or RAID 5 for best performance and data protection, video editing workstations may use RAID 0 for speed. Along with these conventional RAID levels, there are also some proprietary RAID implementations that provide additional features and capabilities. These approaches might have different flavors of the standard levels, or they might have completely new ways of distributing and protecting data. There are primarily two types of RAID implementations, software RAID, which is based on an implementation from the operating system, and hardware RAID which is based on dedicated, physical RAID controller. Hardware RAID provides a higher level of performance and reliability because the RAID processing is offloaded from the CPU, whereas software RAID is more cost-effective and more flexible. Choosing a RAID level is a decision that balances performance and cost from the perspective of redundancy. Making the most appropriate selection operates based on having a crystal clean insight of the particular non-IT related demands of the usage, in addition to all of the high-level attributes of the RAID amounts on offer. Moreover, new RAID formulations and optimization methods have emerged, due to ongoing changes in storage technology like with solid-state drives (SSDs) and NVMe. They provide far superior performance to legacy hard disk drives (HDDs) and they need different techniques in order to implement RAID. It is also leading to more and more RAID-animal hybrids with SSD and HDD storage as writing in storage can be more costly but would require only a fraction of the speed needed for a read. RAID technology has been sold on many fronts, and the future of RAID will most certainly lead to more seamless integration with new storage methods and technologies and to more advanced data protection

solutions. This will encompass improvements on error correction, prediction of possible failures and automated data recovery systems. It aims at building intelligent storage systems which are self-managed, fault free and performant.

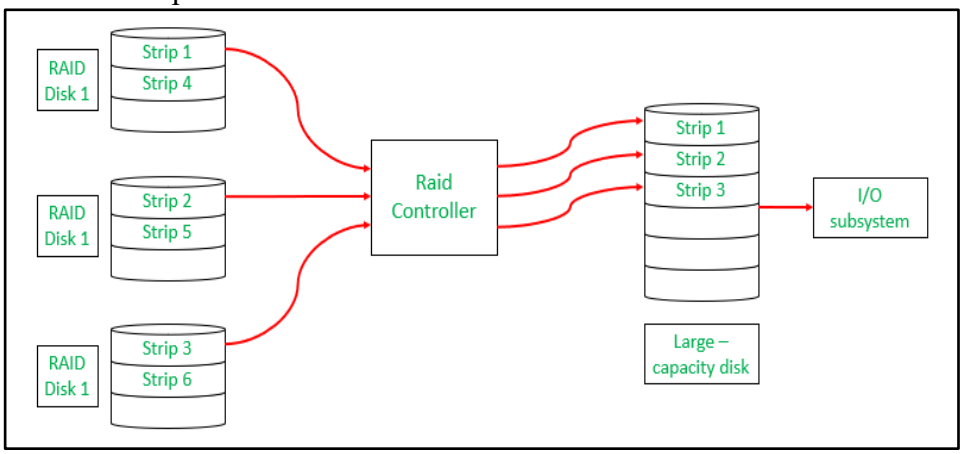


Figure 4.4.3: Raid Controller

Source: RAID (Redundant Arrays of Independent Disks) - GeeksforGeeks

4.4.4 Distributed System Structure

A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. Such computers (or nodes) exchange messages over a network and coordinate their actions. Distributed system's main purpose is to share resources and it achieves scalability, fault tolerance, and improved availability. Distributed systems, in contrast to centralized systems (where a single server processes and stores data), reduce the risk of failure and increase system performance by distributing processing and storage over multiple nodes. While they offer numerous benefits, distributed systems can be challenging to implement and require careful design and management to ensure reliability and efficiency. Distributed systems and their architecture play a vital role in promoting performance, scalability and fault-tolerance. A very common architecture pattern is a client-server where there are clients sending requests to the server in orders to get some services. This is a very common model in web applications where web browsers (clients) serve requests for web pages from web servers. Another architectural pattern you can choose is P2P (peer-to-peer), where all nodes are equal and

have the same role and responsibility. Typically, P2P networks are utilized for file sharing and distributed computing. Another one is Layered Architecture, where the system is organized into layers, where each layer provides a particular range of services. As a result, it encourages modularity and results in a simpler system design. Microkernel architecture, where the operating system kernel provides the fewest number of services necessary and other services run in user space. This architecture expands both flexibility and fault tolerance. In this approach, the operating system itself is distributed among multiple nodes, that is, a more integrated distributed operating system. This alternative offers users a more transparent and seamless experience. Of course, there are things to consider when designing a distributed system, such as communication, synchronization, fault tolerance, and security. Nodes communicate with each other by means of message passing, that can be either synchronous or asynchronous. The former involves synchronous communication; the sender must be willing to wait for a response from the receiver, whereas with the latter, the sender can keep on doing their processing without waiting for a response. Synchronization is vital to managing the functionality of different nodes, so that they act in a coherent and consistent way. This can be done using different mechanisms like distributed locks and consensus algorithms. The tolerance of faults of the system is the ability of the given system to keep working with the failure of the nodes. Redundancy and replication are how this is accomplished. As distributed systems are vulnerable to multiple types of attacks, such as denial-of-service attacks and data breaches, security is also a crucial issue in these systems. Protecting the system and its data requires security measures such as encryption and authentication. The scalability of a distributed system refers to its capacity to manage higher workloads with the addition of nodes. Horizontal scaling adding nodes to the system, or vertical scaling upgrading the nodes' hardware, will allow those storage systems to scale out and handle more traffic. The decision tree for whether to scale horizontally vs. vertically is app-specific. And how do we define the reliability of a distributed system? They do this by the use of techniques such as redundancy, full copies of data, and error correction and detection. Its ability to perform tasks efficiently and effectively is the performance of a distributed system.

Load balancing, caching, and parallel processing are a few techniques that will help- Distributed systems are set to witness advancements in

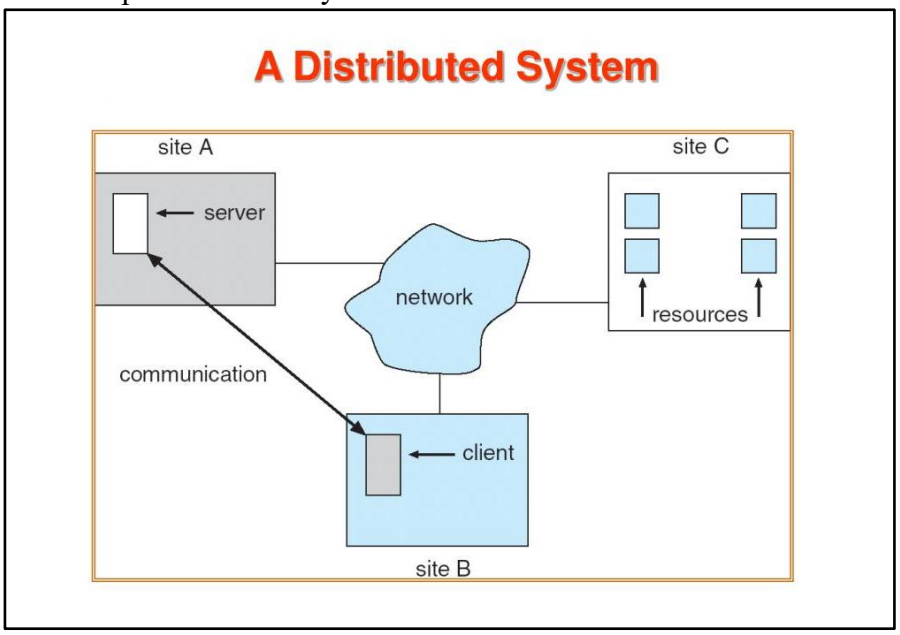


Figure 4.4.5: A Distributed System

Source: https://image2.slideserve.com/3941643/a-distributed-system-l.jpg

the realm of cloud computing, edge computing, and the Internet of Things (IoT) in ongoing future. With data being the operative word, fall of data sizes means the systems need to be adequately sophisticated to manage such data internally or over the network only.

