

OPERATING SYSTEM

Chapter 2 Operating System Structures

Study notes for Engineering students

Section	Topic
2.1	System components: process management, main memory management, file management, I/O system management, secondary storage management
2.2	Operating system services
2.3	System calls: uses, process control, file management, device management, information maintenance, communication
2.4	Operating system structure: simple, layered, monolithic, microkernel
2.5	Booting
2.6	Virtual machine
	Practice: 15 MCQs with answer key and 10 broad questions

Format of these notes: each topic is explained in paragraph form, followed by lists for quick revision, tables for comparison and diagrams where a picture explains the idea better than words.

Chapter Overview

An operating system (OS) is system software that acts as an intermediary between the user of a computer and the computer hardware. It manages the hardware resources (CPU, memory, files and devices), provides a convenient environment for running programs and makes the machine efficient to use. A modern operating system is one of the largest and most complicated pieces of software ever written, so it cannot be understood as a single block. This chapter therefore studies the operating system from several viewpoints.

Viewpoint	Question it answers	Section
Components	What does the OS manage? (processes, memory, files, I/O devices, disks)	2.1
Services	What does the OS offer to users and programs?	2.2
System calls	How does a program request those services from the kernel?	2.3
Structure	How is the OS internally organised? (simple, monolithic, layered, microkernel)	2.4
Booting	How does the OS get loaded when the computer is switched on?	2.5
Virtual machine	How can one computer run several operating systems at once?	2.6

2.1 System Components

An operating system is large and complex, so it is designed by dividing it into smaller **components**. Each component is a well-defined portion of the system with carefully specified inputs, outputs and functions. Not every operating system has exactly the same components, but almost all of them contain the five components listed below, together with a few supporting ones (protection, networking and a command interpreter).

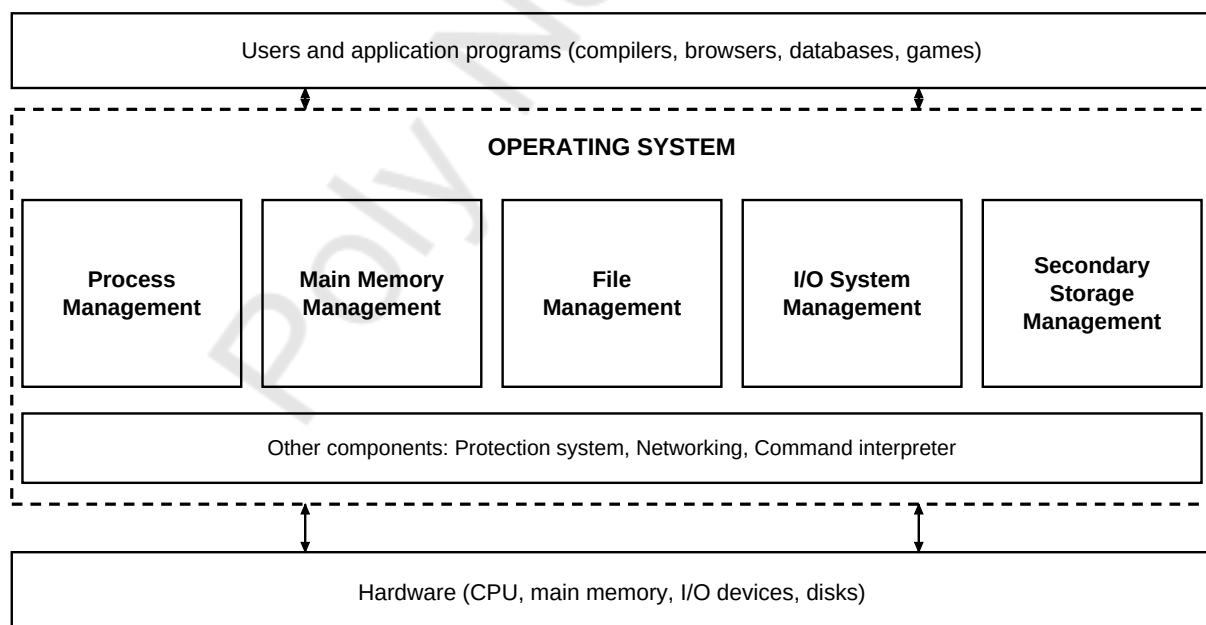


Fig 2.1: Main components of an operating system

2.1.1 Process Management

A **program** is a passive entity: a set of instructions stored on disk. A **process** is a program in execution, an active entity. A process needs resources to finish its job: CPU time, memory, files and I/O devices. These resources are given to the process either when it is created or while it is running. A single-threaded process has one program counter that tells which instruction is to be executed next; a multi-threaded process has one program counter per thread. A system contains both user processes (executing user code) and operating-system processes (executing system code). All of them can run concurrently, because the CPU is multiplexed among them.

Basis	Program	Process
Nature	Passive entity (a file of instructions)	Active entity (a program in execution)
Resources	Needs only disk space to be stored	Needs CPU, memory, files and I/O devices
Location	Resides in secondary storage	Resides in main memory while it runs
Lifetime	Exists until it is deleted	Exists from creation until termination
Relation	One program can create many processes	Each process is one execution of a program

Memory layout of a process has four parts:

- **Text section:** the executable program code.
- **Data section:** global and static variables.
- **Heap:** memory allocated dynamically while the process runs (malloc, new).
- **Stack:** temporary data such as function parameters, return addresses and local variables.

Responsibilities of the operating system in process management:

- Creating and deleting both user and system processes.
- Scheduling processes and threads on the CPU.
- Suspending and resuming processes.
- Providing mechanisms for process synchronisation.
- Providing mechanisms for process communication.
- Providing mechanisms for deadlock handling.

Process states

As a process executes, it changes state. At any instant a process is in exactly one of the five states shown in Fig 2.2. On a single-processor system only one process can be running at a time, but many can be ready or waiting.

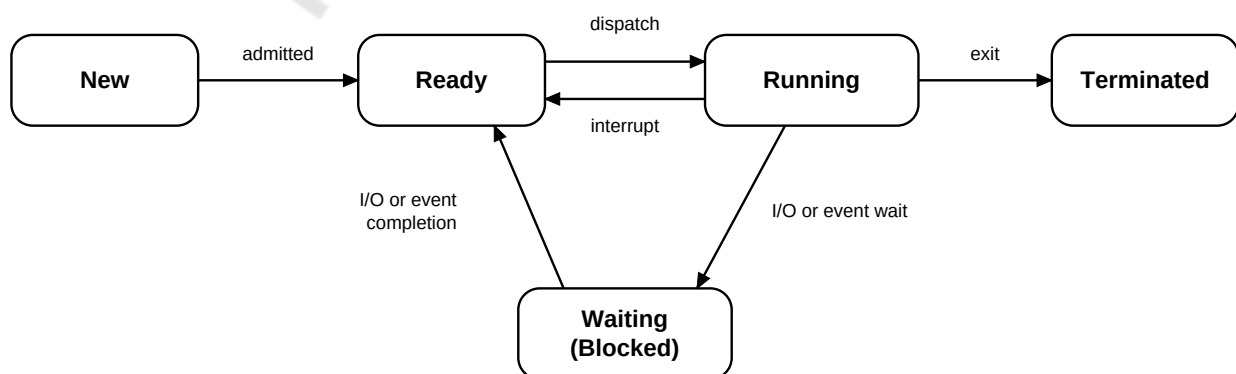


Fig 2.2: Process state diagram

State	Meaning
New	The process is being created.
Ready	The process is in memory and waiting to be assigned to the CPU.
Running	Instructions of the process are being executed by the CPU.
Waiting (Blocked)	The process is waiting for some event, such as completion of an I/O operation or a signal.
Terminated	The process has finished execution and its resources are being reclaimed.

