Architecture of Operating system

1

1

Introduction

- This chapter covers the basic functions of an operating system (OS) and key terms.
- Booting is the initial setup process of the OS.
- The OS manages all I/O operations, which users must request through system calls. System calls serve as a communication channel between users and the OS. Different system call types and execution are essential for understanding OS operations.
- The chapter also explores various architectures developed to meet evolving hardware technology requirements.

2

2

General Working Of An Operating System

- The operating system's function begins when the computer system is powered on and continues until it is powered off.
- This section addresses the location of the operating system within the computer system, how it is loaded, and how it initiates.
- Before exploring the operation of an operating system, it is important to grasp some fundamental definitions and concepts.

3

3

BIOS

- The Basic Input-Output System (BIOS) is a software that provides low-level input-output functions for interfacing with various devices such as keyboard, mouse, monitor, and ports. It also includes initialization functions to load the operating system (OS) into memory.
- Initially embedded in ROM or flash-RAM, BIOS has been replaced by more efficient systems like Extensible Firmware Interface (EFI) for modern OSs like Linux and Windows.
- BIOS is now primarily used for booting and initialization, with device drivers handling hardware access during regular system operation.

4

4

Booting Process

- When a system is powered on for the first time, it lacks an operating system (OS). The OS needs to be loaded from secondary storage to memory. This process of loading the OS into memory is called booting.

5

5

Boot Software and Boot Device

- **Boot Software** The set of instructions required for booting, known as boot software or boot loader, is responsible for loading the OS into RAM.
- **Boot Device** The OS is initially stored in non-volatile secondary storage like a hard disk or CD. During booting, the system searches for this storage device to load the OS into RAM. This storage device is referred to as the boot device.

6

6

Privileged Instructions

- Certain operations require direct interaction with hardware devices, which users are not allowed to access. These instructions are passed to the OS, which then interacts with the devices on behalf of the user. These instructions, not directly executed by the user but passed to the OS, are called privileged instructions.

7

7

System Call

- All privileged instructions that interact with hardware and resources and are passed to the OS for execution are known as system calls. For instance, when a user wants to display output on the screen, they include the appropriate instruction in the program, which is a system call.

8

8

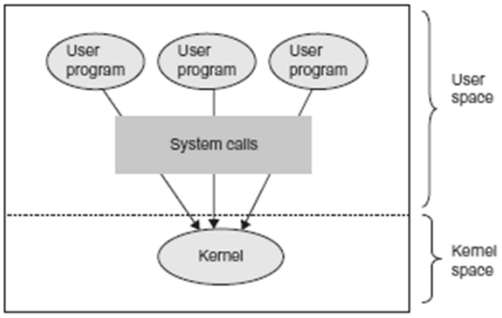
System Call

- System calls play a crucial role in the operation of the operating system by facilitating communication between user programs and the kernel. The system operates in two modes: user mode for executing user processes and system mode for privileged operations.
- User programs and kernel functions run in separate memory partitions. To perform privileged operations, user programs must use system calls as an interface to access kernel-mode functions.
- System calls enable user programs to request the OS to perform operations that require hardware access or other privileged actions, such as accessing files, directories, or communicating with other processes.

9

9

System Call



10

10

System Programs

- **System programs** are utilities that assist users in developing and running applications. They are distinct from system calls, as system programs help users by utilizing system calls. Examples of system programs include file management tools, compilers, debuggers, communication programs, and status information programs.

11

11

System Generation Programs

- **System generation programs** configure the operating system based on the specific hardware specifications of a machine. These programs gather information about the processor type, disk formatting, memory size, CPU and disk scheduling algorithms, and I/O device details. Using this information, the system generation program selects and compiles code modules from a library to create the customized OS for the machine.
  - The processor type and its options selected
  - Disk formatting information, its partitions
  - Size of memory
  - CPU scheduling algorithm
  - Disk scheduling algorithm
  - I/O device type and model, its interrupt number

12

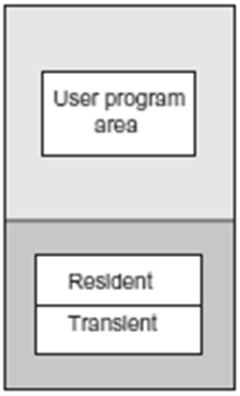
12

General structure of OS

OS was structured into two parts: Resident part or kernel and Transient part.