4.4.5 Distributed File Systems (Approx. 1900 words)

A distributed file system (DFS), is a file system that enables clients to access and share files stored on multiple servers over a network as if they are stored on a single, local file system. DFSs are critical for supporting collaboration and sharing in distributed settings. They offer a single namespace so that users can access files without understanding the underlying location of the file. We can find very interesting features of DFS which make it scalable, available, fault tolerance and cover the performance maximally. Scalability The file system's capacity to manage growing volumes of data and user requests. This is done in two ways either adding more servers to the system. Availability: It allows the file system to be still available in the failure of the servers. This is done with replication and redundancy. Fault tolerance refers to the ability of the file system to tolerate errors or failures and continue operating correctly. Data redundancy and error detection and correction

mechanisms are employed to achieve this. Performance defines how fast and efficiently the file system can grant access to the files. It can be performed with caching, load balancing, and parallel processing. A common architecture of a DFS is a client/server model where the client accesses files from the server. The metadata about the files, including things like their names, permissions, and locations, are stored on one or more metadata servers. This is where the actual file data is stored which is on data servers. The metadata servers maintain the namespace and information about where the file is contained, while data servers store and access file data. Network File System (NFS), Andrew File System (AFS) and Hadoop Distributed File System (HDFS) are common DFS architectures. One of the most popular examples of DFS is NFS (Network File System), which enables clients to access files located on remote servers on the network. It uses client-server architecture and provides an easy and efficient way to share a file. AFS is a more advanced version of DFS with added strength, security, scalability, and so forth. It employs a distributed caching mechanism to boost performance. HDFS is a DFS for large-scale data processing. It enhances the performance of the Hadoop Framework and gives a high throughput as well as fault tolerance. With that said, designing a DFS comes with many challenges such as naming, caching, replication, and consistency.

4.4.6 Naming and Transparency Remote File Accesses

Naming and transparency are paramount themes in distributed systems, especially for remote file accesses. These principles provide a way for users and applications to behave as if they were working with files on local disk, to speak with data located on remote servers. At core, naming is about establishing a logical, human-friendly way to identify and locate files in a distributed setting. This includes creating a naming scheme which adds a layer of abstraction between the physical location of the data and the logic used to access it, permitting users to specify a file with a symbolic name, rather than a complex network address. Transparency, in contrast, is the extent to which a distributed nature of the system is hidden from users. A distributed file system should fundamentally attempt to be as transparent as possible, where accessing remote files should seem no different from accessing a local file. This includes location transparency (the user doesn't know where the file

actually resides), access transparency (the same access methods are employed for local and remote files), and concurrency transparency (simultaneous access to a shared file is together without interference from users). The difficulties in working to implement these transparencies are profound: they involve coordinating operations across a collection of machines, isolating the impact of network latencies, and so on, including reintegrating which nodes may fail. The nomenclature schemes used have to be robust, scalable and be able to adapt to the dynamic nature of a distributed environment. These higher-level abstractions are often implemented using techniques such as hierarchical naming (where names to files are organized into logical structures through directories and subdirectories, and through physical access paths) and attribute-based naming (where files are referenced based on their attributes). Moreover, name resolution must also be performed by the system, as symbolic names must translate to physical addresses efficiently.

If you have to implement remote file access systems, I suggest you think about the different design choices you make and the overall performance, scalability, and reliability of the system that you end up with. The right file access protocols is one key factor. They establish the methods of communication between clients and servers, outlining the process of file requests and data retrieval. In file sharing, protocols like Network File System (NFS) and Server Message Block (SMB) are prevalent, each with its unique benefits and trade-offs. For example, NFSis known for its simplicity and platform independence; SMBis frequently used in home windows environments and has strong support for file sharing and printing. Caching strategy is another important design consideration. Caching refers to holding repeatedly accessed information on the machines of the client, eliminating the need to resubmit requests over a network. This can often improve performance markedly, but also brings cache consistency challenges. When the same file is coached by multiple clients, it must ensure that all clients have the latest version. To solve this problem there are some techniques such as cache invalidation, write through cache and others. And, the system needs to deal with fault tolerance. In a distributed system, failures are not a bug; they are a feature. Servers can crash, and networks can be disconnected, and data can be corrupted. The files system needs to be built to endure such failures and should guarantee

that services are available and data is not lost. This can include techniques such as data replication to multiple servers, error detection and recovery mechanisms, and using distributed consensus algorithms to ensure consistency in the presence of failures. Another vital component of accessing files remotely is security. The system will only allow authorized users to expose sensitive data, and must implement access control mechanisms for this purpose. For instance, you might implement authentication protocols to confirm the identity of users, apply encryption to keep data secure while it's being transmitted, and create access control lists that limit what specific users and groups are allowed to do.

The rise of the internet and distributed computing has left an imprint on the development of remote file access. The initial systems emphasized simple file sharing in localized networks. With the increasing prevalence of networks, there was demand for more advanced systems able to operate in large-scale distributed settings. Evolution in remote file access: From e-mails to cloud computing In the 1990s, the use of e-mail grew exponentially. The level of scaling and availability of these services is like never before with the ability to access your data from virtually anywhere on the planet. But they also bring mew risks concerning data security, privacy, and compliance. With the rising data generation and storage, works on efficient ways of data storage and retrieval have also increased. Today's distributed file systems are designed to store petabytes and even exabytes of data, employing techniques such as data striping, erasure coding and distributed hash tables. There are also file access systems for mobile devices that are more adapted to low-bandwidth and intermittent network connections. Offline caching and data synchronization techniques are often employed to ensure that the data served by the application is available even if the user is not connected to the network. This trend toward edge computing, in which data processing and storage are pushed closer to the edge of the network, is also affecting how remote file access systems are designed. Edge computing can help reduce network latency and improve performance by processing data locally. These trends are expected to shape the future of remote file access, which will determine how we seamlessly, securely, and efficiently access data in increasingly complex and distributed environments.

There are numerous, difficult trade-offs to make when you strive for transparency in remote file access. Network latency Fixed by: Implementing zero-trust principles One of the key obstacles is network latency. Data network round trip latency degrades both the network file access response time and the remote file system performance. Many systems approach this through techniques like caching and prefetching that try to predict your data and pull data on your behalf before you actually ask for it. Yet, these techniques also add additional complexity regarding cache consistency and data staleness. The second issue is partial failure. In a distributed environment, you can have failures in some components while the rest are running. If this is not controlled properly it can lead to data inconsistencies and corruption. To do this we use distributed consensus algorithms (like Paxos and Raft) to make sure that all replicas in the system of a file are in sync, even in the presence of failures. These algorithms enable a set of machines to reach consensus on a value, even if some fail. Yet, their implementation may also be rather complex and can incur a performance overhead. Another major concern is security. The remote file access systems must protect the data from unauthorized access, modification, and disclosure. This calls for strong authentication and authorization mechanisms, together with encryption to secure data at rest and in transit. This has led to a great focus towards security in distributed file systems due to the rising frequency of cyber-attacks. Scalability is another important factor to consider. File systems need to scale as they scale to continue to feast on more users and more data. This necessitates thoughtful design of data structures, algorithms and protocols. Sharding (partitioning data across multiple machines) and load balancing (distributing requests among servers) are applied to achieve scalability. Another challenge is the variety of operating systems and devices each client has. Client support: File systems need to interact with a diverse set of clients, which may have varying levels of capabilities and limitations. This may include platform-independent protocols and data formats. the principles of naming and transparency are central to the design and implementation of remote file access systems. High levels of transparency in using Reveal require overcoming many performances, consistency, fault tolerance, security, and scalability challenges. The need for better remote file access has emerged with the rise of distributed computing and the internet and

later cloud computing, which led to the development of solutions for accessing data in new, massive and distributed environments, given the necessary emphasis on access without intervening systems on the data access process, while maintaining security and efficiency in data transfer. It is probable that some of the trends that will either directly or indirectly define the future of remote file access will include edge computing, mobile computing, and the growing volume of data, as there will be a need to build even more intelligent and adaptive systems. The continued evolution of new technologies and protocols will further enhance the performance, reliability, and security of remote file access, allowing users to access their data from anywhere and at any time.

Summary

Disk scheduling and distributed systems are vital components in the operation of modern computing environments. Disk scheduling refers to the method by which operating systems decide the order in which read and write requests to the disk are processed. Since multiple processes may request access to a disk simultaneously, an efficient scheduling algorithm ensures optimal disk utilization and reduced seek time. Common algorithms include FCFS (First Come First Serve), SSTF (Shortest Seek Time First), SCAN, C-SCAN, and LOOK, each offering different trade-offs in terms of fairness, speed, and complexity. These algorithms aim to reduce the movement of the disk's read/write head and improve the response time for processes.