Process Control Block (PCB)

The operating system keeps all the information about a process in a data structure called the **Process Control Block**, also known as the task control block. The PCB is created when the process is created and is the only evidence of the process for the OS; during a context switch the state of the old process is saved in its PCB and the state of the new process is loaded from its PCB.

PCB field	Information stored
Process ID (PID)	Unique number identifying the process
Process state	New, ready, running, waiting or terminated
Program counter	Address of the next instruction to execute
CPU registers	Accumulators, index registers, stack pointer and other registers saved on an interrupt
CPU-scheduling information	Priority, pointers to scheduling queues
Memory-management information	Base and limit registers, page tables or segment tables
Accounting information	CPU time used, time limits, account numbers
I/O status information	List of open files and I/O devices allocated to the process

2.1.2 Main Memory Management

Main memory (RAM) is a large array of words or bytes, each with its own address. It is a **volatile** storage device that is shared by the CPU and the I/O devices. The CPU fetches instructions from main memory and reads or writes data in it. For a program to be executed it must be mapped to absolute addresses and loaded into main memory. To improve CPU utilisation and the response time to users, several programs are kept in memory at the same time, so the operating system needs memory-management schemes. The choice of scheme depends mainly on the hardware design of the system.

Responsibilities of the operating system in memory management:

- Keeping track of which parts of memory are currently being used and by whom.
- Deciding which processes are to be loaded into memory when memory space becomes available.
- Allocating and de-allocating memory space as per need.

Technique	Basic idea
Contiguous allocation	Each process gets one continuous block of memory (fixed or variable partitions).
Paging	Physical memory is divided into fixed-size frames and the process into pages of the same size; pages may be placed in any free frames.

Technique	Basic idea
Segmentation	The process is divided into logical units (code, data, stack) of different sizes called segments.
Virtual memory	Only part of the process is kept in memory; the rest stays on disk and is brought in on demand (demand paging).
Swapping	A process is moved temporarily from memory to a backing store and brought back later.

Related ideas worth remembering:

- **Logical (virtual) address** is generated by the CPU; **physical address** is the actual location seen by the memory unit. The hardware Memory-Management Unit (MMU) maps logical addresses to physical addresses at run time.
- **Internal fragmentation** is unused space inside an allocated block; **external fragmentation** is unused space scattered between allocated blocks.
- Address binding of instructions and data to memory addresses can happen at compile time, load time or execution time.

2.1.3 File Management

Computers store information on different physical media such as magnetic disks, SSDs and tapes, and each medium has its own characteristics. To make the system convenient, the operating system provides a uniform **logical** view of stored information: the **file**. A file is a collection of related information defined by its creator and mapped by the OS onto physical devices. Files may contain programs (source or object form) or data, and the data may be numeric, alphabetic, alphanumeric or binary. Files are organised into **directories** to make them easy to locate and use. When many users share files, access control (who may read, write or execute) is also needed.

Responsibilities of the operating system in file management:

- Creating and deleting files.
- Creating and deleting directories to organise files.
- Supporting primitives for manipulating files and directories (open, read, write, reposition, close).
- Mapping files onto secondary storage.
- Backing up files onto stable (non-volatile) storage media.

File attribute	Description
Name	The only attribute kept in human-readable form
Identifier	Unique tag (number) used by the file system internally
Type	Needed in systems that support different kinds of files
Location	Pointer to the device and place on the device where the file is stored
Size	Current size of the file in bytes, words or blocks
Protection	Access-control information: who can read, write or execute
Time, date, user ID	Creation, last modification and last use; useful for protection and monitoring

Topic	Common forms
File operations	Create, write, read, reposition (seek) within a file, delete, truncate

Topic	Common forms
Access methods	Sequential access, direct (random) access, indexed access
Directory structures	Single-level, two-level, tree-structured, acyclic-graph, general graph

2.1.4 I/O System Management

One purpose of an operating system is to hide the peculiarities of specific hardware devices from the user. In UNIX, for example, the peculiarities of I/O devices are hidden from the bulk of the operating system by the **I/O subsystem**. This subsystem consists of three parts:

- A memory-management component that includes **buffering, caching and spooling**.
- A general **device-driver interface**.
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of the specific device to which it is assigned; it acts as a translator between the operating system and the device controller. Data can be transferred by **programmed I/O** (the CPU polls the device), **interrupt-driven I/O** (the device interrupts the CPU when it is ready) or **DMA** (direct memory access, where a controller moves a whole block between device and memory without CPU involvement).

Technique	Purpose	Example
Buffering	Temporarily holds data in memory while it is being transferred between two devices or a device and an application, to match their different speeds.	Keyboard input buffer; network packets
Caching	Keeps a copy of frequently used data in a faster storage so that later accesses are quicker. A cache holds a copy, a buffer may hold the only copy.	Disk cache in RAM
Spooling	Simultaneous Peripheral Operations On-Line: output for a device that cannot accept interleaved data is held on disk and sent one job at a time.	Print queue for a printer

2.1.5 Secondary Storage Management

Main memory is too small to hold all programs and data permanently and it loses its contents when power is switched off. A computer system therefore provides **secondary storage** to back up main memory. Most modern systems use disks (hard disks and SSDs) as the principal on-line storage for programs and data, which stay on disk until they are loaded into memory. The overall speed of the computer depends heavily on the speed of the disk subsystem, so its management is very important. Slower, cheaper media such as magnetic tape and optical discs are called **tertiary storage** and are used mainly for backup and archives.

Responsibilities of the operating system in secondary storage management:

- **Free-space management:** keeping track of unused disk blocks (bit vector, linked list, grouping, counting).
- **Storage allocation:** deciding where file data is placed (contiguous, linked or indexed allocation).
- **Disk scheduling:** ordering disk requests to reduce head movement (FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK).

