Operating System

Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

Outline

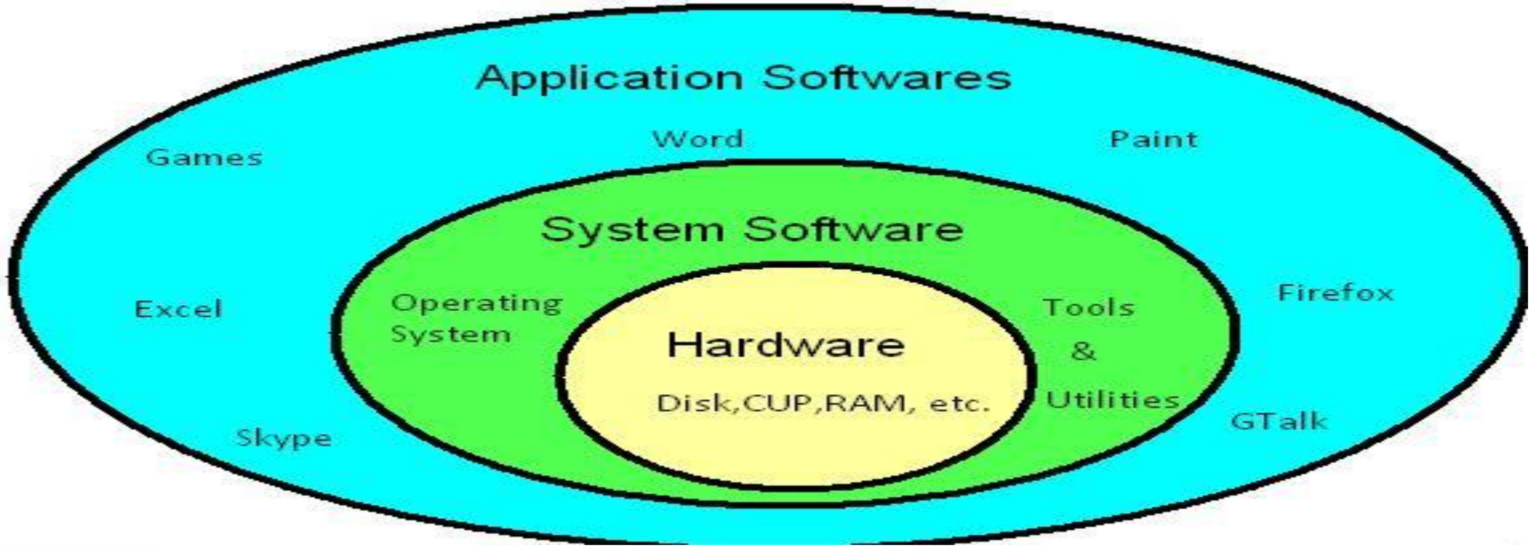
- System Software
- Introduction to Operating System
- Operating System Services
- Function of Operating System
- Evolution of Operating System

System Software

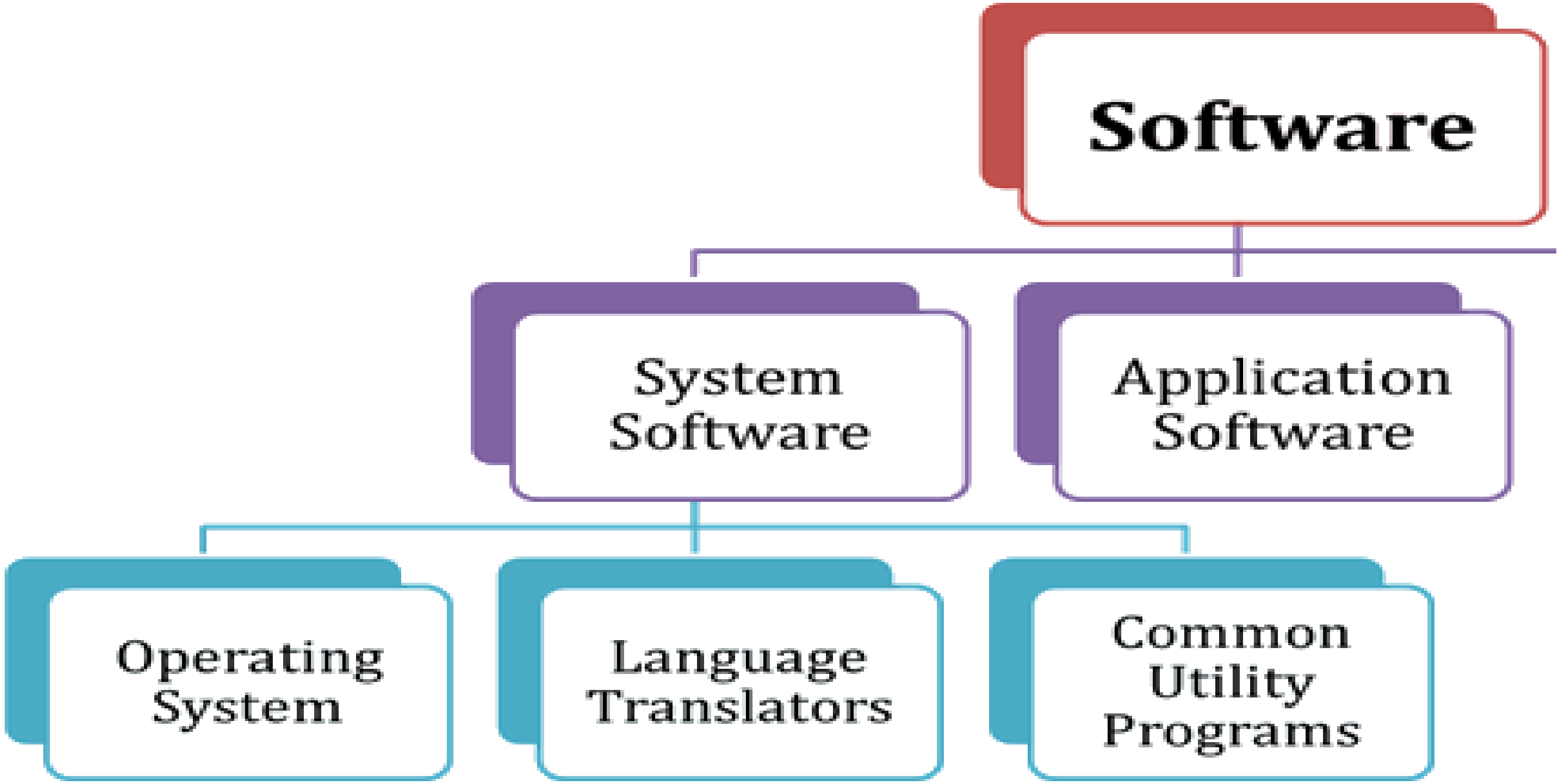
- It can be defined as act of building **Systems Software using System Programming Languages.**

System Software (continued)

- System software consists of the program that control or maintain the operations of the computer and its devices.
- System software serves as the interface between the user, the application software and the computer hardware.



Types of Software

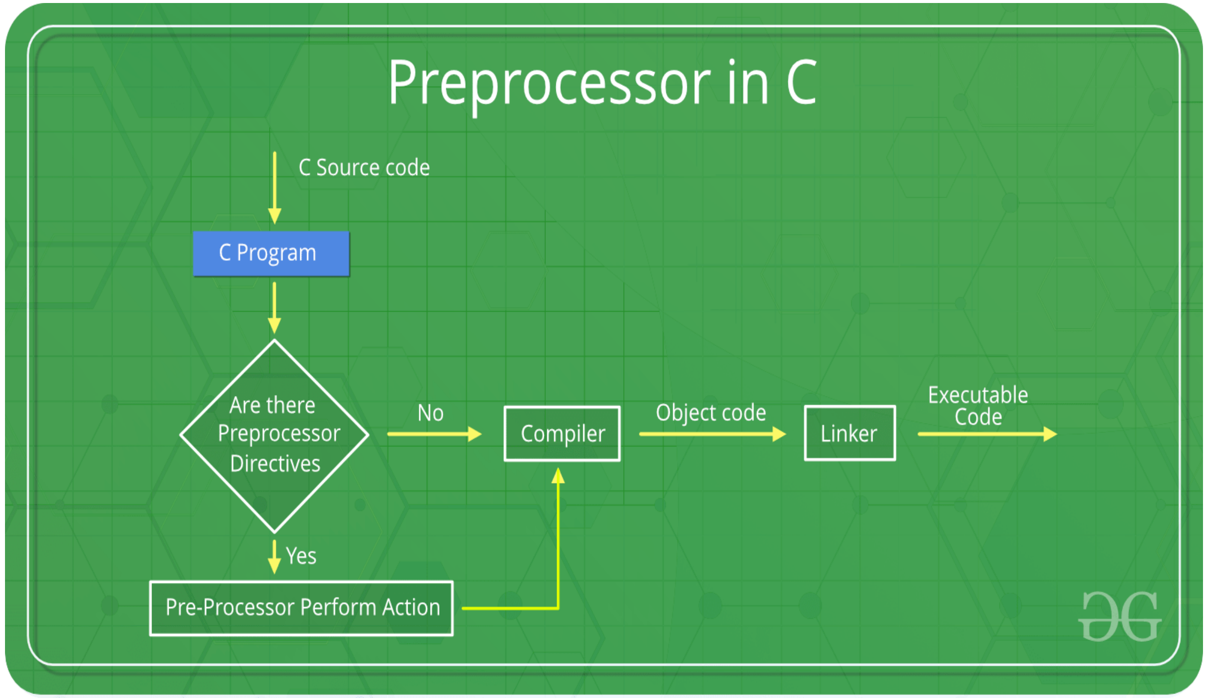


System Programs

- Macro processor
- Assemblers
- Compiler
- Interpreter
- Linkers
- Loaders
- Operating System
- Device drivers

Macro Processor

- Pre-processors are programs that process our source code before compilation.