The resident part may contain the following programs or data structures:

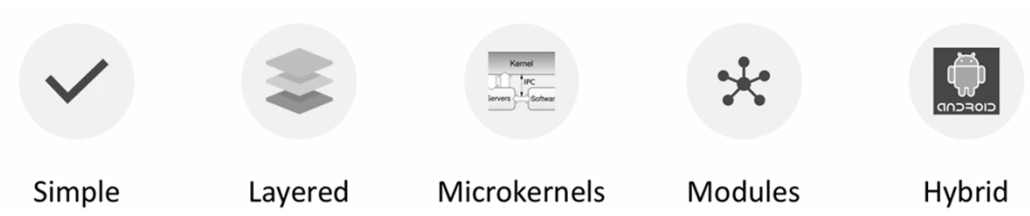
- **Resource Allocation Data Structures.**
- **Information Regarding Processes and their States.**
- **System Call Handler.**
- **Event Handlers**
- **Scheduler.**



13

13

Operating System Structure



14

14

Monolithic Architecture

OSs were developed to meet various requirements, such as CPU busyness, multiple jobs in batch systems, and user friendliness.

However, the initial architecture was not efficient and consisted of few modules due to limited functionality. The kernel only added all functionalities, allowing efficient intercommunication between modules.

This development process was not planned and led to unplanned OS development.

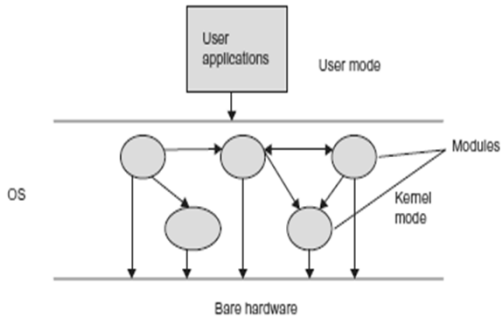


Fig. Monolithic architecture

15

15

Monolithic Architecture

Multi-programming expanded the size of the operating system (OS), leading to a complex structure where every module directly accesses hardware.

Therefore, there is a large gap in understanding the operations at OS level and machine level, as the OS consists of algorithms for process allocation and scheduling, while hardware operations are performed at machine instructions.

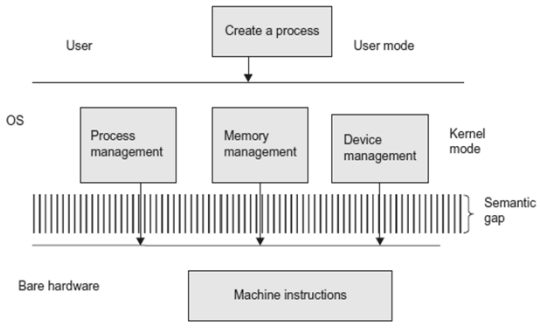


Fig. Semantic gap between the OS and hardware

16

16

Monolithic Architecture

The monolithic architecture of operating systems combined all functionalities in a single layer, making it challenging to modify modules due to extensive interfacing.

Additionally, the lack of protection in this architecture posed a security risk, allowing unrestricted access among modules and potential for malicious code.

This structure was not suitable for multi-programming/multi-tasking environments as any user job could corrupt another job or the operating system.

17

17

Monolithic Architecture

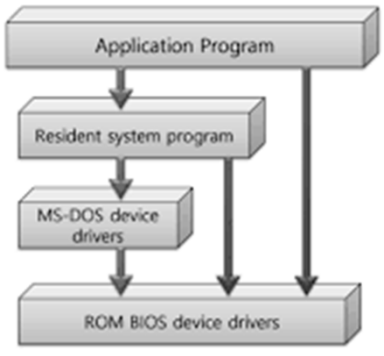
MS-DOS

It was original designed and implemented by a few people.

It was written to provide the most functionality in the least space.

In MS-DOS, the interfaces and levels of functionality are not well separated.

Vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.