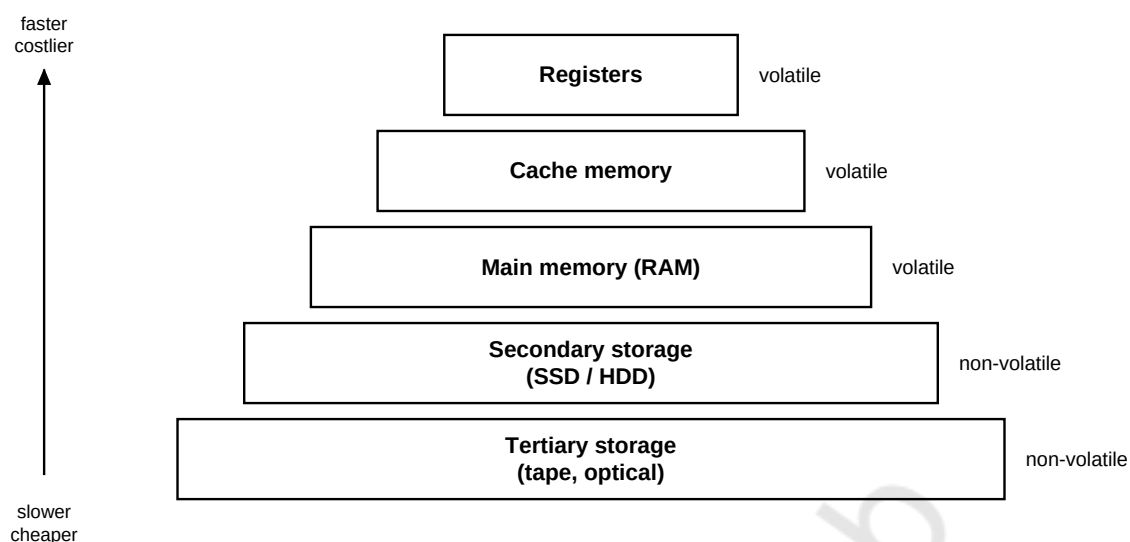


Fig 2.3: Storage hierarchy (speed and cost per bit increase upwards, capacity increases downwards)

Level	Typical capacity	Typical access time (approx.)	Volatile?
Registers	Less than 1 KB	About 1 ns or less	Yes
Cache	KBs to a few MBs	A few ns	Yes
Main memory	GBs	50 to 100 ns	Yes
SSD	Hundreds of GB to TB	Tens to hundreds of microseconds	No
Hard disk	TBs	5 to 10 ms	No
Tertiary (tape, optical)	TBs to PBs	Seconds to minutes	No

Other supporting components

- **Protection system:** controls the access of programs, processes and users to the resources defined by the computer system. It specifies the controls to be imposed and provides a means of enforcement.
- **Networking:** processors in a distributed system are connected by a communication network; the OS provides the mechanisms for remote file access, remote execution and message exchange.
- **Command-interpreter system:** the interface between the user and the OS (the shell). It reads and executes the user commands, which in turn use process, file, I/O and storage management.

Quick revision: 2.1

- Process = program in execution; program is passive, process is active.
- Process states: new, ready, running, waiting, terminated; information is kept in the PCB.
- Memory manager tracks usage, decides which process to load and allocates/de-allocates space.
- File manager creates/deletes files and directories, maps files to storage and takes backups.
- I/O subsystem = buffering + caching + spooling, a general driver interface and device drivers.
- Secondary storage manager handles free space, allocation and disk scheduling.

2.2 Operating System Services

An operating system provides an **environment** for the execution of programs. It offers certain services to programs and to the users of those programs. The exact set of services differs from one OS to another, but they fall into two groups: services that are **helpful to the user**, and services that exist to **ensure the efficient operation of the system itself** (mostly useful when many users or jobs share the machine).

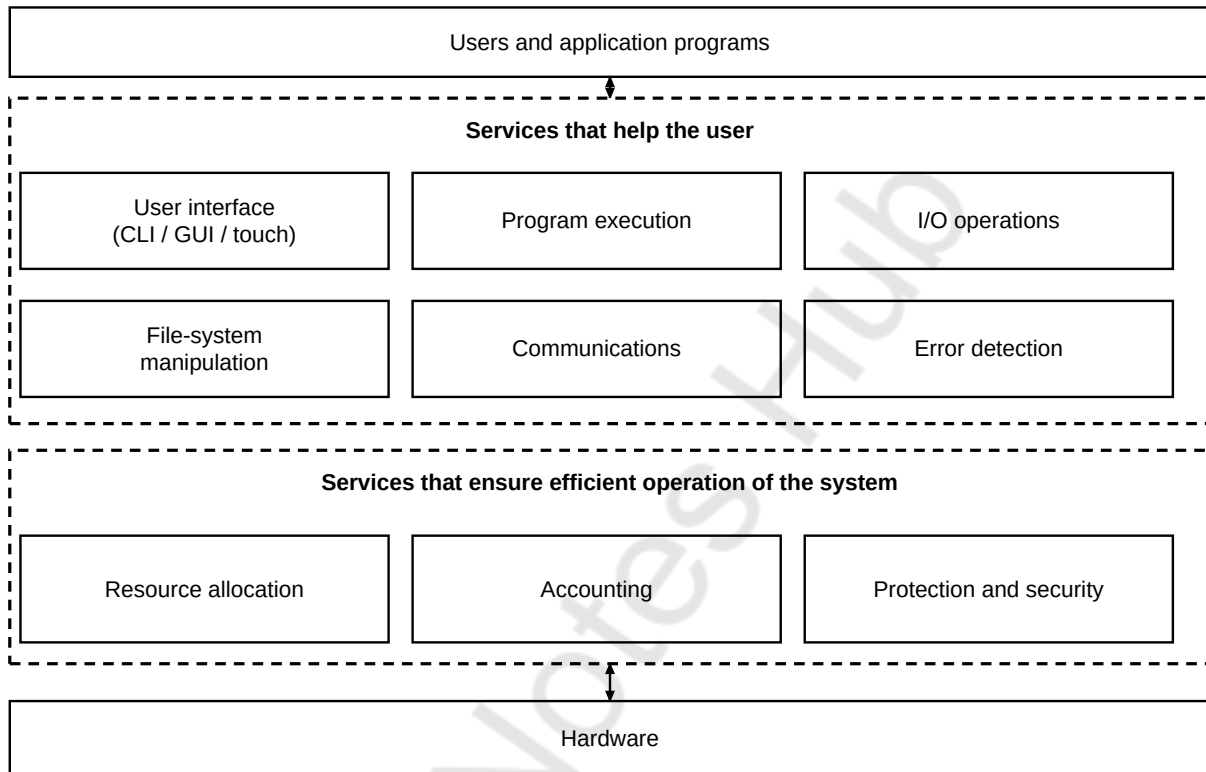


Fig 2.4: Operating system services

A. Services helpful to the user

Service	Description
User interface	Almost all systems have a user interface: a command-line interface (CLI) using typed commands, a graphical user interface (GUI) using windows, icons and a pointing device, or a touch-screen or batch interface.
Program execution	The system must load a program into memory, run it and end its execution either normally or abnormally (indicating an error).
I/O operations	A running program may need I/O from a file or device. For efficiency and protection, user programs cannot control I/O devices directly, so the OS provides the means to do I/O.
File-system manipulation	Programs need to read and write files and directories, create and delete them, search for them, list file information and manage permissions.
Communications	One process may need to exchange information with another, on the same computer or over a network. It is implemented through shared memory or message passing.

Service	Description
Error detection	The OS must constantly detect and correct errors in the CPU and memory hardware (memory error, power failure), in I/O devices (parity error, network failure, printer out of paper) and in user programs (arithmetic overflow, illegal memory access, too much CPU time).