Assembler

- An assembler translates assembly language into machine code.
- **Purpose of an Assembler:**
  - Simplifies programming by allowing programmers to write instructions using symbolic names rather than binary.
  - Translates these symbolic instructions into executable binary code.

Assembler (continued)

```python
# Python Code
num1 = 5
num2 = 3
result = num1 + num2
print("The sum is:", result)
```

```
; Assembly Code
MOV AX, 05h          ; Load the first number (5) into the AX register
MOV BX, 03h          ; Load the second number (3) into the BX register
ADD AX, BX           ; Add the contents of BX to AX (AX = AX + BX)
MOV RESULT, AX       ; Store the result (8) in a memory location named RESULT
```

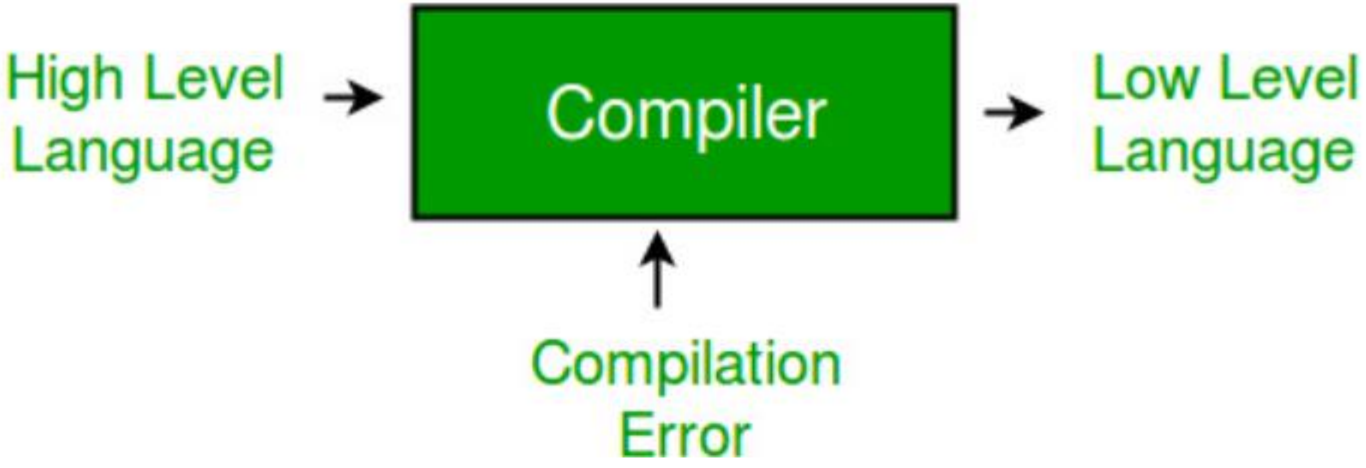
Assembler (continued)

- **Advantages of Assembly Language**
  - Provides direct control over hardware.
  - Used in performance-critical and hardware-specific tasks (e.g., embedded systems, device drivers).
  - Easier to debug than raw binary.

- **Applications of Assemblers**
  - Performance optimization in real-time systems.
  - Developing operating system kernels.

Compiler

- The language processor that reads the complete source program written in high level language **as a whole in one go and translates it** into an equivalent program in machine language is called as a Compiler.



Compiler (Continued)

The compilation process involves multiple stages:

- **Lexical Analysis:**
  - Breaks the source code into tokens (e.g., keywords, identifiers, symbols).
- **Syntax Analysis (Parsing):**
  - Checks the code structure against the grammar of the language.
- **Semantic Analysis:**
  - Checks the meaning of the code (e.g., type checking, variable declaration).
- **Intermediate Code Generation:**
  - Converts the code into an intermediate representation (IR) for optimization.
- **Optimization:**
  - Improves performance by optimizing code without changing its functionality.
- **Code Generation:**
  - Converts IR to machine code (binary instructions).
- **Code Linking:**
  - Combines compiled code with libraries or other program modules to create an executable file.

- **Lexical Analysis:**
  - Tokens: num1, =, 5, +, etc.
- **Syntax Analysis:**
  - Ensures correct syntax like variable assignment and the print statement.
- **Semantic Analysis:**
  - Checks compatibility (e.g., integers can be added).
- **Intermediate Representation:**

```
# Python Code
num1 = 5
num2 = 3
result = num1 + num2
print("The sum is:", result)
```

```
t1 = 5
t2 = 3
t3 = t1 + t2
print(t3)
```

- **Machine Code:**

```
MOV AX, 5        ; Load 5 into register AX
MOV BX, 3        ; Load 3 into register BX
ADD AX, BX       ; Add BX to AX
CALL PRINT       ; Call a function to print the result
```

Interpreter

- An interpreter, like a compiler, **translates high-level language into low-level machine language.**

- Why is an interpreter needed?

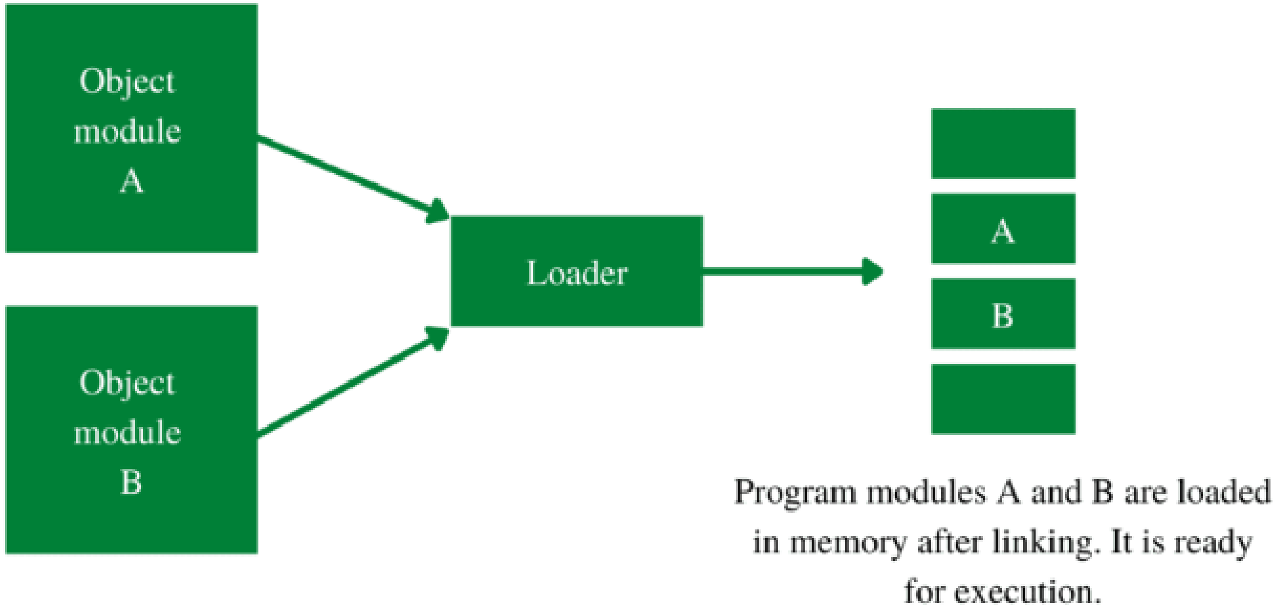
Interpreter (cont…)

- The difference lies in the way they read the source code or input.
  - A compiler reads the **whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes.**
  - An interpreter reads **a statement from the input, executes it, then takes the next statement in sequence**.
- If an error occurs,
  - **an interpreter stops execution and reports it. The interpreter moves on to the next line for execution only after removal of the error.**
  - whereas a compiler **reads the whole program even if it encounters several errors.**

COMPILER	INTERPRETER
A compiler is a program which coverts the entire source code of a programming language into executable machine code for a CPU.	interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met
Generates intermediate object code.	No intermediate object code is generated.
Examples: C, C++, Java	Examples: Python, Perl

Loaders

- A **loader** is a system software program responsible for **loading an executable program into memory** so that it can be run by the operating system.