18

18

Monolithic Architecture

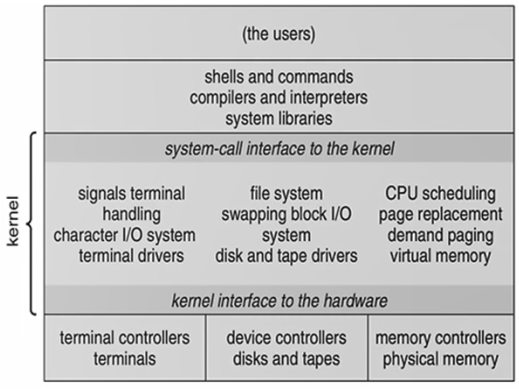
UNIX

Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs.

The kernel is further separated into a series of interface and device drivers.

The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

This monolithic structure was difficult to implement and maintain.



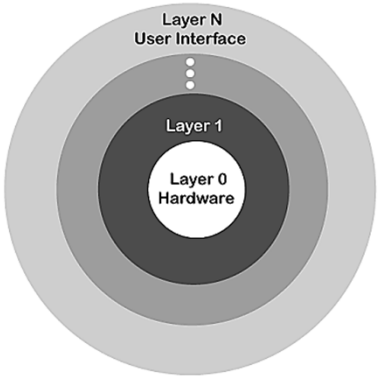
19

19

Layer Architecture

The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



20

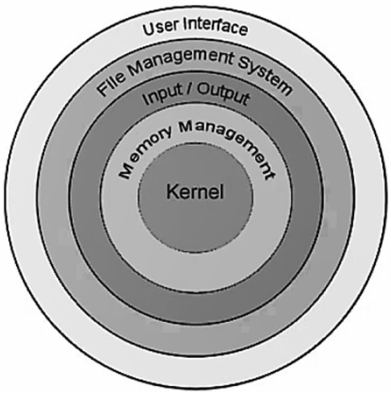
20

Layer Architecture

Less efficient.

For instance, when a user program executes an I/O operation. it executes a system call that is trapped to the I/O layer, which calls the memory management layer, which in turn calls the CPU scheduling layer, which is then passed to the hardware.

Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a non-layerd system.



21

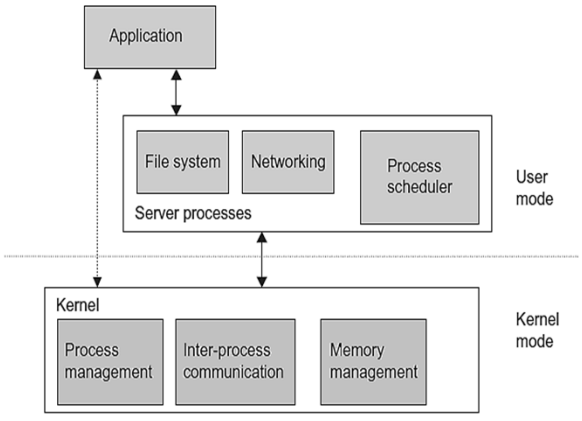
21

Microkernel Architecture

Removes all nonessential components from the kernel and implements them as system and user-level programs. The result is a small kernel.

Microkernel provide minimal process and memory management.

The main function of the microkernel is to provide communication between the client program and the services running in user space.



22

22

Microkernel Architecture

Easier to port from one hardware design to another.

The microkernel also provides more security and reliability.

If a service fails, the rest of the OS remains untouched

23

23

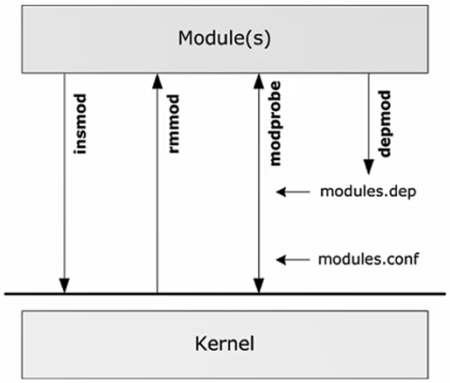
Modules Architecture

The best current methodology for operating system design involves using loadable kernel modules.

The kernel har a set of core components and links in additional services via modules, either at boot time or during run time.

The kernel to provide core services while other services are implemented dynamically, as the kernel is running.

Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.