B. Services for efficient operation of the system

Service	Description
Resource allocation	When several users or jobs run at the same time, resources (CPU cycles, main memory, file storage, I/O devices) must be allocated to each of them. The OS uses CPU-scheduling routines and allocation routines to do this fairly and efficiently.
Accounting	Keeping track of which users use how much and what kinds of computer resources. This record is used for billing or for gathering usage statistics.
Protection and security	The owners of information in a multi-user or networked system want to control its use. Protection ensures that all access to system resources is controlled. Security requires users to authenticate themselves and defends external I/O devices from invalid access attempts.

Basis	Protection	Security
Meaning	Mechanism to control access of processes and users to resources	Defence of the system against internal and external attacks
Concerned with	Internal threats: one process interfering with another	External threats: viruses, unauthorised users, denial of service
Example	File permissions, memory limit registers	Passwords, authentication, encryption

Quick revision: 2.2

- User-oriented services: user interface, program execution, I/O operations, file-system manipulation, communications, error detection.
- System-oriented services: resource allocation, accounting, protection and security.
- Services are made available to programs through system calls (next section).

2.3 System Calls

2.3.1 Introduction and Uses

A **system call** is the programmatic way in which a user program requests a service from the kernel of the operating system. System calls provide the interface between a running process and the operating system. They are generally available as functions written in C or C++, although some low-level tasks (for example, direct hardware access) may need assembly language. A user program never touches the hardware directly; it asks the kernel, which runs in privileged mode, to do the job on its behalf. This protects the system from faulty or malicious programs.

Uses of system calls:

- Creating, running and terminating processes.
- Creating, opening, reading, writing and deleting files and directories.
- Requesting and releasing I/O devices and performing I/O.

- Allocating and freeing memory.
- Getting and setting system information such as date, time and process attributes.
- Communicating between processes, on the same machine or across a network.
- Setting and checking protection (permissions) on files and resources.

How a system call works

1. The program calls a library function (for example `printf()`), which in turn calls the system call (`write()`).
2. The library places the **system-call number** and the parameters in registers (or memory or stack).
3. It executes a **trap** (software interrupt) instruction. The CPU switches from user mode to kernel mode (the mode bit changes) and saves the state of the program.
4. The kernel uses the system-call number as an index into the **system-call table** and finds the service routine.
5. The service routine runs in kernel mode and performs the requested work (for example, talking to the disk driver).
6. The result (return value or error code) is stored, the mode bit is reset to user mode and control returns to the instruction after the call.

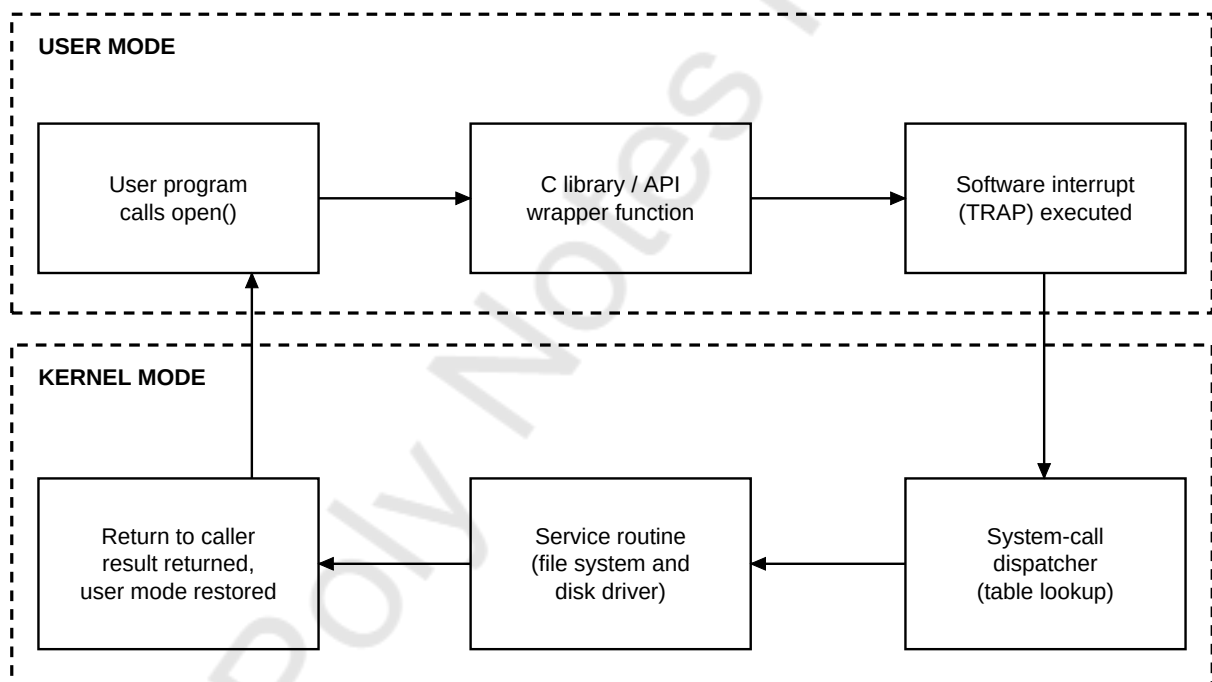


Fig 2.5: Execution of a system call (example: `open()`)

System call vs API

Application programmers normally do not call system calls directly. They use an **Application Programming Interface (API)**, a set of functions with defined parameters and return values. The three most common APIs are the Win32 API for Windows, the POSIX API for UNIX, Linux and macOS, and the Java API for the Java virtual machine. The run-time support system of the language (the library of functions in the compiler) provides a system-call interface that links each API function to the matching system call inside the kernel.

Basis	API	System call
Level	Higher-level function library	Entry point into the kernel

Basis	API	System call
Portability	Portable across systems that support the same API	Specific to the operating system
Mode	Runs in user mode (until it invokes a system call)	Executes in kernel mode
Convenience	Easier to use; hides details	Low-level; programmer must know the call number and parameters
Example	printf(), CreateProcess()	write(), NtCreateProcess()

Passing parameters to the operating system

A system call usually needs more information than just its identity (for example, which file and where to put the data). Three general methods are used to pass parameters to the OS:

Method	How it works	Remark
Registers	Parameters are placed directly in CPU registers.	Simplest and fastest, but there may be more parameters than registers.
Block (table) in memory	Parameters are stored in a block or table in memory, and the address of the block is passed in a register.	Used by Linux and Solaris; no limit on the number or length of parameters.
Stack	Parameters are pushed onto the stack by the program and popped off by the operating system.	Also has no limit on the number or length of parameters.

Example: copying one file to another

Even a simple program such as "copy file A to file B" uses a long series of system calls:

1. Write a prompt on the screen and read the name of the input file (write and read calls).
2. Prompt for and read the name of the output file.
3. Open the input file; abort with an error message if it does not exist.
4. Create the output file; abort if it already exists (or ask whether to overwrite).
5. Loop: read a block from the input file and write it to the output file, until the read reports end of file.
6. Close the output file and the input file.
7. Write a completion message on the screen and terminate normally (exit system call).

2.3.2 Types of System Calls

System calls can be grouped roughly into five major categories: process control, file management, device management, information maintenance and communication. A sixth group, protection, is also found in many systems.

(a) Process control

These calls create and manage processes. A running program must be able to end its execution normally (**end**) or abnormally (**abort**). A process can create a new process, load and execute another program, wait for it to finish and read or change its attributes.