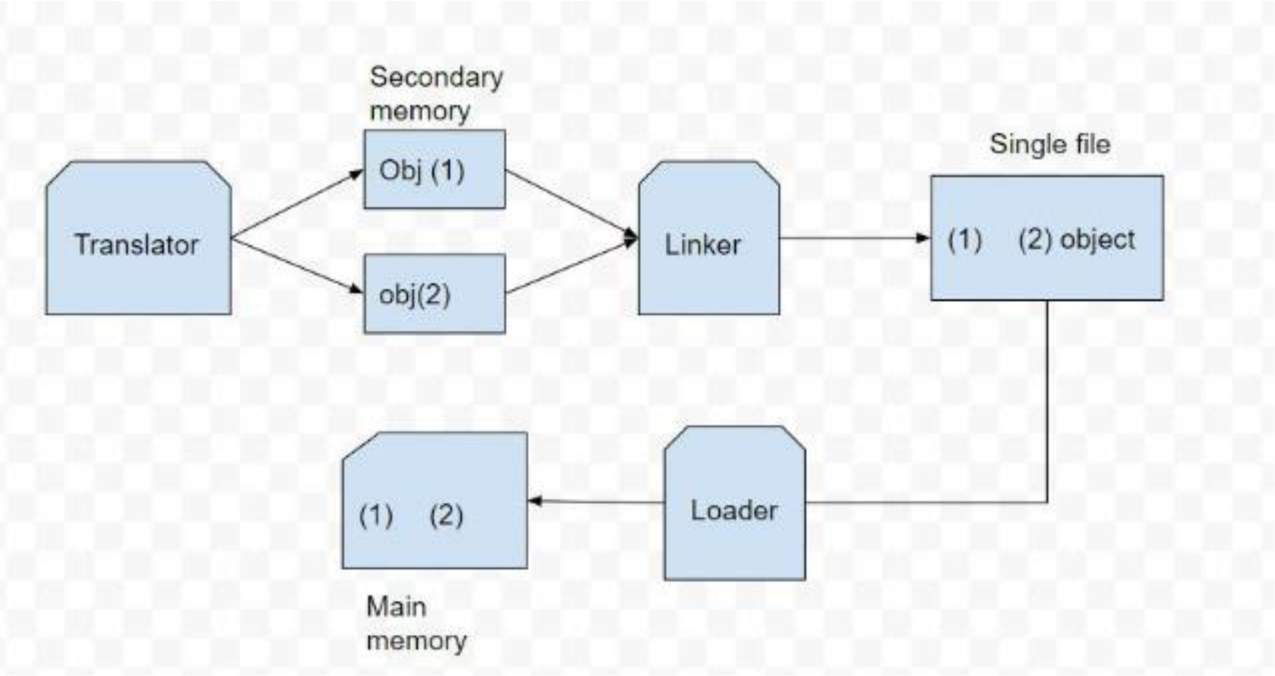
General loading scheme

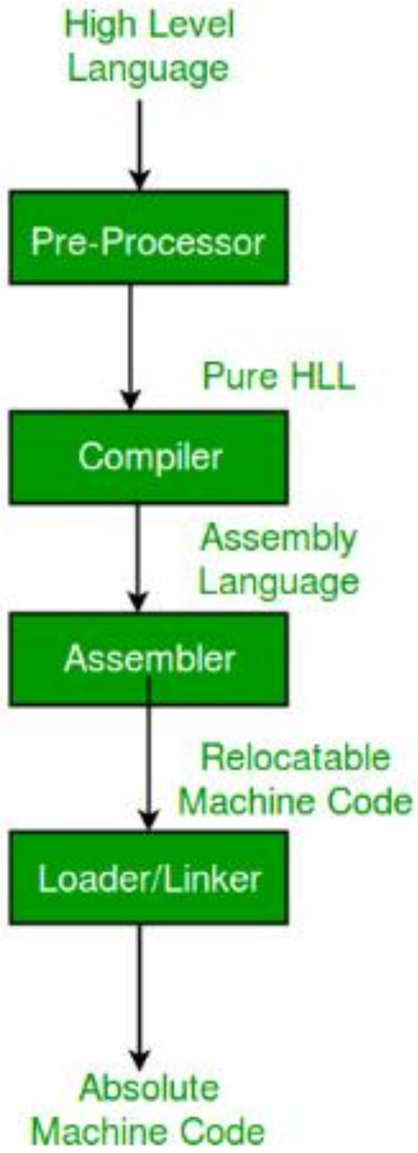
Loader (continued)

- Loader performs its task via four functions, these are as follows:
  - **Allocation**
  - **Loading**
  - **Relocation**
  - **Linking**

Linker

- It combines various object files and libraries into a single executable file.

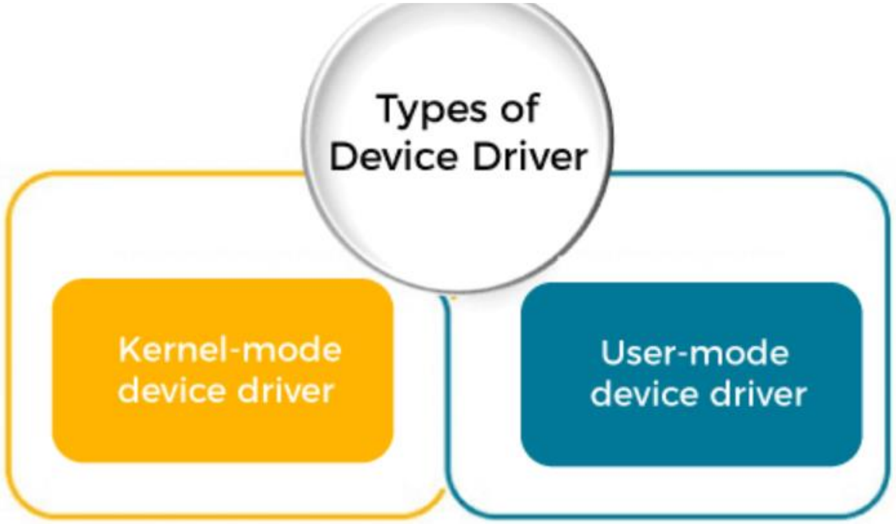




High-Level Language to Machine Code

Device Drivers

- A device driver is defined as a software program
  - without a user interface (UI) that manages hardware components
  - or peripherals attached to a computer and enables them to function with the computer smoothly.



COMPILER	INTERPRETER
A compiler is a program which coverts the entire source code of a programming language into executable machine code for a CPU.	interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met
Generates intermediate object code.	No intermediate object code is generated.
Examples: C, C++, Java	Examples: Python, Perl

Introduction to Operating System

- An **Operating System (OS)** is a category of system software designed to manage a computer's hardware and software resources.
- It acts as an interface between application software and the underlying hardware, ensuring smooth communication and operation.
- Examples of popular operating systems include

Why use an Operating System?

- The operating system helps in improving the computer software as well as hardware. **Without OS, it became very difficult for any application to be user-friendly**.
- The Operating System provides **a user with an interface** that makes any application attractive and user-friendly.
- Today, the operating system provides a **comprehensive platform** that identifies, configures and manages a range of hardware, including processors; memory devices and memory management; chipsets; storage; networking; port communication.

Objectives of an Operating System

- **Convenience:** Make the computer easier to use.
- **Efficiency:** Enable the computer system to operate efficiently by managing hardware and software resources.
- **Ability to Evolve:** Ensure the operating system can be updated to accommodate new hardware and software requirements.

These aspects are served as –

The Operating System as a User/Computer Interface,

The Operating System as Resource Manager

The Operating System as a User/Computer Interface

The operating system acts as a bridge between users and the computer hardware.

It provides:

- **Command-Line Interfaces (CLI)**
- **Graphical User Interfaces (GUI)**
- **APIs (Application Programming Interfaces)**

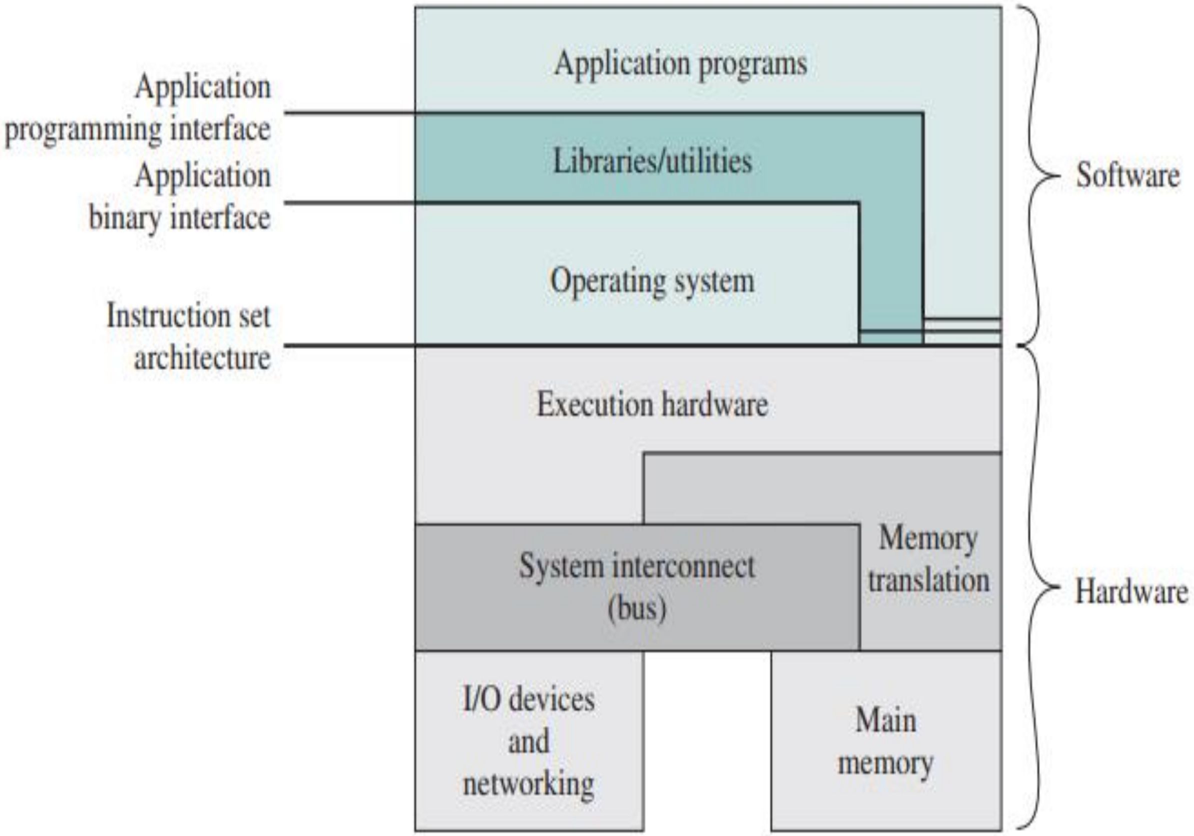


Figure 2.1 Computer Hardware and Software Structure

The Operating System as Resource Manager

As a resource manager, the OS

- **Manages Hardware Resources**
- **Manages Software Resources**
- **Ensures Fair Resource Distribution**