I/O hardware forms the interface between the system and external devices. It consists of components such as device controllers, buses, and ports that facilitate communication and data transfer. The operating system interacts with I/O hardware using device drivers and I/O control methods like polling, interrupts, and Direct Memory Access (DMA). Efficient I/O handling is essential for system performance, as it minimizes the time the CPU waits for data input or output operations.

Disk structures define how data is organized and accessed on the physical storage medium. This includes sectors, tracks, cylinders, and platters. Understanding the physical structure of disks helps in designing better disk scheduling algorithms and optimizing file systems. In distributed systems, where resources and data are spread across multiple networked computers, coordination and consistency are crucial. Distributed systems aim to provide users with a seamless

experience of a unified system while handling complexities like data replication, synchronization, and fault tolerance in the background. Together, disk scheduling and distributed system principles ensure that storage and data access are managed efficiently in both local and networked environments.

Multiple-Choice Questions (MCQs)

1. Which of the following is NOT a disk scheduling algorithm?
    a) First-Come, First-Served (FCFS)
    b) Shortest Seek Time First (SSTF)
    c) Round Robin (RR)
    d) SCAN

    (Answer: c)
2. Which component is responsible for managing input and output operations in a computer?
    a) CPU
    b) I/O Controller
    c) Cache Memory
    d) Registers

    (Answer: b)
3. What is the purpose of an I/O interface?
    a) To facilitate communication between the CPU and storage devices
    b) To execute user programs
    c) To process high-priority interrupts
    d) To store temporary data

    (Answer: a)
4. Which of the following is a primary function of disk management?
    a) Process scheduling
    b) Memory fragmentation
    c) Formatting and partitioning disks
    d) Program execution

    (Answer: c)
5. Which RAID level uses striping without redundancy?
    a) RAID 0
    b) RAID 1
    c) RAID 5

d) RAID 10

(Answer: a)

6. What is the key characteristic of a distributed system?
    a) Centralized control over all processes
    b) Multiple independent processors working together
    c) Use of a single file system for all devices
    d) Only local execution of processes

    (Answer: b)
7. Which of the following is NOT an advantage of a distributed file system?
    a) Scalability
    b) Data redundancy
    c) Single point of failure
    d) Remote file access

    (Answer: c)
8. What is transparency in a distributed system?
    a) The ability to hide implementation details from users
    b) A mechanism for encrypting data
    c) The process of data fragmentation
    d) A technique for improving network latency

    (Answer: a)
9. Remote file access allows users to:
    a) Access files stored on a local disk only
    b) Retrieve and modify files stored on another system over a network
    c) Use physical hard drives instead of cloud storage
    d) Remove files permanently from all servers

    (Answer: b)
10. Which disk scheduling algorithm favors the request closest to the current head position?
    a) FCFS
    b) SSTF
    c) LOOK
    d) C-SCAN

    (Answer: b)

Short Questions

1. What is the purpose of disk scheduling in an operating system?

2. List two common disk scheduling algorithms and briefly explain them.
3. What is an I/O interface, and why is it important?
4. Explain the basic structure of a hard disk.
5. What is the function of a RAID system, and why is it used?
6. Differentiate between RAID 0 and RAID 1.
7. What are distributed systems, and how do they improve computing efficiency?
8. Define naming transparency in a distributed file system.
9. What is remote file access, and how does it benefit users?
10. How does a distributed file system differ from a traditional file system?

Long Questions

1. Explain the need for disk scheduling and discuss different disk scheduling algorithms.
2. What are the key components of I/O hardware, and how do they function?
3. Discuss the applications of an I/O interface in operating systems.
4. Explain the structure of a hard disk and its role in data storage.
5. Compare different RAID levels and their advantages and disadvantages.
6. What is a distributed system, and how does it improve resource utilization?
7. Discuss the features and architecture of a distributed file system.
8. Explain the concept of naming transparency and its importance in distributed systems.
9. How does remote file access work, and what are the security concerns associated with it?
10. Analyze the challenges in implementing distributed systems and how they can be overcome.

MODULE 5
STATEFUL VERSUS STATELESS SERVICE AND SHELL PROGRAMMING

LEARNING OUTCOMES

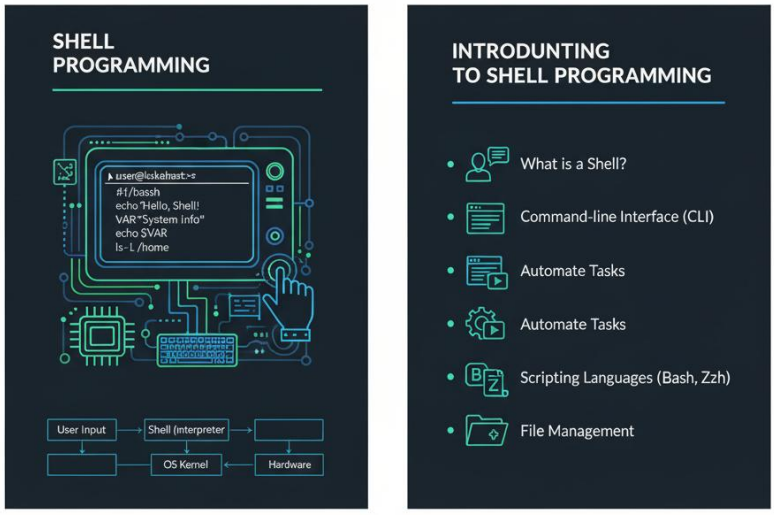
- To understand stateful and stateless services in OS.
- To explore different shell programming techniques.
- To study command execution processes and shell scripting.
- To analyze decision-making selections and function parameter passing in shell scripts.

Unit 5.1: Shell Programming & Introduction to Shell Programming

5.1.1 Shell Programming & Introduction to Shell Programming

In the realm of operating systems, particularly Linux and Unix, the shell is a crucial command-line interpreter, bridging the gap between the user and the kernel, the core of the operating system. It allows users to interact with the computer by executing commands, managing files, and controlling system processes via a text-based interface. Shell programming is simply writing the shell command language scripts for repeating the tasks and is used to create powerful utilities. Essentially, the shell is a command line interpreter which takes commands from the user and translates them into instructions that the kernel can comprehend and execute. This feature is not limited just to running a single command; all in one and you can write complex scripts to automate repetitive tasks, manage the system configuration, and process data in a complex structure. By combining, controlling execution, and calling various available commands or built-in functionality, the shell is very powerful and flexible. The basics of shell programming using various shells are taught as an essential aspect of the concepts of operating system development by undergraduate students, highlighting their significance in system administration as well as automation. The shell environment gives students a direct view into the inner workings of the operating system: they can experiment with system commands and see their effects firsthand, gaining a hands-on understanding of how the operating system works. Students pursuing careers in computer programming, software development, and systems administration gain important hands-on experience. Users write shell scripts that are a single file combining multiple commands, automating complex workflows and eliminating manual steps. Tools and Utilities: You can write shell scripts to create custom utilities and tools that extend the OS's capabilities, enabling users to customize their environment according to their specific requirements. Additionally, shell scripting offers a programming environment with access to variables, control flow (loops and if statements), and functions, making it a powerful medium for writing complex programs. With the help of variables, users can store and manipulate data within scripts, and with control structures the flow of execution can be controlled based on

specific conditions. With functions, a user can encapsulate reusable code blocks, promote modularity and facilitate code reuse. Shell also has a rich set of built-in commands and utilities like file manipulation,



references, and system administration commands which you can use inside the scripts. All these built-in functions, along with the shell's ability to use scripts, make it a powerful platform for building all sorts of applications. Shell programming is a skill set that is fundamental and a necessary basic building skill for more advanced aspects of programming. Learning to write shell scripts teaches students critical skills such as the ability to solve problems, think through their logic, and break complex tasks into smaller, more readable actions. Students aiming to be proficient programmers and system administrators require this hands-on learning.