- End, abort.
- Load, execute.
- Create process, terminate process.

- Get process attributes, set process attributes.
- Wait for time; wait for event, signal event.
- Allocate and free memory.

In UNIX, `fork()` creates a child process that is a copy of the parent; the child normally calls `exec()` to replace its memory image with a new program, while the parent calls `wait()` until the child finishes. The child ends with `exit()`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();           /* create a child process */
    if (pid < 0) {
        perror("fork failed");   /* error */
    } else if (pid == 0) {
        execlp("/bin/ls", "ls", NULL); /* child: load a new program */
    } else {
        wait(NULL);              /* parent: wait for the child */
        printf("Child complete\n");
    }
    return 0;
}
```

(b) File management

- Create file, delete file.
- Open, close.
- Read, write, reposition (seek).
- Get file attributes, set file attributes.
- Create directory, delete directory (in many systems).

The OS must know the name and attributes of each file. A program must open a file before using it; the OS returns a file descriptor (handle) that is used by the later read, write and close calls.

(c) Device management

- Request device, release device.
- Read, write, reposition.
- Get device attributes, set device attributes.
- Logically attach or detach devices.

A process may need several resources (main memory, disk drives, access to files) to execute. If they are available they are granted, otherwise the process must wait. Files and devices are so similar that many systems, such as UNIX, merge them into a combined **file-device structure** in which devices are accessed as special files (for example, `/dev/sda`).

(d) Information maintenance

- Get time or date, set time or date.
- Get system data (OS version, amount of free memory), set system data.
- Get process, file or device attributes; set process, file or device attributes.
- Memory dump and profiling for debugging.

Many calls exist only to transfer information between the user program and the operating system, for example `getpid()` and `time()`.

(e) Communication

- Create and delete a communication connection.
- Send and receive messages.
- Transfer status information.
- Attach or detach remote devices.

Two common models of inter-process communication exist:

Basis	Message-passing model	Shared-memory model
Data exchange	Processes exchange messages through the kernel	Processes read and write a shared region of memory
Kernel involvement	Kernel is involved in every send and receive	Kernel is needed only to set up the shared region
Speed	Slower because of system calls and copying	Faster, runs at memory speed
Synchronisation	Handled by the message system	Programmer must synchronise access
Best for	Small amounts of data; processes on different computers	Large amounts of data on the same computer

(f) Protection

Calls that control access to resources: set permission, get permission, allow user and deny user. They are used by every multi-user system, for example `chmod()` in UNIX.

Summary: equivalent calls in Windows and UNIX

Category	Windows	UNIX
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System programs (system utilities) sit between the user interface and the system calls. They provide a convenient environment for program development and execution: file manipulation (copy, delete, list), status information, file modification (editors), programming-language support (compilers), program loading and execution, communications and background services. Most users see the operating system through these programs rather than through the system calls themselves.

Quick revision: 2.3

- System call = interface between a process and the OS; executed through a trap that switches to kernel mode.
- Programmers use an API (Win32, POSIX, Java) which calls the actual system calls.
- Parameters are passed through registers, a block in memory or the stack.
- Categories: process control, file management, device management, information maintenance, communication (and protection).
- `fork()` creates, `exec()` replaces the program image, `wait()` waits, `exit()` terminates.

2.4 Operating System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. The **design goals** are of two kinds: *user goals* (the system should be convenient, easy to learn, reliable, safe and fast) and *system goals* (the system should be easy to design, implement and maintain, and also flexible, reliable, error-free and efficient). A key principle in design is to separate **policy** (what will be done) from **mechanism** (how it will be done), because policies may change over time and the separation gives flexibility. The commonly studied structures are described below.