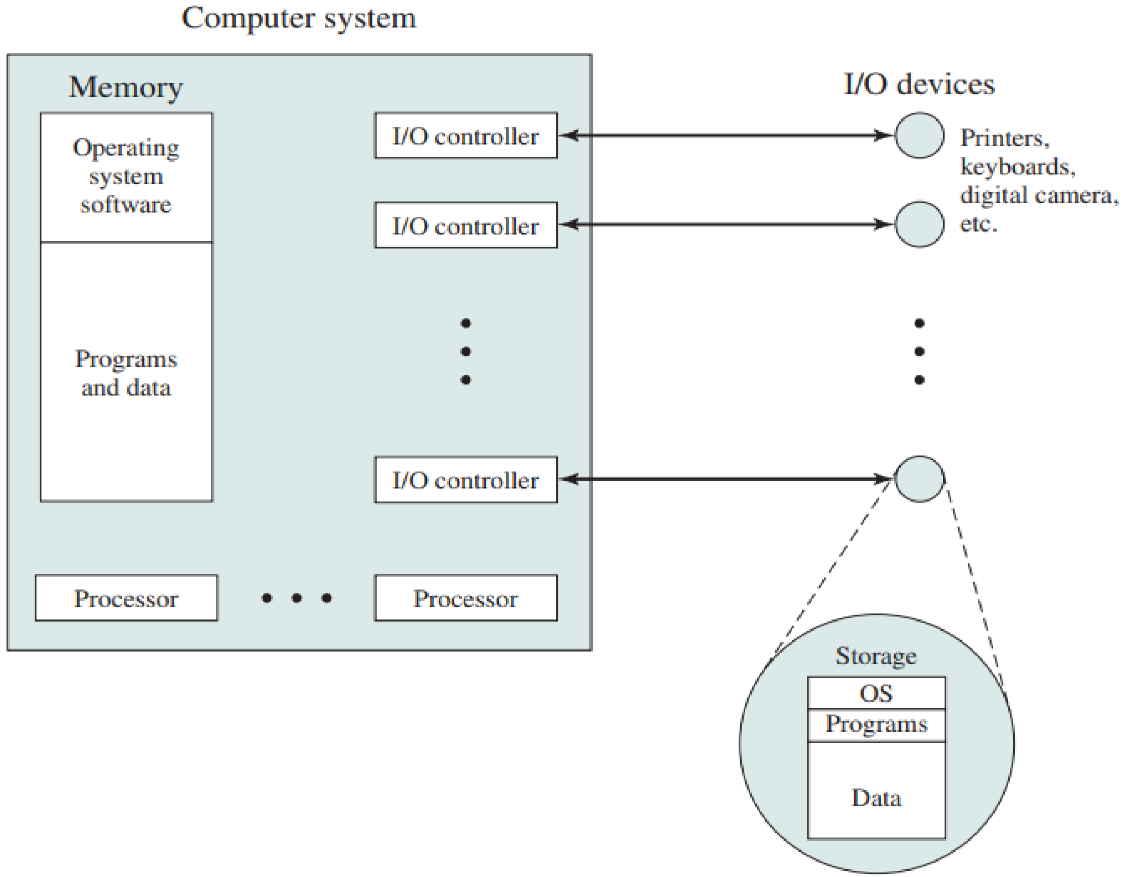


Figure 2.2 **The Operating System as Resource Manager**

Services of Operating System

1. Program Development

- The OS offers tools and utilities, such as editors, debuggers to support implement programs.

2. Program Execution

- Loading instructions and data into main memory.
- Initializing I/O devices and files.
- Allocating required resources.
- This service ensures smooth execution by handling these tasks transparently for the user.

Services of Operating System (continued)

3. Access to I/O Devices

- The OS provides a uniform interface to abstract these complexities, enabling programmers to interact with devices using simple commands like **read** and **write**.

4. Controlled Access to Files

- The OS manages file access, considering:
  - The nature of the I/O device (e.g., disk or tape drive).
  - The data structure within the files.
- For multi-user systems, the OS implements protection mechanisms to regulate access, ensuring secure file usage.

Services of Operating System (continued)

5. Error Detection and Response

- The OS identifies and responds to various errors, such as:
    - **Hardware errors:** Memory faults, device failures, or malfunctions.
    - **Software errors:** Division by zero, access violations, or resource unavailability.
- Responses vary depending on the severity:
    - Terminating the offending program.
    - Retrying the operation.
    - Logging or reporting the error to the user or application.

Services of Operating System (continued)

6. System Access

- For shared or public systems, the OS:
  - Manages user authentication and system-wide access control.
  - Protects resources and data from unauthorized access.
  - Resolves resource contention among users.

7. Accounting

- The OS monitors and collects statistics about resource usage and performance parameters like response time. In multi-user systems, these metrics can be used for:
  - Billing purposes.
  - Identifying areas for system optimization and future upgrades.

Operating System Services (continued)

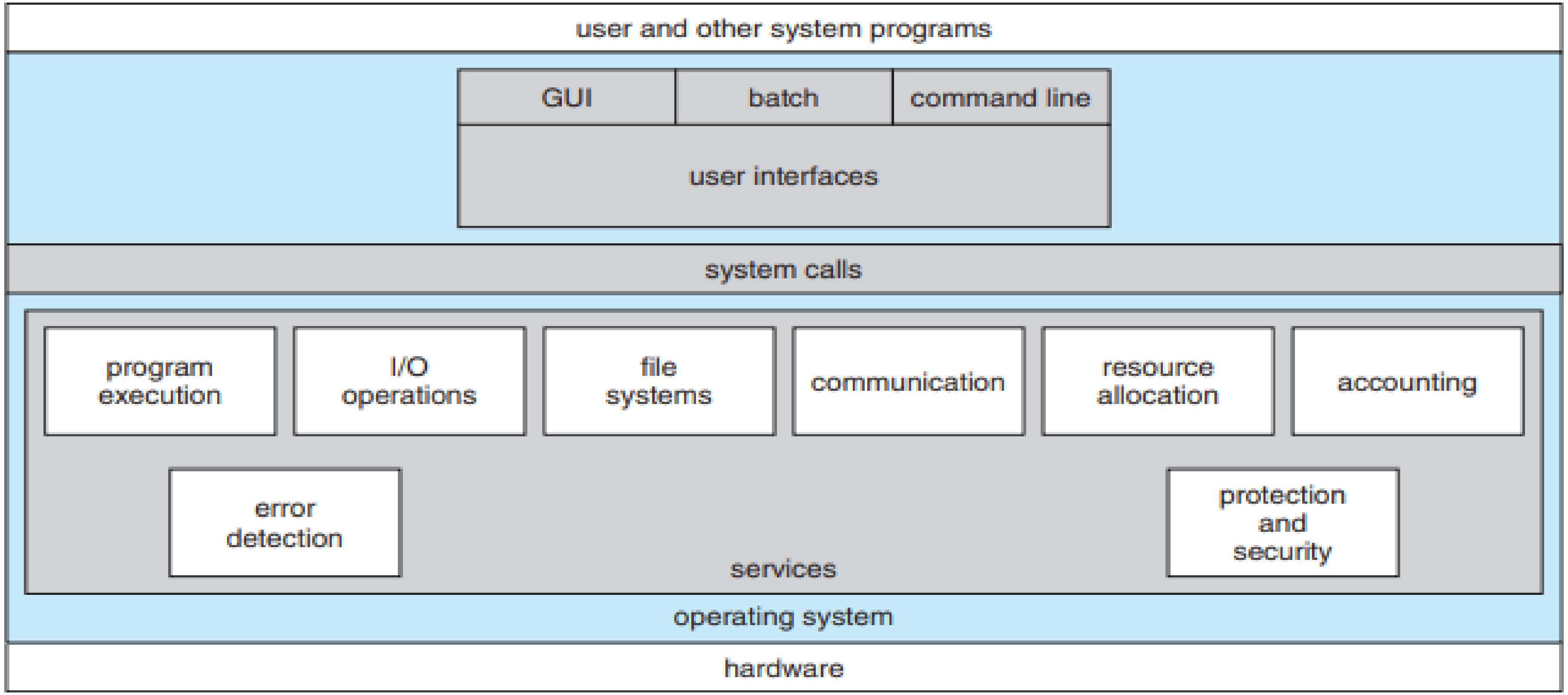


Figure 2.1 A view of operating system services.

