Implementation of modern operating systems

Chapter One
Introduction to Design and Implementation of OS

Introduction to Design and Implementation of Operating Systems

- An Operating System (OS) is a critical software layer that manages hardware resources, provides an interface between users and hardware, and enables the execution of applications.
- The design and implementation of an OS are complex tasks that involve balancing performance, security, scalability, and resource management.

Key Components of OS Design

- **Kernel:**
  - The kernel is the core of an OS, responsible for managing hardware resources (CPU, memory, I/O devices). It interacts with the hardware through device drivers.
  - Types of Kernels:
    - ✓ Monolithic Kernel: All operating system services run in kernel mode, offering high performance but with risks of system crashes.
    - ✓ Microkernel: Only essential functions run in the kernel, with other services running in user mode, enhancing system stability and security.
    - ✓ Hybrid Kernel: A combination of monolithic and microkernel architectures, balancing performance and security.

CONT...

- **Process Management**
  - ❖ Processes represent running programs.
  - ❖ The OS is responsible for creating, scheduling, and terminating processes.
  - ❖ Key Concepts:
    - ✓ Scheduling: Determines which process runs at any given time (e.g., round-robin, priority-based scheduling).
    - ✓ Multithreading: Allows a process to be divided into multiple threads that can run concurrently, improving responsiveness and resource utilization.
    - ✓ Concurrency: The OS must handle multiple processes and threads running in parallel without conflicts (e.g., race conditions).

CONT...

- **Memory Management**
  - ❖ Memory management involves allocating and de allocating memory to processes while ensuring efficient usage of system RAM.
  - ❖ Key Techniques:
    - ✓ Virtual Memory: Extends physical memory by using disk space, allowing the system to run larger applications than would fit into physical RAM alone.
    - ✓ **Paging and Segmentation:** Techniques to manage how memory is allocated and protected.

CONT…

- **File Systems:**
  - ❖ The file system manages how data is stored and retrieved from storage devices.
  - ❖ Key Functions:
    - ✓ **File Storage:** Storing data in structured formats (files and directories).
    - ✓ **File Access Control:** Ensuring security by controlling who can access, read, or modify files.
    - ✓ Consistency and Recovery: Handling system crashes and ensuring the file system remains consistent.
-

CONT...

- **I/O Management**
  - Input/output management :- involves handling communication between system hardware (disks, printers, keyboards) and processes.
  - The OS uses device drivers to interact with various hardware components.
  - Buffering and Caching are techniques used to improve the performance of I/O operations by temporarily storing data.

CONT…

- **Security and Protection:**
  - Modern operating systems implement mechanisms for ensuring the security of processes, memory, and files.
  - User Authentication: Ensures only authorized users can access system resources.
  - **Access Control:** Prevents unauthorized access to system resources and ensures isolation between processes.

CONT…

- **Networking:**
  - Networking capabilities allow an OS to support communication between systems via protocols like TCP/IP.
  - Sockets: Enable inter-process communication over a network.
  - Network Security: Includes encryption, firewalls, and secure protocols to protect data transmission.

CONT…

- **Device Management:**
  - The OS controls and coordinates the use of hardware devices such as printers, keyboards, and displays.
  - Device Drivers: These are special programs in the OS that facilitate communication between the system and hardware devices.

OS Implementation Approaches:

- - Low-level Language (e.g., C): OS kernels are often implemented in low-level languages like C for direct control over hardware.
- **- Modular Design:** Modern operating systems follow a modular approach, where different components (like the file system, memory manager) are separate modules, improving maintainability and extensibility.
- **- Layered Architecture:** Some operating systems use a layered approach, where the OS functions are divided into layers. Each layer provides services to the layer above it.

Challenges in OS Design:

- Performance vs. Security: Balancing system performance while maintaining high levels of security is a key challenge in OS design.
- Scalability: The OS must efficiently handle the growing number of processes, users, and resources as the system scales.
- Real-Time Requirements: Some systems, like real-time operating systems (RTOS), need to meet strict timing constraints, which require specialized design approaches.

Design Goals: User Goals and System Goals

- When designing an operating system (OS) or any software system, it's essential to balance user goals and system goals to create an efficient, user-friendly, and reliable product.
- Both sets of goals shape the overall architecture and behaviour of the system.