2.4.1 Simple Structure

Many early operating systems did not have a well-defined structure. They started as small, simple and limited systems and then grew beyond their original scope. **MS-DOS** is the best example. It was written to provide the most functionality in the least space, so it was not divided into modules. Although it has some structure, its interfaces and levels of functionality are not well separated: application programs are able to access the basic I/O routines and write directly to the display and disk drives. This leaves MS-DOS vulnerable to errant (or malicious) programs, and the entire system can crash when a user program fails. MS-DOS was also limited by the hardware of its time: the Intel 8088 had no dual mode and no hardware protection. The original UNIX was similarly limited, with only two separable parts, the kernel and the system programs.

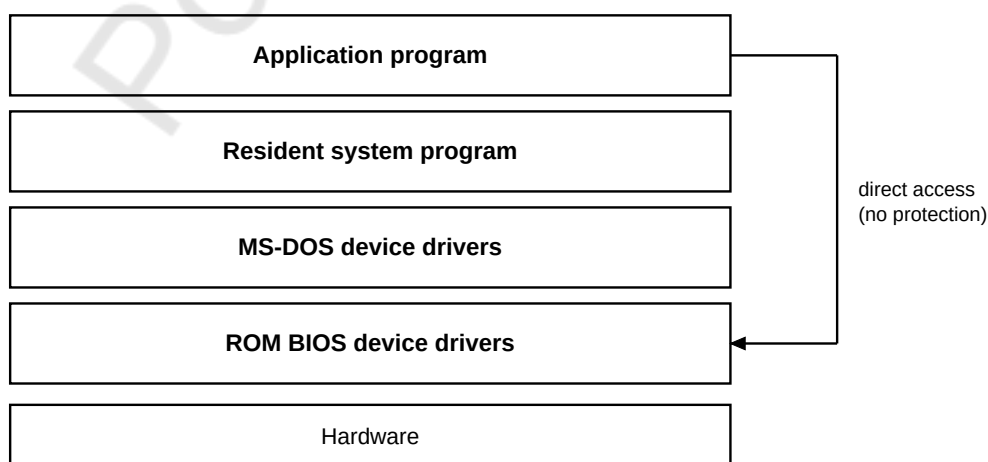


Fig 2.6: MS-DOS layer structure (simple structure)

Advantages	Disadvantages
Small, simple and fast; easy to design and implement	No clear modules; hard to understand, maintain and extend
Very little overhead because programs can reach the hardware directly	No protection between user programs and the OS; one faulty program can crash everything
Suitable for small, single-user systems with limited hardware	Difficult to add new features; not suitable for large systems

2.4.2 Monolithic Structure

In a **monolithic** operating system the entire OS works in kernel space as one large program. All services (process management, memory management, file system, device drivers, networking and so on) are compiled into a single binary and run in kernel mode in the same address space. The procedures inside the kernel can call each other freely, so communication between the parts is just a function call. The traditional **UNIX** kernel is the classic example: everything below the system-call interface and above the physical hardware is one huge kernel that provides the file system, CPU scheduling, memory management and other operating-system functions through system calls. Linux is also monolithic, but it uses **loadable kernel modules** so that drivers and file systems can be added or removed while the system is running.

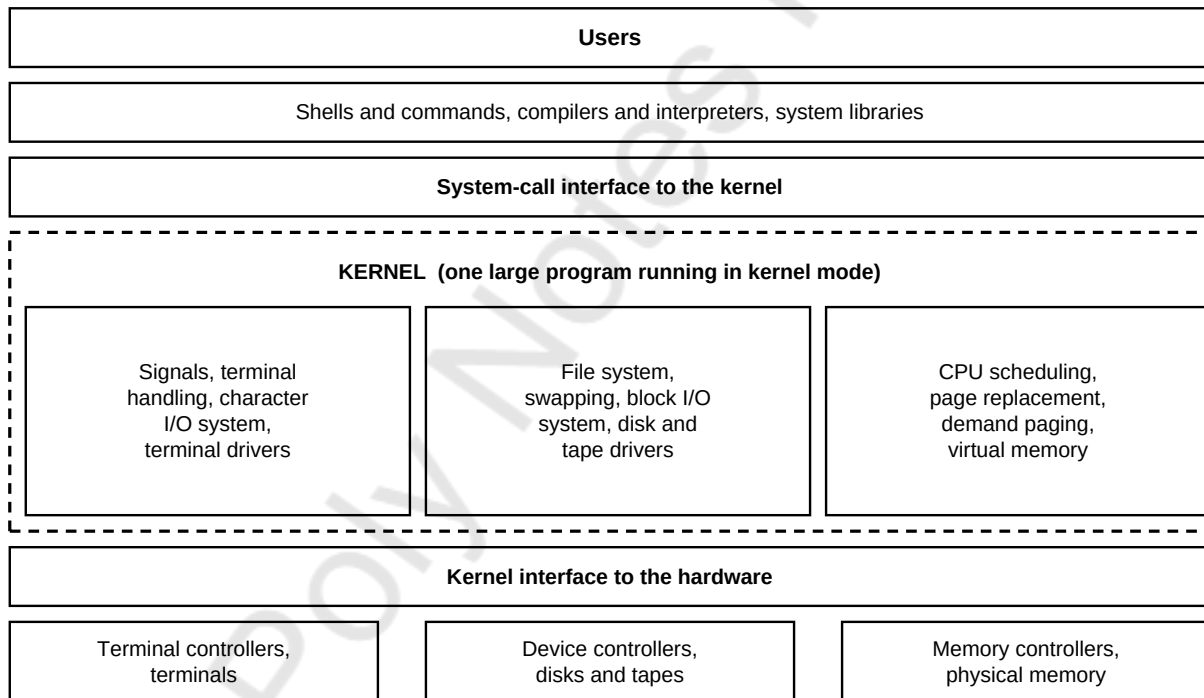


Fig 2.7: Traditional UNIX system structure (monolithic kernel)

Advantages	Disadvantages
High performance: services call each other directly, with very little overhead	The kernel is very large and hard to maintain, debug and extend
Efficient use of CPU and memory	A bug in any part (for example a device driver) can crash the whole system
Simple design compared with layered or microkernel systems at the implementation level	Poor isolation; less secure because all code runs with full privilege

2.4.3 Layered Structure

In the **layered approach** the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware and the highest layer (layer N) is the user interface. Each layer is built on top of the layers below it and uses the functions (operations) and services of **only lower-level layers**. A layer does not need to know how the operations of the lower layers are implemented, only what they do; so each layer hides the existence of certain data structures, operations and hardware from higher layers. The first layered system was the **THE system** designed by E. W. Dijkstra and his students (1968).

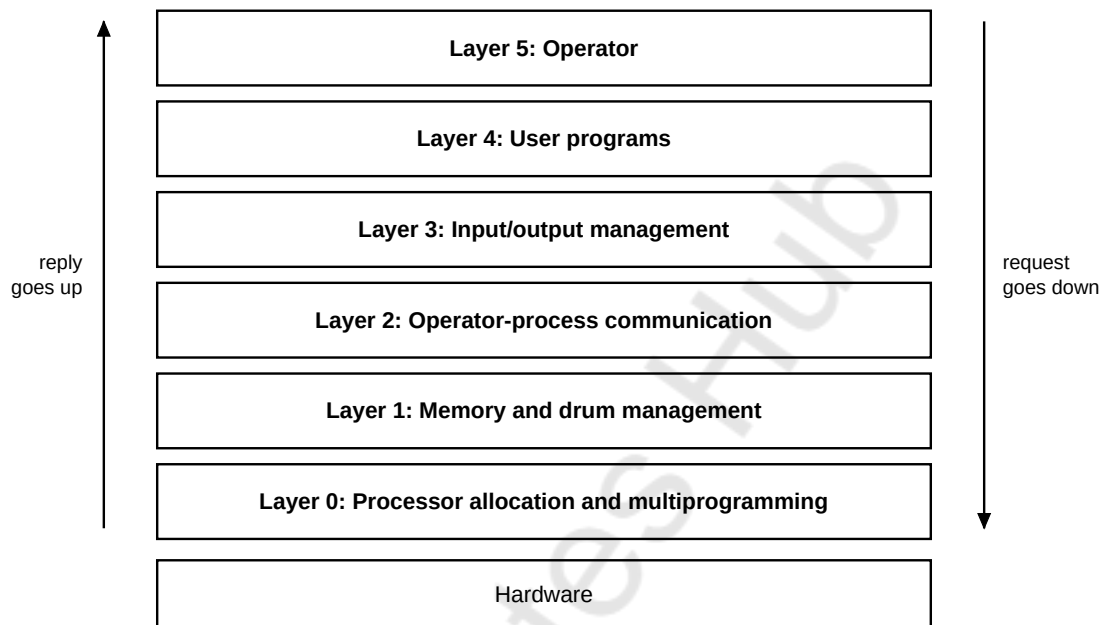


Fig 2.8: Layered structure (the THE operating system)

Advantages	Disadvantages
Simplicity of construction and debugging: layer 0 is tested first, then layer 1 and so on	Careful planning is needed because a layer can use only lower layers (for example, the backing-store driver must be below memory management)
Modularity and information hiding: a layer can be re-implemented without affecting others	Less efficient: a request passes through many layers, each adding overhead (parameters, data copying)
Easy to verify and maintain	Many functions cannot be ordered strictly, so real systems use only a few layers

2.4.4 Microkernel Structure

As UNIX expanded, its kernel became large and difficult to manage. In the mid-1980s researchers at Carnegie Mellon University developed an operating system called **Mach** that modularised the kernel using the **microkernel** approach. This method removes all non-essential components from the kernel and implements them as system and user-level programs (servers) running in user space. The result is a smaller kernel. The microkernel provides only minimal process and memory management and a communication facility. The main function of the microkernel is to provide communication between the client program and the various services (file server, device drivers, network server) running in user space; communication is done by **message passing** through the microkernel.