Functions of Operating System

- Memory Management
- Process Management
- Device Management
- File Management
- I/O Management
- User Interface
- Security

Evolution of OS

1. **Early Systems (1940s-1950s) - Batch Processing Systems**:
   - These systems processed one job at a time in a batch mode.
   - Jobs were collected and executed sequentially without user interaction.
   - Examples: IBM 701, UNIVAC.
2. **Simple Batch Systems (1950s-1960s) - Resident Monitor:**
   - An early form of the OS that resided in memory to handle job sequencing.
   - Jobs were processed in batches with minimal manual intervention.
   - Examples: IBM 1401.
3. **Single-User, Single-Tasking Systems (1960s-1970s):**
4. **Single-User, Multiprocessing Systems (1970s-1980s):**
   - Examples: Early versions of macOS (Classic Mac OS), MS-DOS with TSR (Terminate and Stay Resident) programs.

Evolution of OS (continued0

5. **Multiprogramming Systems (1960s):**
   5. Allowed multiple programs to be loaded into memory and executed concurrently by the CPU.
   6. Examples: IBM System/360, CTSS (Compatible Time-Sharing System).
6. **Time-Sharing Systems (1960s-1970s):**
   5. Examples: MULTICS (Multiplexed Information and Computing Service), Unix.
7. **Multi-User, Multiprocessing Systems (1970s-1980s):**
   5. Examples: Unix, VMS (Virtual Memory System).
8. **Personal Computer Operating Systems (1980s-1990s) - Single-User, Multiprocessing:**
   5. Graphical User Interfaces (GUIs) gained popularity, making computers more accessible.
   6. Examples: Windows 95, Mac OS.

Evolution of OS (continued)

9. **Network Operating Systems (1980s-1990s)**:
   9. Enabled file sharing, printer sharing, and inter-process communication over networks.
   10. Examples: Novell NetWare, Windows NT, early Unix-based systems with networking extensions.
10. **Distributed Operating Systems (1990s):**
    9. Managed independent computers as a single coherent system.
    10. Examples: Amoeba, Plan 9, early versions of Linux with distributed capabilities.
11. **Modern Operating Systems (2000s-Present) - Multi-User, Multiprocessing Systems:**
    - Support advanced multitasking, multi-user environments, and extensive networking.
    - Emphasize security, stability, and user-friendly interfaces.
    - Examples: Modern Windows, macOS, Linux distributions, Android, iOS.

Evolution of OS (continued)

12. Mobile Operating Systems (2000s-Present):

12. Designed for mobile devices with touch interfaces, efficient power management, and connectivity.
13. Emphasize app ecosystems and seamless user experiences.
14. Examples: iOS, Android.

13. Cloud and Virtualization (2010s-Present):

- Examples: VMware, Hyper-V, Kubernetes, cloud-based OS like Google Chrome OS.

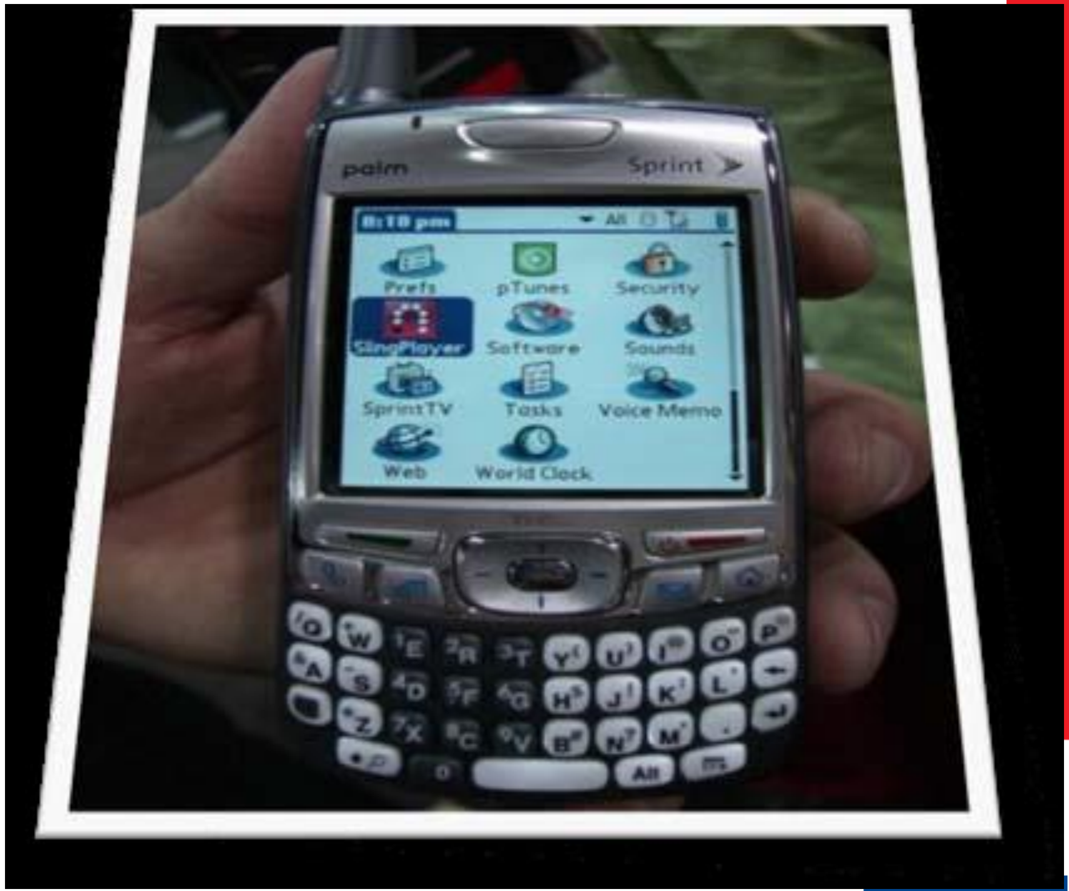
Evolution of Operating System

- Within the broad family of operating systems, there are generally seven types, categorized based on the types of computers they control and the sort of applications they support.
- The categories are
  - Single User Single Task
  - Single User Multi-Tasking
  - Multi-user
  - RTOS
  - Distributed
  - Multiprocessing
  - Parallel

Single User Single Task

Designed for one user to perform one task at a time.

- **Example**
  - **Mobile phone -** There can only be one user using the mobile and that person is only using one of its applications at a time.
  - **MS-DOS**: The user can only execute one program at a time, such as typing in Notepad or executing a command in the terminal.

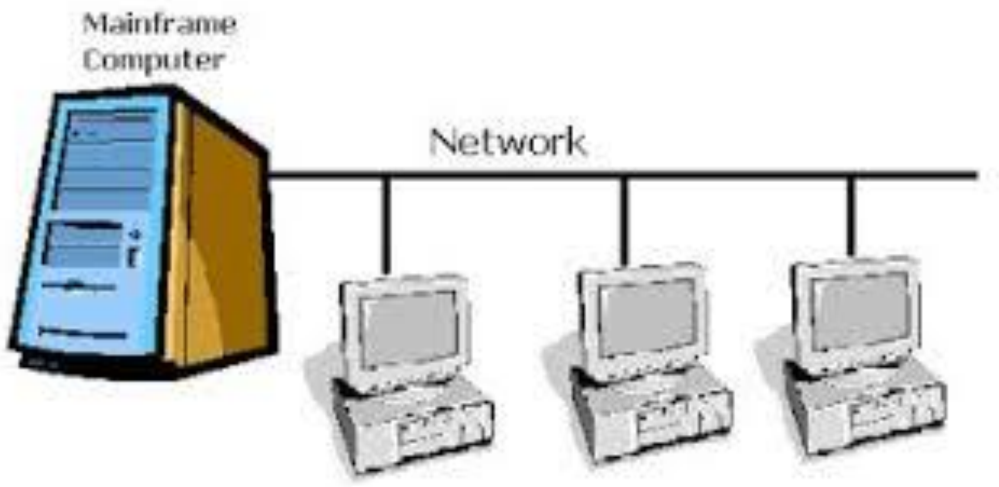


Single User Multi-tasking

- Allows one user to run multiple applications simultaneously.
- **Example**:
  - **Microsoft Windows**: A user can browse the internet, write a document in MS Word, and play music at the same time.
  - **MacOS**: Similarly supports multitasking for a single user.

Multi-user

- Enables multiple users to access a single system's resources simultaneously.
- **Example**:
  - **Unix/Linux Servers**: Multiple users can log in via terminals and execute tasks like programming or file management concurrently.