Shell programming is the foundation of understanding how to use commands. The command language of the shell is made to be predictable and simple to Joomla, its key strengths being simplicity and versatility. Commands are usually specified as a command name, its options, and its arguments. If options customize how a command runs, arguments define what data or files the command manipulates. The -l option lists the contents in a long format; so, for one example, ls lists the contents of a directory. Structured... Shell scripts are usually written in some text editor and saved with a. sh extension. The first line of a shell script contains a command that indicates which shell interpreter to use to execute the script, usually #! /bin/bash for the Bash shell. This is called the shebang, it informs the operating system that

the script uses the interpreter that follows it. The shell script you can call with a simple command like shellscriptname, and it will execute commands sequentially, and any command can take the output of another command using pipes and redirection. Pipes enable the output of one command to be passed as input for another, while redirection enables you to redirect the input and output of a command to files or other devices. Shell programming variables: Variables are used to store data in a shell programming script. Values are assigned to variables using the = operator, while values can be accessed using the $ symbol. Built-in Shell Variables: The shell also has a set of built-in variables that provide information about the shell environment, such as the current working directory, and the user's login name. Control structures, (if statements, for loops, etc.) these are used by users to execute flow control in scripts. If statements are used to execute different commands based on specific conditions, and for loops that enable users to iterate over a set of values or files. Functions allow users to wrap reusable code sections, fostering modularity and code reuse. Undergraduate students, within their learning and understanding of these basic concepts can also start building their own shell scripts to automate some of their tasks and processes. The big power of shell programming being able to connect all other programming languages and tools together. Some of the programs could be written in other languages, such as C, Python, and Perl, and shell scripts could invoke these programs and also pass data from one program to the other. Since shell can integrate other utilities, it is suitable for building complicated applications/systems. A shell script, for example, can be used to compile and run a C program, or to manipulate data produced by a Python script. Shell scripts can mix in with other UNIX and Linux commands to carry out basic tasks or perform more complex actions. Shell Environment for Debugging and Troubleshooting (ShellNamespaces.com) The set command will enable debugging options such as command execution tracing and variable value output. Here at SCRIPT execution the value of the messages and the value of the variables get displayed with the help of echo command, these are very helpful to understand errors and find the bugs. Command-line debuggers, like bashdb, are also supported with shell programming and offer advanced debugging capabilities, including but not limited to breakpoints, stepping, and variable inspection. These debugging tools

help undergraduate students to learn how to write solid and dependable shell scripts. System administrators also benefit from shell programming since they use shell scripts to automate routine tasks such as system maintenance, managing user accounts, and monitoring system performance. Shell scripts also enable administrators to build custom tools and utilities that can be used for system administration tasks, making it easier for them to work efficiently. As I mentioned earlier, a shell script can be used to automate some tasks such as creating user accounts, installing software packages, or copying system files. Finally, the study of shell programming is important for undergraduate students aspiring to build careers in computer science and its allied fields. It is a command line shell that serves as a powerful and flexible environment to automate tasks, to customize the operating system and to integrate with other programming languages and tools. Students learn the command line and write their first shell program, without any knowledge, become the groundwork for learning about operating systems and basic building blocks of sysadmin and automation. If you don't know how shell scripting sessions work, that's fine but you should, because writing scripts isn't enough to be a good shell programmer. The shell is a command line interpreter which allows the user to input command to manage the processes, files and configurations on the system. Shell programming is an important tool for both system administrators and developers as it has a close link with the operating system. Moreover, the shell provides advanced scripting features that allow you to combine multiple commands into a sequence of actions. The versatility of the shell comes from its ability to compose existing commands, control the flow of the program, and to leverage its rich set of built-in capabilities to manipulate data. Shell programming also exposes students (primarily in their undergraduate curriculum) to a programming paradigm which they can extend into other languages as they learn them. These types of programming quizzes can help students practice their problem-solving skills, as shell scripting requires not only knowledge and skills IT but they know how to put it to use. This hands-on experience is critical for students who want to become competent programmers and system administrators. The Shell Shells are essential because they enter every organization with X applications. Shell programming allows for building complex systems, from automating software product development workflows

via web servers and databases. With the advancement of technology, the need for skilled shell programmers will only rise; it is an essential skill for students to learn.

5.1.2 Various Types of Shells and Their Comparisons

Many more shell implementations were developed over the years, each featuring different syntax and capabilities, targeting various user bases and needs. These different types of shells play a significant role in system administration and software development. Purely repercussive shells including the Bourne shell (sh) were the very early shells whilst targeting simplicity and efficiency, looking only towards basic command execution and scripting functionality. The syntax of the Bourne shell, while perhaps not as powerful as its successors, served as the blueprint for the development of future shells. With advancements in computing, users began to have different needs, and thus, more advanced shells were created that had better features and functionalities. You might also implement a more interactive feature such as command history, job control, and aliases with csh, etc. Its C-like syntax attracted C users, though the C shell's scripting capabilities had received complaints as inconsistent and limited. David Korn wrote the Korn shell (ksh), which attempted to merge the best features of the Bourne and C shells in an interactive and scripting environment. It also added command-line editing, job control improvements and many other features that made it popular among system administrators. The Bourne-Again shell (bash) is an improved version of the original Bourne shell that adds many features from the Korn shell and C shell and is the default shell for most Linux distributions. Bash has numerous more advantages and options for the interactive user and the script writer, together with command-line completion, history growth, and plenty of scraping choices. This popularity is due to its ability to run Bourne shell scripts, as well as its extensive feature set and availability. In addition, yet another popular shell is the Z shell (zsh), which is built on top of bash to provide advanced features like advanced command-line completion, spell correction, and plugin support. Make Zsh Your Own (and Others Again) Zsh is highly customizable and extensible, which is why it is loved by power users and developers. Now when it comes to comparing these shells, things like syntax, scripting capabilities, interactive features, and

customization options come into play. With simplicity and efficiency, the Bourne shell didn't offer many user-friendly features introduced in later shells. While the C shell is interactive, it isn't that great for scripting. The Korn shell strikes a good balance between interactive and scripting features, while bash and zsh have plenty of features aimed more at interactive use with the script features there too. The normal shell to use is a matter of preference this leads to bash and zsh being the most popular and recommended for use due to the large set of features.

5.1.3 Command Execution

Here are the steps of command execution in a shell: parsing the command line, executing the corresponding command, collecting the result. The shell is the command-line interface that is responsible for processing user input. It expands. The shell makes lots of expansions variable expansion, tilde expansion, wildcard expansion, and so on to fix any special characters or variables in the command line. Variable expansion will put the value of variables instead of the variables in command, which allows us to build a command dynamically. Tilde expansion refers to the opening of a user account in this directory using the tilde character (~) for ease of use, so that users do not have to write out the full path to the user's home directory when the file or directory is in the home directory. Using Wildcard expansion means expanding the patterns using Wildcards like and?, thus allowing a user to perform operations and actions on multiple files with a single command. Then, after the parsing and argument expansion the shell checks whether the command executed is a built-in command or an external command Built-in commands include the commands that are implemented in shell itself, eg commands like cd, echo, exit etc. The commands here are run in the shell itself without invoking a new sub process. External commands refer to program residing on a file system like ls, grep, and gcc. When an external command is executed, the shell forks a new process calling fork system call and the image of the new process is then replaced with the specified program using the exec system call. Creating a child process through the fork system call including a new process replacing its memory through the exec system call The shell also waits for the process to terminate by using the wait system call once the program executes. Next, the shell collects the exit status of

the process, which tells it if the command successfully executed or if an error occurred. Input Output Redirection (Using and) Shell uses special characters like and to redirect input and output. The input is read from a file using input redirection Used to send the output from one command to another command as input, which gives the user the ability to elaborate commands and create complex operations. Another topic which is essential to command execution is job control, allowing users to run multiple processes at one time. The shell has commands like bg, fg, and jobs to manage the transitions between foreground and background processes, and to list background jobs that are currently running. In addition to handling separate processes, the shell also handles signals, which are messages sent to any process to notify it of asynchronous actions like interrupts, termination requests, and errors. The shell includes commands such as kill that are used to send signals to processes, enabling the user to kill or otherwise control their behavior. The understanding of these steps and functionalities is required to work with command line in an efficient manner as well as scripting.

The shell, as the command-line interface (CLI), is the fundamental program responsible for processing user input and orchestrating the execution of commands. It acts as an interpreter, translating human-readable instructions into actions the operating system can understand. Understanding the steps involved in command execution is crucial for efficient command-line usage and effective shell scripting.

Step 1: Parsing the Command Line and Expansions

When a user types a command and presses Enter, the shell doesn't immediately execute it. Instead, it first parses the command line, breaking it down into individual components (command and arguments) and then performs various **expansions**. These expansions replace special characters and variables with their actual values, constructing the final command string that the system will interpret.