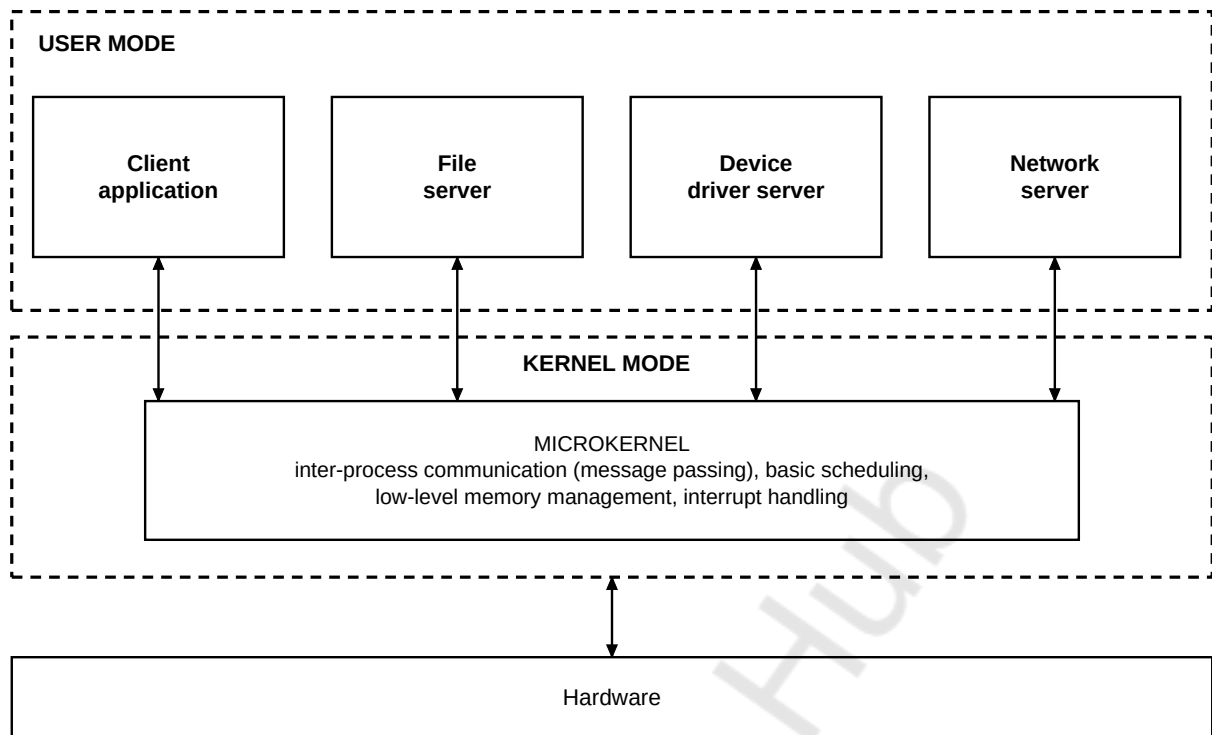


Fig 2.9: Microkernel architecture (services run in user mode)

Advantages	Disadvantages
Easy to extend: new services are added in user space without modifying the kernel	Performance overhead of user-space to kernel-space communication (message copying, extra context switches)
Easier to port the OS to new hardware, since the kernel is small	Many services need several messages to complete, so the system can be slower than a monolithic kernel
More reliable and secure: less code runs in kernel mode and a failed server does not crash the kernel	Design of the message-passing interface is complex

Examples: Mach, QNX, MINIX, L4. Windows NT originally used a layered microkernel organisation, but its performance was lower than that of Windows 95; Windows NT 4.0 moved some layers from user space into kernel space and later versions became closer to monolithic. macOS (Darwin kernel XNU) combines the Mach microkernel with a BSD kernel and is therefore called a **hybrid** system.

Comparison of the four structures

Feature	Simple	Monolithic	Layered	Microkernel
Organisation	No clear structure	One large kernel program	Hierarchy of layers	Small kernel plus user-space servers
Speed	Fast	Fast	Slower (layer overhead)	Slower (message passing)
Protection / reliability	Very poor	Moderate (one bug can crash all)	Good	Very good
Maintenance and extension	Difficult	Difficult	Easy	Easiest
Code in kernel mode	All	All services	Most layers	Minimum

Feature	Simple	Monolithic	Layered	Microkernel
Examples	MS-DOS	UNIX, Linux	THE system	Mach, QNX, MINIX

Quick revision: 2.4

- Simple structure (MS-DOS): no modules, no protection, fast and small.
- Monolithic (UNIX, Linux): all OS services in one kernel program running in kernel mode; fast but hard to maintain.
- Layered: layer 0 = hardware, layer N = user interface; each layer uses only lower layers; easy to debug but slower.
- Microkernel (Mach): minimal kernel (IPC, scheduling, memory), other services in user space; reliable and extensible but slower.

2.5 Booting

Bootting is the process of starting a computer and loading the operating system kernel into main memory so that the system becomes ready for use. The word comes from *bootstrapping*, the idea of "pulling oneself up by one's own bootstraps". When the power is switched on, main memory (RAM) is empty and no operating system is present in it. The CPU therefore needs a small program stored in **non-volatile** memory (ROM, EEPROM or flash), called the **bootstrap program** or firmware, which tells it how to find the operating system and load it.

Because it is inconvenient to change firmware stored in ROM, most systems keep only a tiny bootstrap loader in firmware. This loader brings a larger boot program from a fixed place on disk, called the **boot block**. A disk that contains a boot partition is called a **boot disk** or **system disk**.

Steps in the booting process

1. **Power on / reset:** the CPU starts executing at a fixed address (the reset vector) where the firmware is stored.
2. **Firmware starts:** BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface) takes control.
3. **POST (Power-On Self-Test):** firmware checks the CPU, RAM, keyboard, display adapter and storage controllers. If a fault is found, an error message or a beep code is produced.
4. **Find the boot device:** the firmware initialises hardware and searches the devices in the configured boot order (hard disk or SSD, USB drive, optical disc, network).
5. **Load the boot loader:** with BIOS, the first sector (512 bytes, the MBR) of the boot device is read into memory and executed; with UEFI, the firmware loads an .efi boot-loader file from the EFI System Partition.
6. **Boot loader loads the kernel:** programs such as GRUB (Linux) or Windows Boot Manager show a menu if several operating systems are present, then load the selected kernel (and an initial RAM disk) into main memory.
7. **Kernel initialisation:** the kernel sets up memory management, interrupts and device drivers, and mounts the root file system.
8. **First user process:** the kernel starts the first process (init or systemd on Linux, the session manager on Windows), which starts system services and finally the login prompt or the graphical desktop.