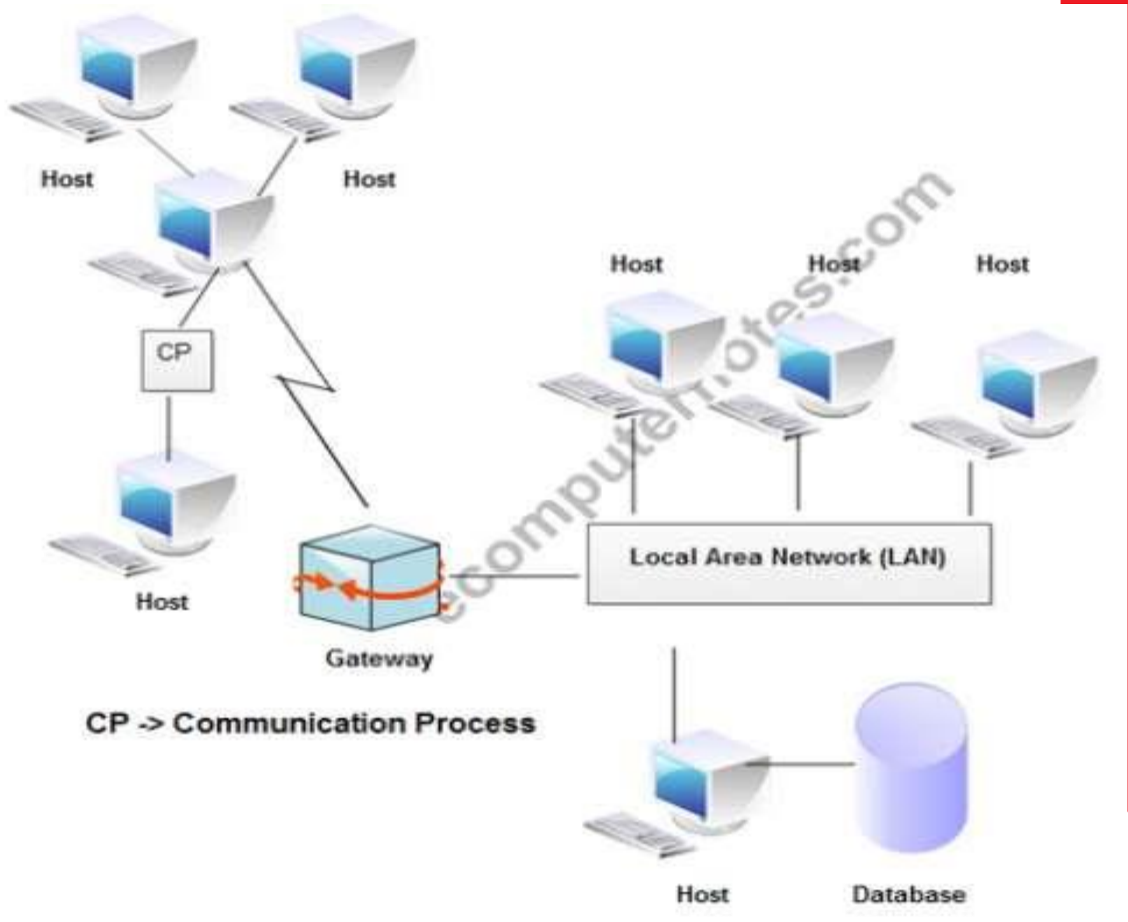


Real Time Operating System (RTOS)

- Processes data and provides responses within a strictly defined time frame. Commonly used in time-critical systems.
- **Example**:
  - **FreeRTOS**: Used in embedded systems.
    - Amaxon Echo, Fitbit, Tesla
  - **VxWorks**: Used in aerospace systems.
    - Mars Rover, Boeing 787

Distributed

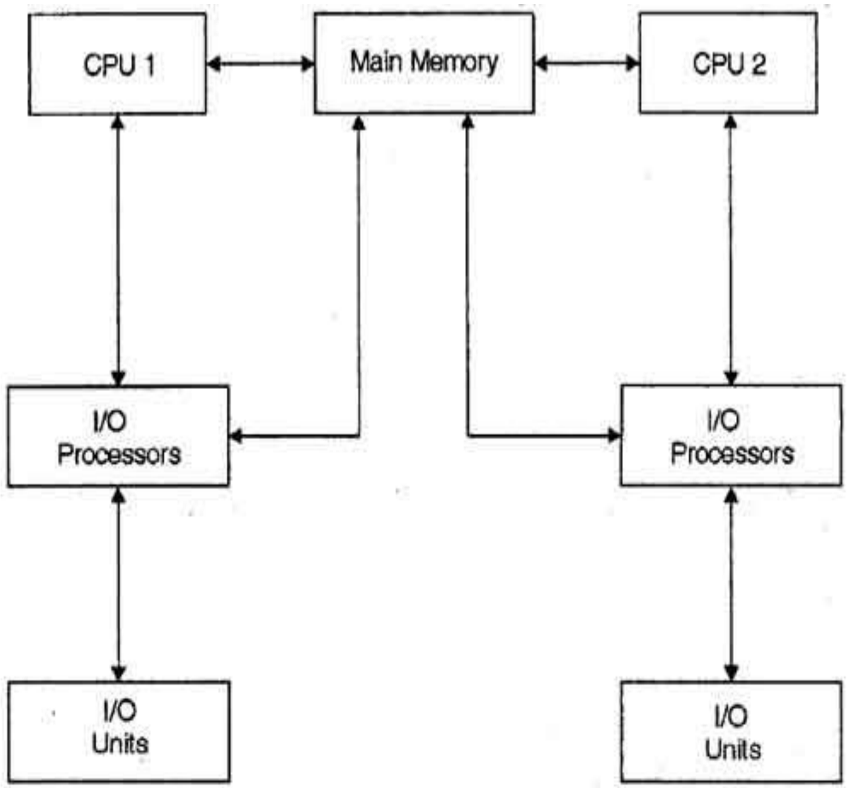
- A system where computing resources are distributed across multiple machines but appear as a single cohesive system to the user.
- **Example**:
  - **Google's Android System**: Combines cloud and local resources.
  - **Windows Server with Distributed Computing**: Data and processes are shared among multiple nodes.



A Typical View of Distributed System

Multiprocessing

- Utilizes multiple CPUs for faster execution of tasks.
- **Example**:
  - **Linux SMP (Symmetric Multiprocessing)**: Allows efficient distribution of tasks across multiple CPUs.
  - **Unix-based Systems**: Designed for multiprocessing.



Parallel

- Specifically designed for systems that execute tasks in parallel using multiple processors.
- **Example**:
  - **IBM's Blue Gene**: Used in supercomputers for high-performance computing.
  - **Cray OS**: Designed for parallel computation in advanced research.

Types of Architectures - Operating System

- Monolithic Architecture
- Layered Architecture
- Micro-Kernel Architecture
- Hybrid Architecture

Monolithic Architecture

- Each component of the **operating system is contained in the kernel** and the components of the operating system **communicate with each other using function calls**.
- E.g. OS/360, VMX, UNIX and LINUX.

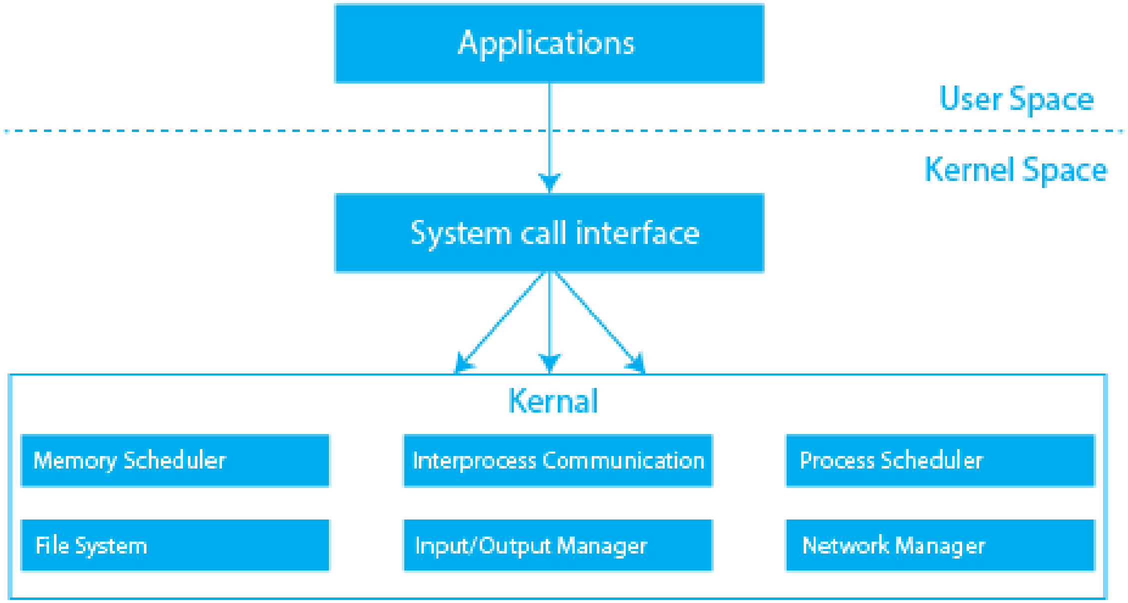
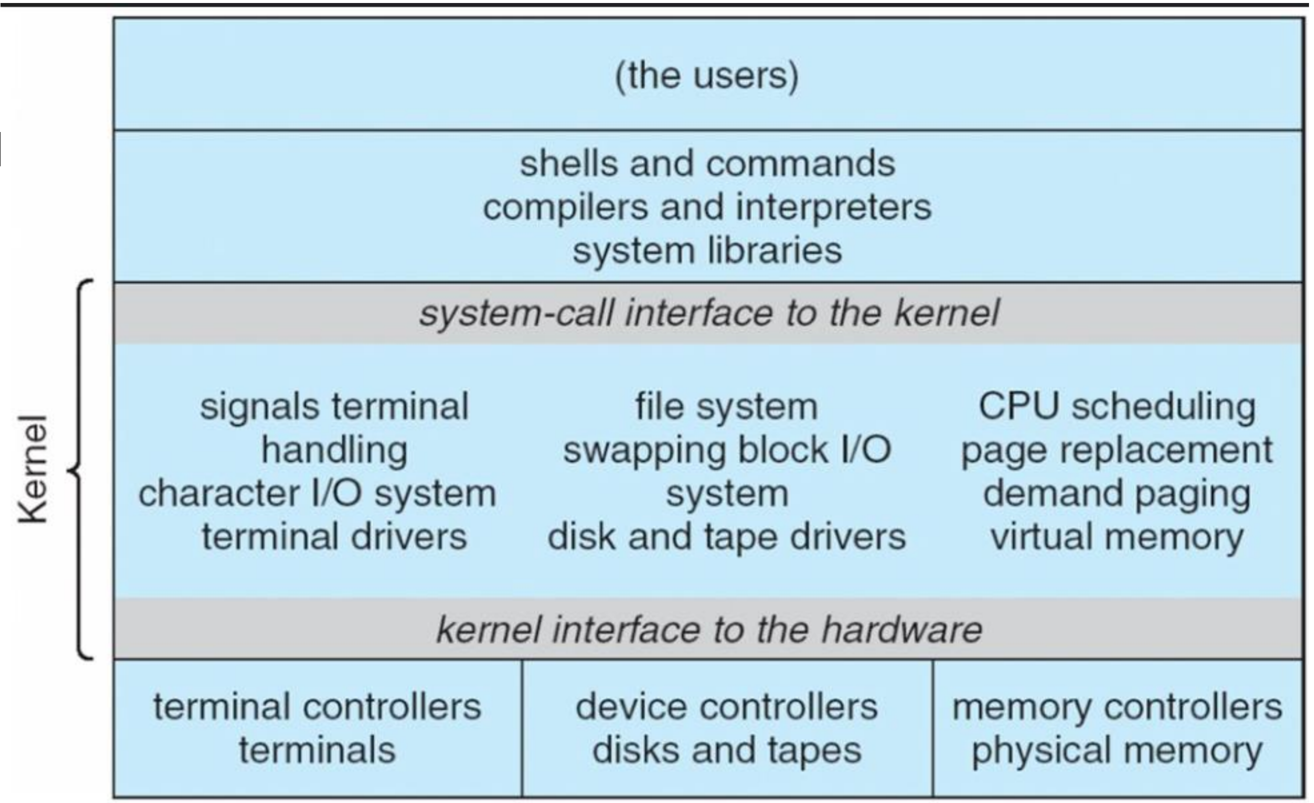