1. **Variable Expansion**: This is one of the most common expansions. When the shell encounters a variable (e.g., $HOME, $PATH, or user-defined variables like $my_var), it replaces the variable name with its stored value. This allows for dynamic command construction, where parts of a command can change based on the environment or user input.

Example: echo "My home directory is $HOME" will replace $HOME with /home/username (or /Users/username on macOS).

2. **Tilde Expansion**: The tilde character (~) is a convenient shortcut for a user's home directory. When the shell sees a ~ at the beginning of a path, it expands it to the full path of the current user's home directory (e.g., /home/username). This saves typing and makes commands more portable.

Example: ls ~/documents is expanded to ls /home/username/documents.

3. **Wildcard Expansion (Globbing)**: Wildcards are special characters that allow users to specify patterns for multiple files or directories. The shell expands these patterns into a list of matching file names before the command is executed.

- * (asterisk): Matches any sequence of zero or more characters.

Example: ls *.txt expands to ls file1.txt report.txt data.txt

- ? (question mark): Matches any single character.

Example: mv file?.log expands to mv fileA.log fileB.log

- [ ] (brackets): Matches any one of the characters enclosed within the brackets, or a range of characters.

Example: rm [abc]*.tmp expands to rm afile.tmp bdata.tmp cdoc.tmp

After these expansions, the shell has a fully resolved command and a list of arguments ready for execution.

Step 2: Determining Command Type and Execution

Once the command line is parsed and expanded, the shell determines whether the command is a **built-in command** or an **external command**. This distinction is critical because it dictates how the command is executed.

1. **Built-in Commands**: These are commands that are an integral part of the shell itself. They are implemented directly within the shell's executable code and do not require a separate program to be launched.

Examples: cd (change directory), echo (print text), exit (terminate shell), pwd (print working directory), source (read and execute commands from a file in the current shell context).

Execution: When a built-in command is encountered, the shell executes it directly within its own process. This makes built-ins very fast as they avoid the overhead of creating a new process.

2. **External Commands**: These are executable programs or scripts that reside as separate files on the file system. They are not part of the shell's internal code.

Examples: ls (list directory contents), grep (search text), cat (concatenate files), gcc (GNU C Compiler), python (Python interpreter).

Location: For the shell to find an external command, its location must be specified in the PATH environment variable. The shell searches through the directories listed in PATH (e.g., /usr/local/bin, /usr/bin, /bin) to find the executable file.

Execution Process (Fork and Exec):

Fork: When an external command is to be executed, the shell makes a system call named fork(). This fork() system call creates a **new, exact copy of the current shell process**, known as a **child process**. This child process inherits most of the parent shell's environment, including open file descriptors, environment variables, and current working directory.

Exec: Immediately after the fork(), the child process makes an exec() system call (e.g., execve()). The exec() system call replaces the entire memory image of the child process with the program specified by the external command. This means the child process stops being a copy of the shell and becomes the new program (e.g., ls). The exec() call does not create a new process ID; it replaces the current process with a new program.

Wait: Meanwhile, the original parent shell process (which created the child) typically makes a wait() system call. This wait() call causes the parent shell to pause its own execution and wait for the child process (the executed command) to complete.

Termination: Once the child process (the external command) finishes its execution, it exits. The parent shell, which was waiting, then resumes its own execution.

Step 3: Collecting the Result (Exit Status)

Upon the termination of a command (whether built-in or external), the shell collects its **exit status**. The exit status is an integer value that communicates the success or failure of the command.

- By convention, an exit status of **0 (zero)** indicates successful execution.

- Any **non-zero** exit status (e.g., 1, 2, 127) indicates that an error occurred or the command terminated abnormally. Different non-zero values often correspond to specific types of errors. The exit status is stored in a special shell variable, $?, which can be checked by scripts to control their flow based on the outcome of previous commands.

Advanced Functionalities: Redirection, Pipelines, Job Control, and Signals

The shell offers powerful features that extend simple command execution to enable complex operations and process management.

1. **Input/Output Redirection**: The shell uses special characters to alter where a command reads its input from or sends its output to.

Output Redirection (>, >>):

- >: Redirects a command's standard output to a file, **overwriting** the file if it already exists.
- >>: Redirects a command's standard output to a file, **appending** to the file if it already exists.
- **Example**: ls -l > file_list.txt (sends the ls output to file_list.txt)

Input Redirection (<): Redirects a command's standard input to read from a file instead of the keyboard.

- **Example**: grep "pattern" < input.txt (reads input for grep from input.txt)

Error Redirection (2>): Redirects standard error (stderr) to a file.

- **Example**: command_that_might_fail 2> error.log

2. **Pipes (|)**: The pipe operator allows the output of one command to be directly fed as the input to another command. This enables chaining commands to perform sophisticated operations.

Example: ls -l | grep ".txt" | sort (lists files, filters for .txt files, then sorts the result)

3. **Job Control**: This functionality allows users to manage multiple processes concurrently, moving them between the foreground and background.

- **Foreground Process**: A process actively interacting with the user, receiving input from the terminal.
- **Background Process**: A process running independently in the background, not requiring immediate interaction, freeing up the terminal for other commands.
- **Commands**:

- &: Runs a command in the background immediately.
- Ctrl+Z: Suspends the current foreground process.
- bg: Resumes a suspended process in the background.
- fg: Brings a background or suspended process to the foreground.
- jobs: Lists all currently running or suspended background jobs.

4. **Signals**: Signals are software interrupts or asynchronous notifications sent to processes to inform them of events. The shell allows users to send signals to manage process behavior.

- **Common Signals**:
- SIGINT (Interrupt - Ctrl+C): Typically terminates a process.
- SIGTERM (Terminate): Requests a process to terminate gracefully.
- SIGKILL (Kill): Forcibly terminates a process (cannot be caught or ignored by the process).
- SIGHUP (Hangup): Often used to signal a process to reload its configuration.
- **kill Command**: The kill command is used to send specific signals to processes, identified by their Process ID (PID).
- **Example**: kill 9 12345 (sends SIGKILL to process with PID 12345)

Mastering these steps and functionalities provides a robust foundation for effective command-line interaction and advanced shell scripting, transforming the shell from a simple command runner into a powerful environment for system administration and automation.

5.1.4 Detailed Breakdown of Command Execution Processes

In order to do more details about command execution, we need discuss what happens behind the scene. The shell's parser goes to work as soon as a command is typed; breaking down the input into its parts: the command name, any arguments, and options. Parsing is a critical stage for the shell to know the user's intention. After parsing, the shell begins a sequence of expansions to convert the command line to its equivalent executable form. The heart of Bash, variable expansion replaces variables with their assigned values, enabling the flexible crafting of commands. Variable DIR is set to /home/user/documents so when command cd $DIR runs it is substituted into of cd /home/user/documents before executing. Tilde expansion is a shortcut for navigating and designating files, converting ~ to the user's home directory. Another powerful feature of the shell is wildcard expansion, which lets you apply operations to multiple files based on a pattern. For instance, ls. Txt will show the output of all files with. in the

current directory. Hash table or after expansions, the shell determines whether the command is built-in Unix like operating system command execution is an essential concept that enables all user interactions and system activities. Once a user provides a command to the shell, it goes through a complex sequence of processes that converts the command to actions that can be executed. First, the shell parses the command line, splitting it into separate tokens, like the command name and its arguments. This parsing includes anything from interpreting special characters, to quote-handling to wildcard expansion. The shell also aliases, allowing users to define custom command line shortcuts to possible forward to common command lines. After that, the shell looks for built-ins, commands that are built into the shell itself, like cd, echo, or exit. In case the command is built-in, the shell executes the command itself, so no new process needs to be created. Then if the command is not a built-in, the shell searches the directories in the PATH environment variable for an executable file of this name. PATH is a colon-separated list of directories that the shell looks through, in turns. The shell forks a new process using the fork system call when it has located the executable file. The child process subsequently invokes the exec set of system calls to overlay its image with the executable file of its command. The parent shell process, on the other hand, invokes the wait() system call to block whilst the child process executes. The input and output streams are controlled via file descriptors when the command is executed. Standard input (stdin), standard output(stouts)and standard error(stderr) are usually attached to the terminal, but they can be redirected to files or passed to other commands. The shell also handles environment variables (key-value pairs that hold relevant information to processes). These variables can affect the way commands execute and are passed through to sub processes. After executing the command, the child process exits, providing an exit status reflecting success with zero or an error with a positive integer. The parent then displays its prompt, awaiting the next command. From parsing the command line to managing input/output and environment variables, the shell orchestrates this entire process, and acts as the main interface between the user and the kernel of the operating system. Therefore, for the effective fulfillment of tasks of a system administrator and shell programmer, it is important to have knowledge of this process to understand the very process of command