CONT…

- **User Goals:**
  - User goals refer to the expectations and needs of the end-users who interact with the operating system or software.
  - These focus on the usability, functionality, and overall experience from the user's perspective.
  - Ease of Use:
    - The system should be intuitive and simple to operate. Users should be able to navigate the interface and access system functions without extensive training or technical

CONT...

- ❖Example: A well-designed graphical user interface (GUI) with clear icons, buttons, and navigation helps non-technical users perform tasks easily

- **Performance:**
  - ❖ Users expect the system to be fast and responsive. This includes quick boot times, fast application launch, and efficient file handling.
  - ❖ Example: Optimizing the OS to reduce lag and maximize resource allocation for tasks enhances the user experience.

CONT...

- **Reliability:**

❖ The system must be stable and not crash frequently. Users should trust that the OS will continue to function correctly under normal use without data loss or corruption.

❖ Example: Implementing fault tolerance and recovery mechanisms to handle unexpected errors.

- **Security and Privacy:**

❖ Users want to ensure that their data is secure from unauthorized access and that their privacy is respected.

❖ This includes the protection of sensitive information and secure access control.

❖ Example: Implementing encryption, multi-factor authentication, and access control lists.

CONT...

- **Flexibility:**
  - The OS should support a wide range of user activities, including gaming, programming, or multimedia production, and allow for the easy addition of software and devices.
  - Example: Supporting various applications and devices (printers, scanners, external drives) through device drivers and plug-and-play features.
- **Customization:**
  - Users should be able to personalize the system to their preferences, such as changing themes, window behavior, or setting up custom shortcuts.
  - Example: Allowing users to adjust the desktop environment, modify system preferences, and install custom software.

CONT...

System Goals

- System goals refer to the internal objectives that focus on how the operating system manages hardware resources and provides a stable environment for user applications.
- These goals ensure that the system is efficient, maintainable, and scalable.
- **Efficiency:**
  - The system should use resources (CPU, memory, I/O devices) as efficiently as possible to provide optimal performance while minimizing overhead.
  - Example: Efficient memory management techniques such as paging and caching help the system run smoothly without wasting resources.

CONT…

- **Scalability:**
  - The system should be scalable, meaning it can handle increasing workloads (more users, processes, and devices) without a degradation in performance.
  - Example: Supporting multi-core processors or distributed systems to handle concurrent tasks more effectively.
- Resource Management:
  - Proper management of hardware resources is crucial.
  - This includes scheduling CPU time, allocating memory, and handling disk I/O to ensure that multiple processes can run simultaneously without interference.
  - Example: Implementing priority-based scheduling to manage resource allocation based on the importance of tasks.

CONT...

- **Security:**
  - ❖ At the system level, security focuses on protecting the OS from unauthorized access, malware, and exploitation. This includes protecting system integrity and maintaining control over who can access certain resources.
  - ❖ Example: Incorporating user privilege levels, kernel mode, and user mode separation to limit what processes can do.
- Reliability and Fault Tolerance:
  - ❖ The system must be reliable, meaning it continues operating under normal conditions and can recover from failures without crashing.
  - ❖ Example: Implementing redundancy or backup mechanisms to recover from hardware or software failures.

CONT...

- **Portability:**
  - The OS should be designed to be portable, meaning it can be adapted to run on various hardware platforms without significant modification.
  - Example: Unix-based systems are known for their portability across different hardware architectures due to the use of high-level languages like C.
- **Maintainability:**
  - The system should be easy to maintain, allowing updates and patches to be applied efficiently without affecting the rest of the system.
  - Example: Modular kernel designs allow different components of the OS to be updated independently, making maintenance easier.

TDD (Test-Driven Design), BDD(Behavior-Driven Design

- Test-Driven Development (TDD) and Behavior-Driven Development (BDD) are both methodologies in software development that emphasize testing and validation of code but differ in their approaches and focus.
- Let’s explore these methodologies, along with related concepts like Acceptance Test-Driven Development (ATDD), Unit Testing, and other testing practices that integrate into the development process.

Test-Driven Development (TDD):

- **TDD** is a software development technique where tests are written before writing the actual code.
- It follows a strict process where code is written only to pass pre-defined tests.
- The goal is to ensure that the software meets the requirements at every stage and that any modifications or extensions don’t break existing functionality.
- **Process:**
  - ❖ Red-Green-Refactor:
    - ✓ 1. Red: Write a failing test case.
    - ✓ 2. Green: Write the minimum amount of code to pass the test.
    - ✓ 3. Refactor: Improve the code without changing its functionality.

CONT...