Image source : [Operating System Architecture (prepbytes.com)]

UNIX

- The original UNIX operatin system had limited structuri and was limited hardware functionality
- The UNIX OS
  - System Programs
  - The Kernel



Monolithic Architecture (cont…)

- **Performance**:
  - High due to direct function calls within the kernel.
- **Modularity**:
  - Low, as all components are intertwined in a single large codebase.
- **Maintenance**:
  - Difficult, because changes in one part can affect many others.
- **Security**:
  - Lower, because a bug in any part of the kernel can compromise the entire system.

Layered Architecture

- The OS is separated into layers or levels in this kind of arrangement.
- Layer 0 (the lowest layer) contains the hardware, and layer 1 (the highest layer) contains the user interface (layer N).
- These layers are organized hierarchically, with the top-level layers making use of the capabilities of the lower-level ones.
- E.g. Windows XP, and LINUX

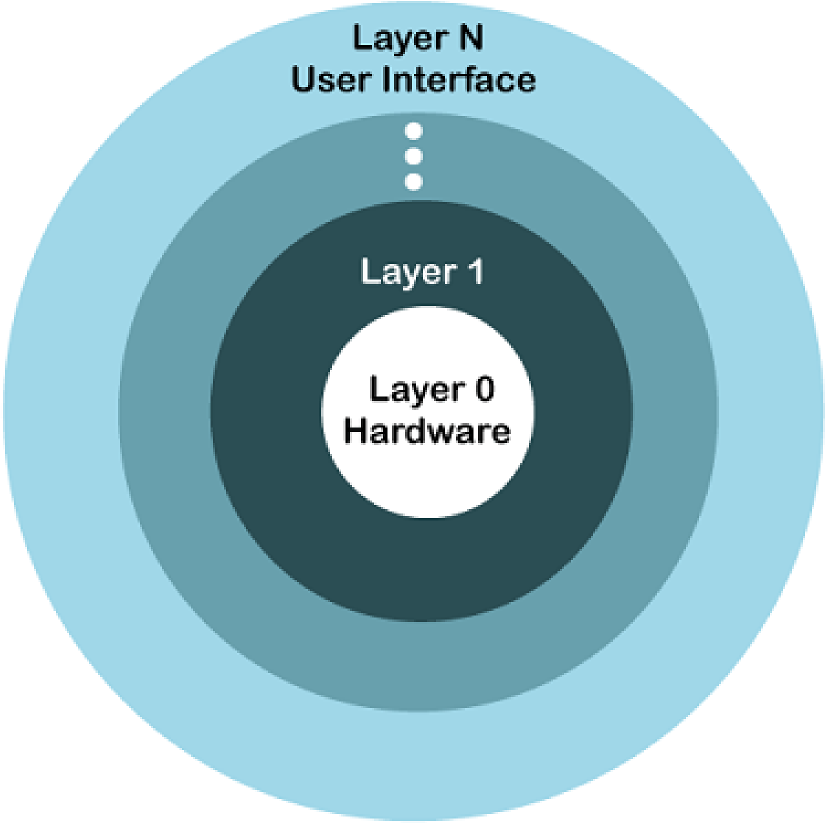


Image source : [Operating System Structure - javatpoint]

Layered Architecture (cont…)

- **Performance**:
  - Moderate, as each layer adds overhead.
- **Modularity**:
  - Higher than monolithic, since each layer has specific functionality.
- **Maintenance**:
  - Easier than monolithic, as changes are localized within layers.
- **Security**:
  - Improved compared to monolithic, but a bug in lower layers can still affect upper layers.

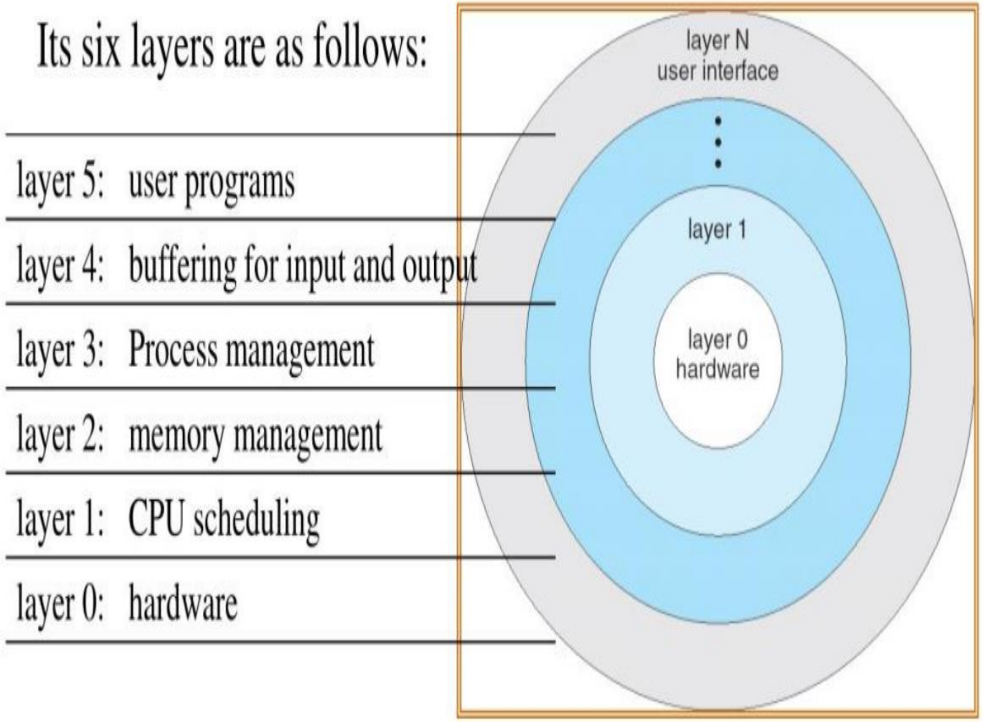


Image source : [Information to know about Operating System – Programmer Prodigy (code.blog)]