execution and control over it. In this context, shell programming, the act of developing scripts that automate and enhance the capabilities of the command-line interface, is an incredibly powerful tool for both system administrators and developers. Different shells, like Bash, Zsh, and Ksh, offer different levels of features and syntax, with their respective strengths and weaknesses. Bash (Bourne Again SHell) is a widely used Linux shell that is the default shell in many distributions, and it is also the most commonly used shell for writing scripts. You can use variables, conditional statements, loops, and functions within Bash scripts to create complex automation routines. Bash variables are dynamically typed and can hold strings, numbers, or arrays. Flow Control; Uses conditional statements for decision making (if, elif, else) Sequence, selection, and repetition: The sequence section specifies a list of commands to execute one after the other, while conditional execution (via an if statement) allows for decisions to be made in the flow of code, and for loops, such as for and while, permit commands to be executed in repeat for a number of times, automating tasks that would otherwise require manual intervention. Bash functions help in making the scripts modular, organized, and reusable. Another popular shell is Zsh (Z Shell), which extends many of the features found in Bash and also offers better tab completion, spell checking, and theming. Zsh is highly customizable and configurable, making it great for users that want to make their shell environment suited to their specific freedoms. Enter Ksh (Korn Shell) a shell that merges the best of Bourne shell and C shell, providing an efficient and powerful scripting environment. These two features make ksh widely used in terms of performance while retaining compatibility for older shell scripts. Another thing to keep into consideration when writing shell scripts is that there are best practices—using comments to explain what the code is doing, using proper variable names, error handling, and so on. Error handling is done by conditional statements and the trap command, which enables the execution of certain commands on receiving certain signals. Shell also communicates with the operating system by making system calls or executing any external commands.

Unit 5.2: Shell Programming in Different Shells

5.2.1 Shell Programming in Different Shells

The script uses grep to search for pattern(s) in files, or use sed to perform basic text transformations. Automation alone is just a small part of writing shell scripts, but to add more utilities that help within the command-line interface. Learning shell programming can help users automate tasks, understand the inner workings of the shell, and improve their productivity. Now a further look into shell programming in one of the most popular shell, Bash shows a lot of rich functionality for automating complex tasks. A bash script starts with a shebang line, Already, the first line starts with /bin/bash, which refers to the interpreter that is used to run the script. Bash does not require explicit types when declaring variables, which are referenced using the prefix. syntax is used for arithmetic operations, and various built-in commands or parameter expansions are used for string manipulations. Conditional statements including if, elif, and else are used to make decisions based on whether an expression evaluates true or false in Bash. These may be comparisons of strings, numbers, or file properties. Bash loops: for, while and until loop in Bash allow you to run the same command multiple times. The for loop is especially handy to iterate over collections of items; whereas while and until loops are used to iterate conditionally. In Bash, you define a function by writing the keyword function, or by writing the function name followed by parentheses. Makes up the arched arguments and return values that enable modular and reusable code. One of the core parts of bash scripting is the input and output redirection. The operator writes standard output to a file, and the operator appends standard output to a file. Pipes Let's us link commands together so that the output from one command is the input to the next command. Bash error handling can be done using conditional statements and trap command. The command trap enables us to execute some specific commands whenever we receive some signals such as, SIGINT (interrupt), SIGTERM (terminate) etc. It offers several built-in commands like grep, sed, awk, and cut for text processing and data manipulation. Together with its scripting capabilities, Bash can easily be one of the most powerful tools in automating work and managing systems. Learning what Bash scripts are and why they matter are key for any Linux or Unix-like operating

systems user looking to write automation that is as efficient and effective as possible.

While Bash has wide availability, features unique to Zsh and Ksh showcase the variety of shell programming. Zsh provides more powerful interactive features like improved tab-completion, spelling correction and a powerful theming system. That is not all, the tab completion in Zsh is context-sensitive, suggesting commands depending upon the type of command and the arguments being passed against them. Say goodbye to typing long command names and file paths, this feature improves our productivity considerably. So, user's after most time are looking for more spelling verification, Zsh automatic corrects typos for command names and file paths, making the interactive experience better. Zsh theming feature provides customizing functionality to alter the look and feel of the shell prompt and surrounding components. In addition to this, Zsh offers advanced scripting features like arrays and associative arrays alongside regular expressions, which makes it an excellent tool for automating complex tasks. Another powerful shell that combines features of a Bourne shell and a C shell is Ksh, the Korn Shell. Ksh does, however, have great performance and compatibility with older shell scripts. Ksh supports functions, arrays, arithmetic operations and all the other bells and whistles a programming language would have. Ksh also supports powerful features like co-processes, enabling commands to run simultaneously. This is Ksh as it plays along with legacy Bourne shell scripts popular with many system admin type users. Shell is broadly categorized into two different ways, which is, one is Zsh and Ksh that have some syntax and file features. Hope this answers your question while most of the fundamental features and syntax (for example: variables, conditional branching statements, looping constructs) are similar albeit with minor variations in all of these shells, the differences can be subtle enough to mess up the behaviour of your scripts. For instance, Bash's syntax for arithmetic operations, or array manipulations, is different from Ksh and Zsh. The aim of this article is to explore these differences and ultimately to write portable and compatible shell scripts, Zsh has many interactive features that make it a better choice for interactive work, while Ksh may perform better for system administration tasks. This can help users to expand their shell programming toolkit and pick the individual shell that may serve them

best. To summarize, command execution process and shell programming are the integral concepts of Unix and Unix-like operating systems. The shell remains the primary interface between the user and the kernel, handling the actual execution of commands and overseeing input/output and environment variables. Shell programming, the art of writing scripts that automate tasks and extend the utility of the command-line interface, is a powerful weapon in the arsenal of any system administrator or developer. There are different shells like Bash, Zsh, and Ksh with different levels of both features and syntax. Here we will focus on "Bash" the popular default shell used on many Linux distributions. The parameter passing is the mechanism in which the values are sent from a function to its caller. When a function is invoked, its caller has to provide the values that will be passed as arguments to the function to perform its operation. The two main forms of parameter passing are pass-by-value and pass-by-reference. Pass by value is when you pass a copy of the value of the argument to the function. The parameter is a local variable within the function that refers to the same object in memory as the argument passed when the function is called. This approach is used when the function needs a copy of the data to work with and is not going to modify the original. This gives the caller more control over their data, providing a degree of safety by preventing unplanned side effects, since the callee never has access to the original data. Instead, pass-by-reference passes the address in memory of the argument being passed to the function. Any modification to the parameter inside the body of the function modifies the argument in the caller. This is required when the function may need to update the data inside caller or typically used for large data structures where copying it would be costly. Pass-by-reference enables functions to alter multiple values and to produce results via their parameters. But it also has the potential for unintended side effects: If the function changes the caller’s data in an unexpected way. Pass-by-constant-reference is a similar variation some programming languages do offer this is when the function can access the caller's data, but there is no ability to modify it. This gives the performance of pass-by-reference, but the data protection of pass-by-value. Many times, you need to decide if you want to pass-by-value or pass-by-reference. If you want a function to be able to change the data from the caller, then pass-by-reference is the way to go. If the function only needs to work

with a copy of the data (for example, if it is going to mutate it), then you should use pass-by-value as it is a lot to pass the data structure as a reference. It is really important to know how args are passed because it matters for writing efficient and correct code passing a parameter incorrectly causes obfuscated bugs which are hard to find and correct. For instance, passing a complex data structure by value incurs an overhead in performance because of the copying process. Just as we can accidentally modify the caller's data by passing the variable by reference manually, we can do this just as easily by passing it by a default value. Parameter passing is not confined to primitive data types; it is also relevant for complex data structures, including arrays, objects, and pointers. Similar rules apply for when passing arrays or objects, however may differ from language to language. Some programming languages may pass arrays by reference, while others may use relinquish via value by default. Details of our parameter passing are also important in function interface designs Developers can write flexible and robust functions by carefully selecting the correct parameter passing strategy. They are able to design reusable components that can be easily integrated into different parts of a program. Long story short, parameter passing is a fundamental concept in programming that enables functions to communicate with their callers.