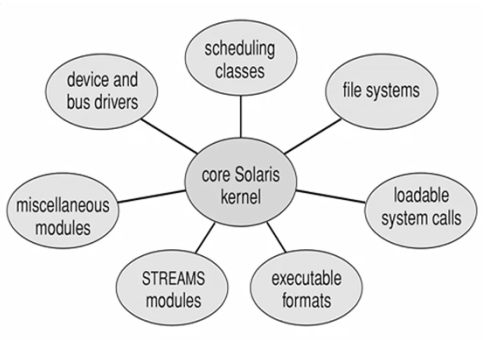
24

24

Modules Architecture

More flexible than a layered system, because any module can call any other module.

More efficient than a microkernel, because modules do not need to invoke message passing in order to communication.



25

25

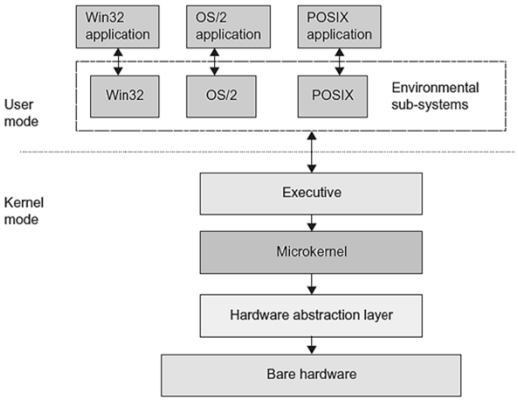
Hybrid kernel Architecture

In practice, very few operating systems adopt a single, strictly defined structure.

They combine different structures, resulting in hybrid systems that address performance, security, and usability issues.

Three hybrid systems:

- Apple Mac OS X
- iOS
- Android

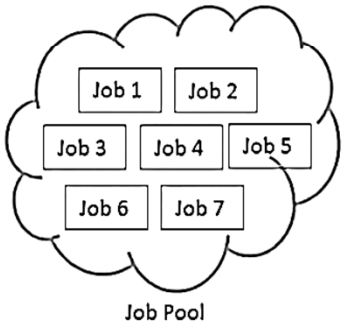


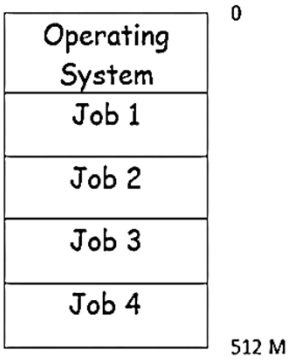
26

26

(i) Multiprogramming

- A single user cannot, in general, keep either the CPU or the I/O devices busy at all times
- Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.





Memory layout for a multiprogramming system

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the Computer system.

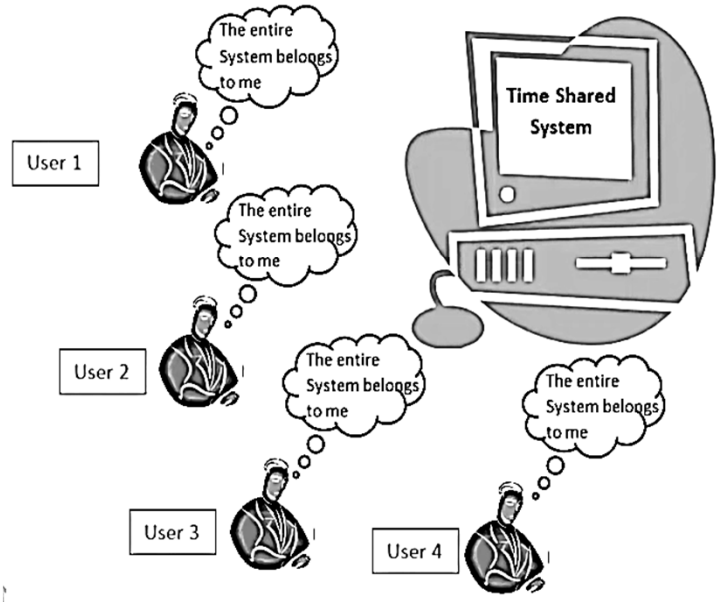
27

(ii) Time Sharing (Multitasking)

- CPU executes multiple jobs by switching among them
- Switches occur so frequently that the users can interact with each program while it is running
- Time sharing requires an interactive (or hands-on) computer system, which provides direct communication between the user and the system.
- A time-shared operating system allows many users to share the computer simultaneously.

28

- A time-shared operating system allows many users to share the computer simultaneously.



- Uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.
- Each user has at least one separate program in memory
- A program loaded into memory and executing is called a "PROCESS"

29