- **Focus:**
  - TDD emphasizes unit tests, which test small parts of the application (typically functions or methods). The idea is to focus on building the system in small, incremental steps, each verified by tests.
- **Advantages:**
  - Encourages writing clean, modular, and testable code.
  - Helps prevent regression bugs because tests are always run when changes are made.
  - Provides a safety net for refactoring and making changes.
- **Disadvantages:**
  - Can be more time-consuming upfront.
  - Developers may focus too much on passing tests rather than on overall system design.

CONT…

- **Example:**
  - ❖ Writing a test to check if a function correctly adds two numbers. After the test fails (since the function hasn’t been written yet), you write the code to implement the function.
- **Popular Tools:**
  - ❖ JUnit (Java), NUnit (.NET), pytest (Python), RSpec (Ruby).

Behavior-Driven Development (BDD):

- BDD builds on TDD by combining the general techniques of test-driven development with domain-specific language that allows stakeholders (including non-technical team members) to define behavior in a way that's understandable by all.
- **Process:**
  - ❖Given-When-Then:
    - ✓ Given: A context or precondition.
    - ✓When: An action or event that takes place.
    - ✓ Then: An expected outcome or result.

CONT...

- **Focus:**
  - BDD focuses more on behavior than implementation. It bridges the gap between developers and non-technical stakeholders by using a common language to describe the system's expected behavior. Tests are written in a more natural, human-readable form, often using the Given-When-Then format.
- **Advantages:**
  - Better communication between developers, testers, and business stakeholders.
  - Helps ensure that the correct functionality is being built.
  - Keeps the focus on the system's behavior, not just isolated units.
- **Disadvantages:**
  - Writing feature files in natural language can sometimes feel redundant for simple tasks.
  - Can become cumbersome for large, complex systems if not managed well.

CONT...