Shell programming, or scripting, is the process of writing commands in a shell to automate tasks, perform administrative functions, and create powerful command-line tools. While all shells serve as an interface to the operating system, different shells offer unique features and syntax, catering to different user needs. The choice of shell often depends on whether the primary goal is robust scripting, interactive use, or a balance of both.

Bash (Bourne-Again Shell)

Bash is the most widely used shell on Linux and macOS, serving as the default shell for many distributions. It's an enhanced version of the original Bourne Shell and is **POSIX-compliant**, which means scripts written for it are highly portable across different Unix-like systems. This makes Bash the de facto standard for general-purpose shell scripting.

- **Key Features:** Bash excels at robust scripting. It supports a comprehensive range of control flow statements (if-else, for

loops, while loops), functions, arrays, and associative arrays. It also includes useful interactive features like command history and command-line editing.

- **Syntax:** Its syntax is well-established and powerful. Variables are declared without special characters (e.g., my_var="hello") and accessed with a dollar sign ($my_var). Conditional expressions often use [ ] or [[ ]].

Example Bash Script:

Bash

```
#!/bin/bash

echo "Hello, what is your name?"

read name

if [ "$name" == "Bash" ]; then

   echo "Welcome, mighty shell!"

else

   echo "Hello, $name."

fi
```

Zsh (Z Shell)

Zsh is a modern and highly customizable shell that builds on the features of Bash. While it is largely **Bash-compatible**, it introduces significant improvements that make it a favorite for interactive use. Zsh's powerful features have led it to become the default shell on macOS since Catalina.

- **Key Features:** Zsh's primary appeal lies in its interactive enhancements, such as intelligent and extensive **tab completion** for commands and file paths, built-in spell correction, and advanced **globbing** (wildcard expansion). It also has a more powerful history command. The most notable feature is its vibrant ecosystem of plugins and themes, particularly through frameworks like **Oh My Zsh**.
- **Syntax:** Zsh's scripting syntax is very similar to Bash, making it easy for Bash users to transition. However, it offers some

advanced features, such as for loops over C-style syntax and an improved way to handle arrays.

Example Zsh Script:

Bash

```
#!/bin/zsh

files=(*.log)

if (( ${#files} > 0 )); then
   echo "Found log files: ${files[@]}"
else
   echo "No log files found."
```

Fish (Friendly Interactive Shell)

Fish is a unique shell designed from the ground up to be user-friendly and interactive. Unlike Bash and Zsh, it is **not POSIX-compliant**, which means its syntax is different and scripts written for it won't run in other shells. This non-compliance allows for a simpler, more intuitive syntax.

- **Key Features:** Fish provides out-of-the-box features that require plugins in other shells. These include **syntax highlighting**, intelligent **auto-suggestions** as you type based on command history and man pages, and a simple configuration process.
- **Syntax:** Fish's syntax is much more like a high-level programming language. It uses end to close blocks (if, for, function), and variables are scoped by default (set for local, set -g for global). It avoids the complex quoting and special characters common in other shells.

Example Fish Script:

Code snippet

```
#!/usr/bin/env fish

echo "Hello, what is your favorite color?"

read color
```

```
if test "$color" = "blue"
   echo "Blue is a great color!"
else
   echo "That's a nice color too."
end
```

5.2.2 Comparison of Shell Features in Detail

Building on the above shell comparisons, the unique set of features of each opens them up for specific use cases and a dedicated user base. The Bourne shell is the most basic (the original) and most portable. Its syntax, although bleak compared to shells in widespread use today, is extremely consistent, meaning it's great for writing scripts that need to work on a huge number of systems. Its main purpose is to execute commands and handle simple scripting tasks. But it does not implement any of the interactive features like command history, job control, aliases or things that modern interactive usage relies upon. The C shell (csh), aimed at a more casual user base, brought many virtual machine-like features that fundamentally changed how users interacted with commands and their arguments. It was much more convenient for interactive use due to its command history, aliases, and job control. Yet, the scripting functionalities were often mocked for their inconsistencies and non-standard syntax. Things like its handling of control structures and variables were seen as clunky and error-prone. To overcome the limitations of both Bourne and C shell, the Korn shell (ksh) was introduced which provided a powerful versatile environment for interactive use as well as scripting. It combined features from shells, including command-line editing, improved job control, and better scripting features. Its scripting syntax, for instance, was bolder and more consistent than that of the C shell, which made it a favorite of systems administrators and developers alike. The Bourne-Again shell (bash) is one of the most popular it is compatible with Bourne shell scripts, comes with many powerful features, and is very commonly available. Bash is as customizable as it gets and has tons of features under the hood for interactive use as well as for scripting. Its powerful command-line completion, history expansion, and rich scripting capabilities make it popular with both casual users and advanced developers. Bash is an acronym for the Bourne Again Shell, signaling

that its scripting syntax is from the Bourne shell, but with many improvements and extensions that deliver much more power and flexibility. Fast forward to zsh, which adds even more advanced features on top of bash. Power users and developers love it for its enhanced command-line completion, spelling correction, and its support for plugins. Zsh is customizable and extensible, enabling users to customize their shell environment according to their needs. Oh My Zsh, its plugin system, offers a large library of plugins and themes, so you can easily extend the shell's functionality and appearance. All shells have their own unique strengths and weaknesses, so users should evaluate based on individual requirements. The Bourne shell might be enough for some simple scripting tasks. The C shell or Korn shell might be good for writing the shell scripts interactively. For a robust and flexible shell that works great interactively and can be scriptable faster than you can say "reverse-timestamp-auto complete", bash or zsh is the way to go. If you prefer a command-line shell that is compatible with most Unix systems, Bash would be a good option, whereas if you want extensive customizability, zsh would be preferable.

Summary

Shell programming is a powerful method for automating tasks in Unix-like operating systems by writing scripts composed of shell commands. It serves as a user interface between the user and the operating system, allowing command execution, file manipulation, program execution, and text output in a programmable format. Shell scripts can include loops, conditionals, and variables, making them useful for automating repetitive tasks such as backups, software installation, and log analysis. Shell programming supports logic control structures similar to those found in high-level programming languages, allowing complex workflows to be expressed concisely in a script.

Different types of shells are available, each offering unique features and syntax. Common shells include the Bourne Shell (sh), Bourne Again Shell (bash), C Shell (csh), Korn Shell (ksh), and Z Shell (zsh). Bash is the most widely used, particularly in Linux environments, and provides extensive scripting capabilities, compatibility with the original Bourne shell, and advanced features such as command history, job control, and tab completion. C Shell, on the other hand, uses a syntax resembling the C programming language, making it preferable for users with a background in C. Korn Shell combines features of both the Bourne and

C shells, offering advanced scripting functionality and performance improvements.

Writing shell programs involves understanding shell-specific syntax and conventions. A script typically begins with a "shebang" (#!) line that indicates the interpreter to be used, followed by a series of commands or logic structures. Shell programming is particularly valuable in system administration, as it allows administrators to write scripts to monitor systems, manage user accounts, and perform routine maintenance tasks. It simplifies the execution of batch commands and helps users to create customized workflows tailored to their system needs.