Microkernel Architecture

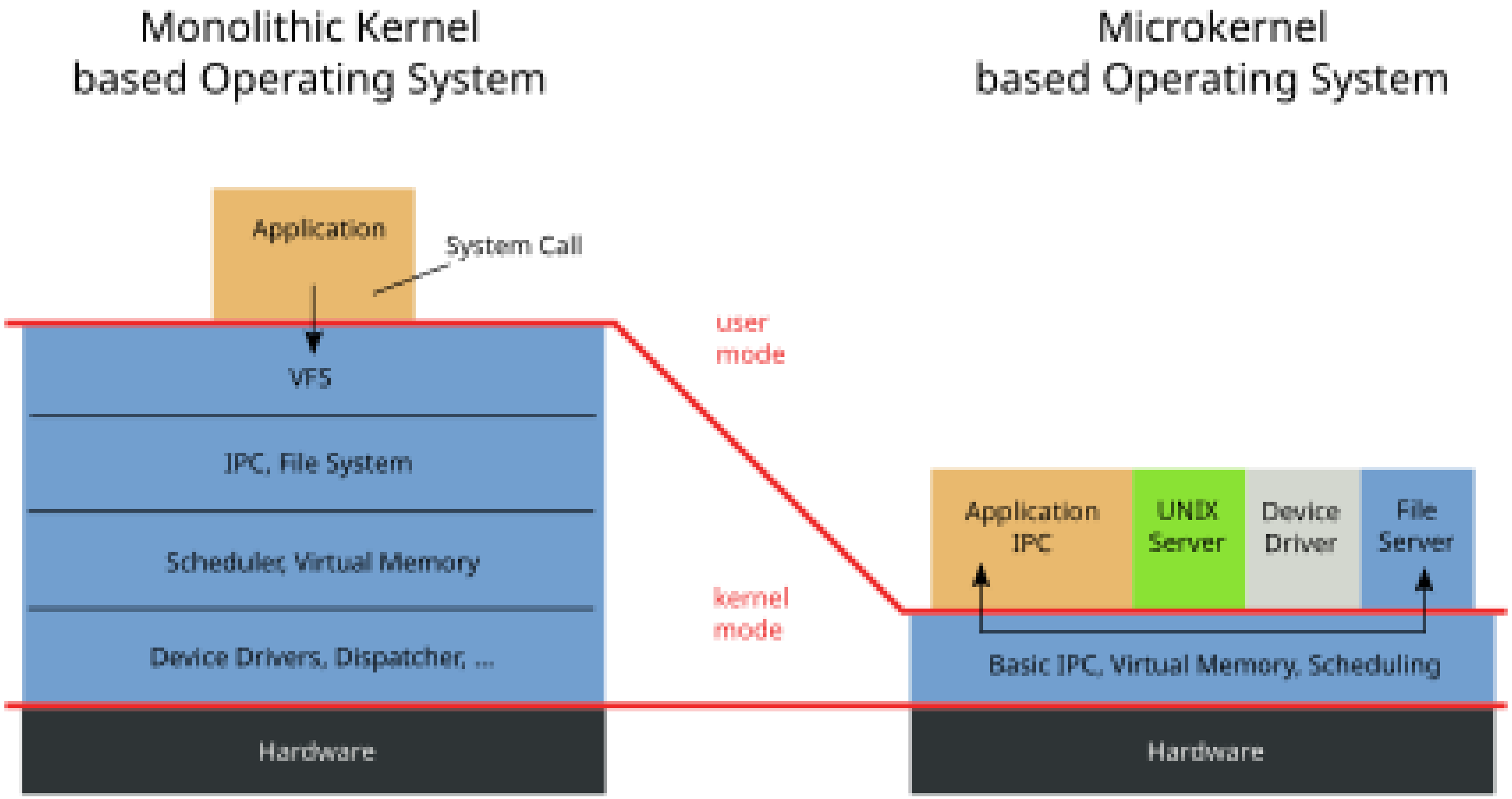


Image source: Wikipedia

Microkernel Architecture

- **Performance**:
  - Lower due to more context switches and inter-process communication.
- **Modularity**:
  - Very high, as only essential services are in the kernel and others run in user space.
- **Maintenance**:
  - Easiest, as most services are user-space programs that can be updated independently.
- **Security**:
  - Highest, because faults in user-space services do not affect the kernel.

Microkernel vs Monolithic

S. No.	Parameters	Microkernel	Monolithic kernel
1.	**Address Space**	In microkernel, user services and kernel services are kept in separate address space.	In monolithic kernel, both user services and kernel services are kept in the same address space.
2.	**Design and Implementation**	OS is complex to design.	OS is easy to design and implement.
3.	**Size**	Microkernel are smaller in size.	Monolithic kernel is larger than microkernel.
4.	**Functionality**	Easier to add new functionalities.	Difficult to add new functionalities.
5.	**Coding**	To design a microkernel, more code is required.	Less code when compared to microkernel

Microkernel vs Monolithic (cont…)

S. No.	Parameters	Microkernel	Monolithic kernel
6.	**Failure**	Failure of one component does not effect the working of micro kernel.	Failure of one component in a monolithic kernel leads to the failure of the entire system.
7.	**Processing Speed**	Execution speed is low.	Execution speed is high.
8.	**Extend**	It is easy to extend Microkernel.	It is not easy to extend monolithic kernel.
9.	**Communication**	To implement IPC messaging queues are used by the communication microkernels.	Signals and Sockets are utilized to implement IPC in monolithic kernels.
10.	**Debugging**	Debugging is simple.	Debugging is difficult.

Microkernel vs Monolithic (cont…)

S. No.	Parameters	Microkernel	Monolithic kernel
11.	**Maintain**	It is simple to maintain.	Extra time and resources are needed for maintenance.
12.	**Message passing and Context switching**	Message forwarding and context switching are required by the microkernel.	Message passing and context switching are not required while the kernel is working.
13.	**Services**	The kernel only offers IPC and low-level device management services.	The Kernel contains all of the operating system's services.
14.	**Example**	**Example :** Mac OS X.	**Example :** DOS, classical early versions of BSD, Unix, Solaris, Mac OS, etc.

Hybrid Architecture

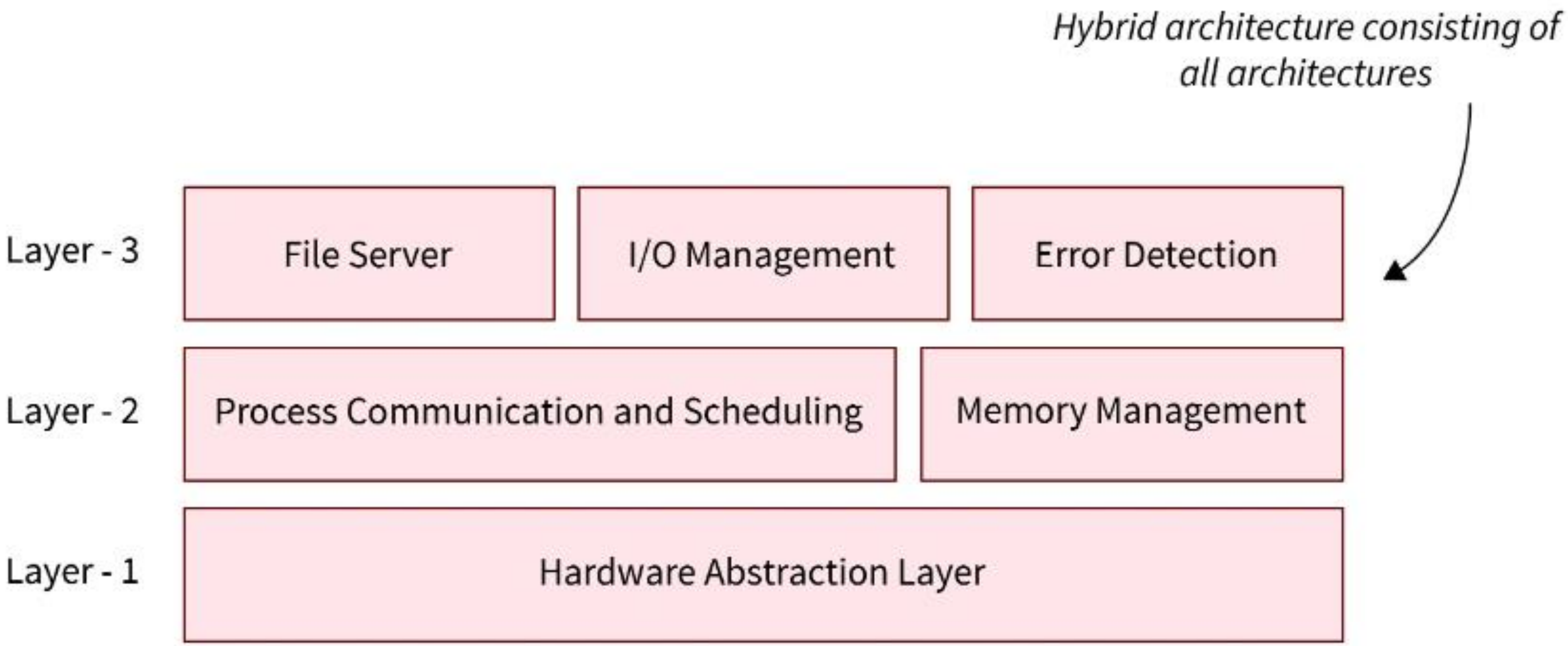


Image source: [Architecture of Operating System - Scaler Topics]

Hybrid Architecture (cont…)

- **Performance:**
  - They allow various architectural components to provide specialized services.
  - This flexibility can lead to better overall system performance.
- **Modularity:**
  - Each layer focuses on specific functionality (e.g., file system, memory management).
  - Developers can work on individual layers independently.
- **Maintenance:**
  - Easier maintenance due to clear module boundaries.
  - Updates or bug fixes can be applied to specific layers without affecting the entire system.
- **Security:**
  - Separating critical services (kernel space) from less critical ones (user space) enhances security.
  - Kernel-level services are protected from user-level code.

Current OS designs

- **Windows NT and successors (2000, XP, Vista, 7, 8, 10, 11)**: Hybrid kernel with microkernel elements.
- **macOS (and iOS, iPadOS, watchOS, tvOS)**: Based on the XNU kernel, combining Mach microkernel and FreeBSD components.
- **Linux distributions**: Monolithic kernel with dynamically loadable modules, offering hybrid-like flexibility.
- **Android**: Built on the Linux kernel, following a similar modular approach.
- **Solaris**: Primarily monolithic but incorporates some microkernel principles with loadable kernel modules.

Question ?