- **Example:**
  - In BDD, you write a test scenario such as:
  - ```

    Scenario: Successful login
    Given the user is on the login page
    When the user enters valid credentials
    Then the user is redirected to the homepage
    ```
- **Popular Tools**:
  - Cucumber (supports multiple languages), SpecFlow (.NET), JBehave (Java).
    ```

Acceptance Test-Driven Development (ATDD):

- **ATDD** is a collaborative practice where acceptance tests are defined before development begins.
- These tests define what the system should do from the user's perspective, ensuring that the system meets the business requirements.
- **Process:**
  - Similar to TDD but involves customer collaboration to write the acceptance tests, ensuring that both developers and stakeholders understand and agree on the required functionality.
- **Focus:**
  - ATDD is more about validating the functionality from an external, user-centered view rather than testing the internal workings of the system.

CONT...

- Advantages:
  - **Ensures alignment between development and user expectations.**
  - **Creates a clear contract of what "done" means for a feature.**
- Example:
  - **Acceptance test: "When a user tries to withdraw more money than they have in their account, the transaction should be declined with an error message."**
- Popular Tools:
  - **FitNesse, Robot Framework, Cucumber (also used in BDD).**

OS Structure: Monolithic, Microkernel, Exokernel, Multi-kernel

- The structure of an operating system (OS) refers to how its core components, such as the kernel, processes, and memory, are organized and interact with hardware.
- Different OS structures have evolved to address specific requirements such as performance, modularity, security, and ease of maintenance.
- Here’s an overview of four major OS structures: Monolithic Kernel, Microkernel, Exokernel, and Multikernel.

Monolithic Kernel:

- **A monolithic kernel is a single large process running entirely in a single address space, where all the operating system's core services (process management, memory management, file systems, device drivers) run in kernel mode.**
- **This structure emphasizes performance since all core services run in the same address space, avoiding context-switching overhead.**

CONT...

- Key Characteristics:
  - **All OS services are tightly integrated into the kernel.**
  - **High performance due to lack of boundary checks between components.**
- **Drawback:**
  - **A failure in one part of the system (e.g., a device driver) can crash the entire system.**

Examples:

- **UNIX and Linux are prime examples of operating systems based on a monolithic kernel structure, although Linux has introduced some modular features.**

CONT...

- Advantages:
  - **High performance due to direct communication between components.**
  - **Efficient system call execution.**
- Disadvantages:
  - **Less modularity, making debugging and maintaining the system harder.**
  - **More prone to system crashes if a bug or fault occurs.**

Microkernel:

- **A microkernel is a minimalistic approach to kernel design, where only the most essential services (like IPC, basic memory management, and scheduling) run in kernel mode.**
- **All other services, such as device drivers, file systems, and network protocols, run in user mode as separate processes.**

CONT...

- Key Characteristics:
  - **Minimizes the kernel size by running only essential services in kernel mode.**
  - **Other services run as user-space processes and communicate with the kernel using inter-process communication (IPC).**
- Drawback:
  - **Can have performance overhead due to the need for more frequent context switches and message passing between user space and the kernel.**

CONT…

- Examples:
  - **Minix and QNX are examples of microkernel-based operating systems.**
  - **macOS and Windows NT are often considered hybrids that incorporate microkernel concepts.**
- Advantages:
  - **Better fault isolation—if a service crashes, it doesn’t take down the entire system.**
  - **Greater modularity makes it easier to update and maintain the system.**
- Disadvantages:
  - **Typically slower than monolithic kernels due to the overhead of message passing.**
  - **More complex interprocess communication mechanisms.**

Exokernel:

- **The exokernel architecture takes a radical approach by providing minimal abstractions over hardware.**
- **Instead of offering high-level abstractions, the kernel allows applications to have direct access to hardware, leaving resource management policies up to the applications**

CONT...

- Key Characteristics:
  - **The exokernel is extremely minimalistic, focusing solely on resource allocation, protection, and multiplexing.**
  - **Applications can define their own abstractions and resource management strategies, leading to potentially more efficient use of hardware resources.**
- Drawback:
  - **It places a significant burden on application developers, who now have to manage hardware resources themselves.**

CONT...

- **Examples:**
  - **Xok (an experimental OS) is a classic example of an exokernel.**
  - **Though not widely adopted in commercial operating systems, the exokernel concept influences systems research and specialized environments.**
- Advantages:
  - **High performance: Applications can directly control hardware resources, leading to potential performance improvements.**
  - **Customizability: Applications can optimize the use of system resources to meet specific needs.**

CONT…

- Disadvantages:
  - **Complexity for application developers, who need to handle low-level details of hardware management.**
  - **Lack of uniform abstractions can make application development and portability more difficult.**

Multikernel:

- **The multikernel architecture is designed for systems with multi-core processors and distributed architectures.**
- **In a multikernel OS, each core runs its own independent kernel, and communication between kernels happens through message passing.**
- **This approach treats each core or processor as a separate entity.**

CONT...

- Key Characteristics:
  - **Each core has its own kernel instance, and they communicate via explicit message passing.**
  - **The goal is to improve scalability on multi-core processors and distributed systems by avoiding shared data structures.**
- Drawback:
  - **Increased complexity in inter-kernel communication.**
- Examples:
  - **Barrelfish, a research OS developed by Microsoft and ETH Zurich, is one of the pioneering examples of the multikernel architecture.**

CONT…

- Advantages:
  - **Scalability: Optimized for multi-core and distributed systems.**
  - **Fault isolation: Crashes on one kernel (core) don’t affect others.**
- Disadvantages:
  - **More complex communication and coordination mechanisms.**
  - **Possible performance overhead due to the communication between kernels.**

OS Structure:

- Modern OS Architecture has evolved **significantly to address the growing needs for scalability, security, multi-core processing, and distributed computing.**
- **While early operating systems followed simpler structures such as monolithic kernels or microkernels, modern systems often blend several architectures and employ modular and hybrid designs.**
- **Here’s an overview of modern OS architecture types and how they are applied today:**

Hybrid Kernel Architecture:

- **A hybrid kernel combines aspects of both monolithic kernels and microkernels, attempting to balance performance and modularity. In a hybrid kernel, core operating system functions such as process management, memory management, and hardware control run in kernel space, but some components such as device drivers and file systems may run in user space.**
- Design: **While the core kernel remains monolithic for performance reasons, some services can run in user mode to enhance fault tolerance and security.**

CONT...

- Examples:
  - **Windows NT family (including Windows 10 and Windows 11) employs a hybrid kernel.**
  - **macOS: Combines elements of Mach (a microkernel) and BSD Unix (monolithic) in a hybrid approach.**
- Advantages:
  - **Offers the performance benefits of a monolithic kernel while improving fault isolation by running some components outside the kernel.**
- **Disadvantages:**
  - **Still maintains some complexity, and large components in kernel mode can still cause system-wide crashes.**

Modular Kernel Architecture

- **A modular kernel is a variation of the monolithic kernel, where parts of the kernel can be loaded or unloaded as modules (e.g., device drivers or file systems).**
- **This allows dynamic adaptation to different hardware configurations and functionality.**
- Design: **Modules can be loaded or unloaded on demand without rebooting the system, making it highly extensible and flexible.**

CONT...

- Examples:
  - **Linux is the best-known example of a modular kernel. Linux allows kernel modules to be loaded at runtime, such as when new hardware is added.**
- Advantages:
  - **Flexibility in updating or extending the kernel without downtime, efficient memory use by only loading modules when needed.**
- Disadvantages:
  - **Dynamic loading of modules can introduce complexity, and misbehaving modules can still crash the system.**

Microkernel Architecture

- **A microkernel is designed to run the minimal set of essential services in kernel space (like scheduling and IPC), while everything else, including device drivers, file systems, and networking, runs in user space.**
- Design: **Since most services run outside the kernel, faults in user space do not directly affect the core kernel, improving security and fault tolerance.**

CONT...

- Examples:
  - **Minix and QNX are examples of microkernel-based systems.**
  - **MacOS uses elements of the Mach microkernel.**
- **Advantages:**
  - **Increased security and modularity, easier maintenance, and enhanced system stability because user-space services can crash without taking down the entire system.**
- Disadvantages:
  - **Performance overhead due to frequent context switching between user space and kernel space, leading to lower performance compared to monolithic kernels.**

Multikernel Architecture:

- **A multikernel architecture is designed for systems with multi-core processors, where each core runs an independent kernel.**
- **The cores communicate via message passing instead of shared memory, allowing for better scalability and fault isolation.**
- Design**: Each core operates as a separate entity with its own kernel instance, and explicit message-passing mechanisms are used for communication.**

CONT...

- Examples:
  - **Barrelfish OS, developed by Microsoft Research and ETH Zurich, is a leading example of multikernel architecture.**
- Advantages:
  - **Highly scalable, ideal for multi-core and many-core systems. Improves fault isolation since each kernel runs independently on its core.**
- **Disadvantages:**
  - **Increased complexity in managing inter-core communication.**

Exokernel Architecture

- **The exokernel architecture is a radical departure from traditional kernels. Instead of providing abstract services like process scheduling or memory management, the exokernel gives applications direct access to hardware resources, leaving resource management to user-level libraries.**
- Design: **The kernel focuses on minimal hardware protection and resource multiplexing. Applications create their own abstractions using libraries, allowing for more flexibility and performance optimization**

CONT...

- Examples:
  - **Xok and Nemesis are research-based exokernel operating systems.**
- **Advantages:**
  - **Provides high performance by minimizing abstraction layers, allowing applications to directly control hardware resources. Efficient for specialized systems where applications need fine-grained control over hardware.**
- Disadvantages:
  - **Complexity for developers: Applications need to handle low-level hardware management, making development more challenging. This design is not well-suited for general-purpose computing environments.**

Microservices and Cloud-Native Architectures:

- **While not directly an "OS architecture," modern cloud-native OS environments, especially in large-scale data centers and cloud infrastructures, employ a microservices approach.**
- **In this architecture, applications are broken into independent services that run in lightweight containers, each with its own operating system instance (or even a single function in serverless computing).**
- Design: **Instead of a monolithic application or kernel, each service runs in a separate container or virtual machine, often managed by orchestration tools like Kubernetes.**

CONT…

- Examples:
  - **Google’s Kubernetes manages containerized applications across clusters of machines.**
  - **AWS Lambda provides a serverless architecture where code runs in response to events, without explicit OS management by the developer.**
- Advantages:
  - **Highly scalable, resilient to failures, and easily deployable in distributed cloud environments.**
- Disadvantages:
  - **Complexity in managing microservices, service discovery, and handling inter-service communication.**

Real-Time Operating Systems (RTOS):

- **An RTOS is an operating system designed to serve real-time applications that require strict timing constraints and deterministic responses. It guarantees that high-priority tasks receive processing time when they need it.**
- Design: **Typically, RTOS is designed with preemptive multitasking and real-time scheduling to ensure predictable timing behavior.**

CONT...

- Examples:
  - **VxWorks, FreeRTOS, and QNX are common real-time operating systems used in embedded systems.**
- Advantages:
  - **Guarantees time-critical operations and provides predictable, low-latency performance.**
- Disadvantages:
  - **Limited flexibility and typically tailored to specific, time-sensitive applications.**