Multiple-Choice Questions (MCQs)

1. Which of the following is NOT a type of shell in Unix/Linux?
    a) Bourne Shell (sh)
    b) Korn Shell (ksh)
    c) Python Shell (pysh)
    d) C Shell (csh)

    (Answer: c)
2. Which shell is the default for most Linux distributions?
    a) C Shell (csh)
    b) Korn Shell (ksh)
    c) Bash (Bourne Again Shell)
    d) Z Shell (zsh)

    (Answer: c)
3. In a shell script, which symbol is used for comments?
    a) //
    b) #
    c) /* */
    d) $

    (Answer: b)
4. Which command is used to make a shell script executable?
    a) chmod +x script.sh
    b) execute script.sh
    c) run script.sh
    d) compile script.sh

    (Answer: a)
5. What is the correct syntax for an if statement in a shell script?
    a) if (condition) then ... fi

b) if [condition]; then ... fi
c) if condition { ... }
d) if: condition -> ... fi

(Answer: b)

6. Which command is used to display the currently running processes in Linux?
   a) ps
   b) ls
   c) pwd
   d) kill

   (Answer: a)
7. What is the purpose of the read command in shell scripting?
   a) To print text on the screen
   b) To read input from the user
   c) To delete a file
   d) To execute another script

   (Answer: b)
8. Which loop structure is used in shell scripting to repeat commands?
   a) while
   b) do-while
   c) until
   d) Both a and c

   (Answer: d)
9. Which symbol is used for passing parameters to a shell script?
   a) &
   b) %
   c) $
   d) #

   (Answer: c)
10. What is the function of the grep command in shell scripting?
    a) To search for a pattern in a file
    b) To copy a file
    c) To move files
    d) To delete files

    (Answer: a)

Short Questions

1. What is shell programming, and why is it used?

2. List and explain three types of shells in Unix/Linux.
3. What is the difference between interactive and non-interactive shells?
4. How does command execution work in a shell?
5. What is the purpose of the shebang (#!) line in shell scripts?
6. How does decision-making work in shell programming? Provide an example.
7. What is a function in shell scripting, and why is it useful?
8. How can parameters be passed to a shell script? Provide an example.
9. Explain the use of filters like grep, awk, and sed in shell programming.
10. What is the difference between $1, $2, and $@ in shell scripting?

Long Questions

1. Explain the concept of shell programming, its importance, and common applications.
2. Compare various types of shells (sh, bash, csh, ksh, zsh) and their differences.
3. Discuss the command execution process in Linux, from user input to execution.
4. Write a shell script to check if a given number is even or odd. Explain the script.
5. What is decision-making in shell scripting? Provide examples of if, case, and for loops.
6. Explain functions in shell scripting, how they work, and their advantages.
7. How does parameter pass and argument handling work in shell scripting? Provide examples.
8. Describe how filtering commands like grep, sed, and awk are used in shell programming.
9. Explain error handling and debugging techniques in shell scripting.
10. Write a shell script that accepts a filename as an argument and checks whether it exists and is readable. Explain the script.

Glossary

Operating System (OS): A system software that manages hardware and software resources, providing services for computer programs.

Process: A program in execution, which includes the current activity, program counter, registers, and variables.

Kernel: The core component of the operating system that controls all system operations and hardware communication.

System Call: A request made by a program to the OS for performing tasks like file manipulation or process control.

Multitasking: An OS feature that allows multiple processes to run concurrently by time-sharing CPU resources.

Process State: The current status of a process, typically categorized as new, ready, running, waiting, or terminated.

Process Control Block (PCB): A data structure maintained by the OS for each process, containing process ID, state, program counter, CPU registers, and memory management info.

Contiguous Memory Allocation: Memory allocation method where each process gets a single, continuous memory block. Prone to fragmentation.

Paging: A memory management technique that breaks physical and logical memory into fixed-size blocks to enable non-contiguous allocation.

Demand Paging: A strategy where pages are loaded into memory only when they are required during execution, reducing memory usage.

Page Table: A data structure used to map logical pages to physical frames in paging systems.

Page Fault: An interrupt triggered when a process tries to access a page not currently in memory.

Thrashing: A condition where excessive paging operations hinder system performance due to insufficient memory.

Disk Scheduling: The method used by OS to determine the order in which disk I/O requests are processed to optimize performance.

FCFS (First-Come, First-Served): A disk scheduling algorithm that serves I/O requests in the order they arrive.

SSTF (Shortest Seek Time First): Disk scheduling algorithm that serves the request closest to the current disk head position.

SCAN and LOOK: Disk scheduling algorithms where the disk arm moves in one direction to service requests, then reverses.

Distributed System: A model where processing is distributed across multiple networked computers, working as a single system.

I/O Hardware: Physical devices used for input/output operations, such as keyboards, printers, disk drives, and controllers.

Disk Structure: The layout of data on a hard disk, including platters, tracks, sectors, and cylinders.

Stateless Service: A service that does not retain client information between sessions. Each request is treated independently.

Stateful Service: A service that maintains state information across multiple requests from the same client.

Shell: A command-line interface between the user and the OS for executing commands and running programs.

Shell Script: A file containing a sequence of shell commands for automated execution.

Bash (Bourne Again Shell): A widely used Unix shell that supports scripting, command history, job control, and more.

sh, csh, ksh, zsh: Different Unix shell types with varying syntax and capabilities for scripting and user interaction.

Echo: A shell command that prints text to the terminal.

Variables (in Shell): Used to store data values in a script, defined using var=value syntax.

Conditional Statements (Shell): Used to make decisions in scripts, using if, else, elif, and case.

Looping (Shell): Shell constructs like for, while, and until used to repeat code blocks.

Command Substitution: Allows the output of a command to replace the command itself using backticks (`) or $(...).

Redirection: Used to direct input/output from/to files instead of the default terminal using >, <, >>.

Pipelines (|): A feature that passes the output of one command as input to another.

References

Chapter 1: Introduction to Operating System

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2021). Operating System Concepts (10th ed.). Wiley.
2. Tanenbaum, A. S., & Bos, H. (2022). Modern Operating Systems (5th ed.). Pearson.
3. Stallings, W. (2023). Operating Systems: Internals and Design Principles (10th ed.). Pearson.
4. Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books. (Available online at: https://pages.cs.wisc.edu/~remzi/OSTEP/)
5. Anderson, T., & Dahlin, M. (2014). Operating Systems: Principles and Practice (2nd ed.). Recursive Books.

Chapter 2: Process Management and Synchronization

1. Tanenbaum, A. S., & Bos, H. (2022). Modern Operating Systems (5th ed.). Pearson. (Chapters on Process Management)
2. Silberschatz, A., Galvin, P. B., & Gagne, G. (2021). Operating System Concepts (10th ed.). Wiley. (Chapters on Process Synchronization)
3. Dijkstra, E. W. (1968). Cooperating Sequential Processes. In F. Genuys (Ed.), Programming Languages (pp. 43-112). Academic Press.
4. Deitel, H. M., Deitel, P. J., & Choffnes, D. R. (2015). Operating Systems (3rd ed.). Pearson.
5. Liu, J. W. S. (2000). Real-Time Systems. Prentice Hall. (Sections on Process Scheduling)

Chapter 3: Storage Management

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2021). Operating System Concepts (10th ed.). Wiley. (Chapters on Memory Management)
2. Denning, P. J. (1970). Virtual Memory. ACM Computing Surveys, 2(3), 153-189.

3. McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. M. (2014). The Design and Implementation of the FreeBSD Operating System (2nd ed.). Addison-Wesley Professional.
4. Love, R. (2010). Linux Kernel Development (3rd ed.). Addison-Wesley Professional. (Chapters on Memory Management)
5. Gorman, M. (2004). Understanding the Linux Virtual Memory Manager. Prentice Hall.

Chapter 4: Disk Scheduling and Distributed Systems

1. Tanenbaum, A. S., & Van Steen, M. (2016). Distributed Systems: Principles and Paradigms (3rd ed.). Pearson.
2. Silberschatz, A., Galvin, P. B., & Gagne, G. (2021). Operating System Concepts (10th ed.). Wiley. (Chapters on I/O Systems)
3. Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., & Patterson, D. A. (1994). RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys, 26(2), 145-185.
4. Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2022). Distributed Systems: Concepts and Design (6th ed.). Pearson.
5. Hennessy, J. L., & Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann. (Chapters on Storage Systems)

Chapter 5: Stateful Versus Stateless Service and Shell Programming

1. Robbins, A., & Beebe, N. H. F. (2005). Classic Shell Scripting. O'Reilly Media.
2. Blum, R., & Bresnahan, C. (2021). Linux Command Line and Shell Scripting Bible (4th ed.). Wiley.
3. Cooper, M. (2021). Advanced Bash Scripting Guide. Linux Documentation Project. (Available online at: https://tldp.org/LDP/abs/html/)
4. Powers, S., Peek, J., O'Reilly, T., & Loukides, M. (2002). Unix Power Tools (3rd ed.). O'Reilly Media.
5. Sobell, M. G., & Helmke, C. (2018). A Practical Guide to Linux Commands, Editors, and Shell Programming (4th ed.). Addison-Wesley Professional

MATS UNIVERSITY

MATS CENTRE FOR DISTANCE AND ONLINE EDUCATION

UNIVERSITY CAMPUS: Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 **Toll Free ODL MODE :** 81520 79999, 81520 29999

Website: www.matsodl.com