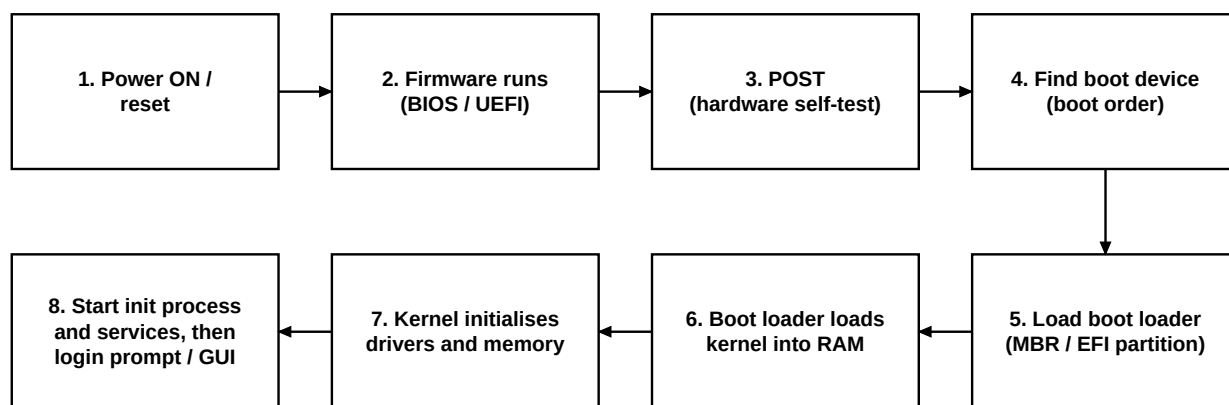


Fig 2.10: Booting sequence

Master Boot Record (MBR)

In the BIOS and MBR scheme, the very first sector of the disk contains the **Master Boot Record**. It holds the first stage of the bootstrap loader and the partition table, which describes how the disk is divided into partitions and which partition is active (bootable). The last two bytes contain the boot signature 55AA.

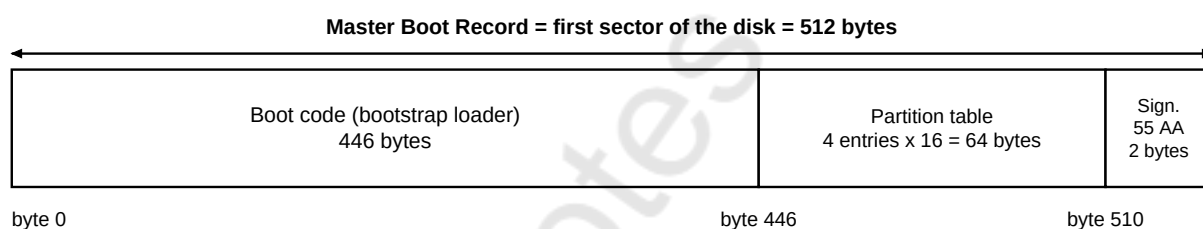


Fig 2.11: Layout of the Master Boot Record

Types of booting

Basis	Cold (hard) boot	Warm (soft) boot
Meaning	Starting the computer from the powered-off state	Restarting the computer while it is already powered on
How	Pressing the power button	Restart command or Ctrl+Alt+Del
POST	Performed in full	Often skipped or shortened
Memory	RAM starts empty	RAM is cleared and reloaded by the restart
Use	Normal start-up	After installing software or recovering from a hang

BIOS and UEFI

Basis	BIOS	UEFI
Full form	Basic Input/Output System	Unified Extensible Firmware Interface
Processor mode	16-bit real mode	32-bit or 64-bit mode
Partition scheme	MBR: disks up to 2 TB, 4 primary partitions	GPT: disks larger than 2 TB, many partitions

Basis	BIOS	UEFI
Boot speed	Slower	Faster
Interface	Text-based, keyboard only	Graphical, mouse support possible
Security	No Secure Boot	Supports Secure Boot (only signed boot loaders run)
Boot code location	First sector of the disk (MBR)	Files on the EFI System Partition

Linux and Windows boot at a glance

Stage	Linux	Windows
Firmware	BIOS or UEFI	BIOS or UEFI
Boot loader	GRUB2	Windows Boot Manager, then winload
Kernel	vmlinux with initramfs	ntoskrnl.exe
First process / services	init or systemd (PID 1)	Session manager (smss.exe), then services

- **Dual boot:** two operating systems are installed and the boot loader lets the user choose one at start-up.
- **Network boot (PXE):** the computer loads its boot image over the network, used in labs and data centres.

Quick revision: 2.5

- Booting = loading the OS kernel into memory at start-up using the bootstrap program stored in ROM/firmware.
- Order: power on, firmware (BIOS/UEFI), POST, boot device, boot loader, kernel, init and services.
- MBR = first 512-byte sector: 446 bytes boot code + 64 bytes partition table + 2 bytes signature.
- Cold boot starts from power off; warm boot restarts without switching off power.

2.6 Virtual Machine

A **virtual machine (VM)** abstracts the hardware of a single computer (CPU, memory, disk drives, network interface) into several different execution environments, creating the illusion that each environment is running on its own private computer. It is the layered approach taken to its logical conclusion: the hardware and the operating system kernel are treated as if they were all hardware, and the virtual-machine software presents an interface identical to the underlying bare hardware. The physical machine is the **host**; each operating system running inside a VM is a **guest**. The software that creates and runs virtual machines is called the **virtual machine monitor (VMM)** or **hypervisor**. The idea was first commercialised by IBM in VM/370 (1972).

Benefits

- **Isolation and protection:** each VM is completely isolated from the others, so a crash or virus in one VM does not affect the rest.
- **Hardware sharing and server consolidation:** many lightly loaded servers can be moved onto one physical machine, saving cost and power.
- **Multiple operating systems:** Windows, Linux and other systems can run at the same time on one computer.
- **Safe testing and OS development:** system development can be done on a VM without disturbing the normal working of the machine.

- **Snapshots, cloning and migration:** the state of a VM can be saved, copied or moved to another host, even while running.
- **Foundation of cloud computing:** cloud providers rent VMs to customers.

Drawbacks

- Providing an exact duplicate of the underlying machine is difficult and complex.
- Performance overhead, since the VMM sits between the guest and the hardware.
- Direct sharing of resources between VMs is difficult; sharing is done through virtual networks or shared folders.

Structure and implementation

Fig 2.12 compares an ordinary computer with one running several virtual machines. In (a) the processes run on a single kernel; in (b) the virtual machine implementation sits directly above the hardware and each VM has its own guest operating system and processes, with its own virtual CPU, memory and disks.

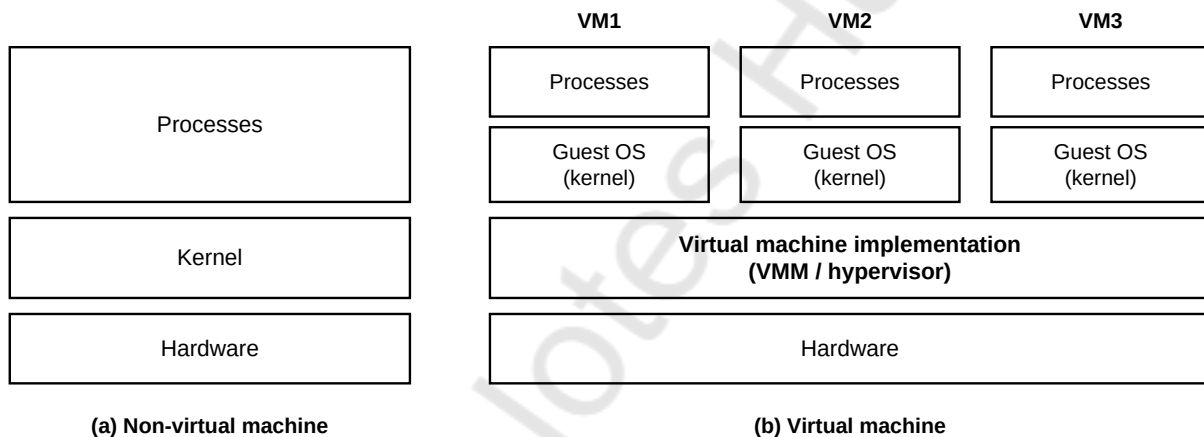


Fig 2.12: (a) Non-virtual machine and (b) virtual machine

Virtual user mode and virtual kernel mode

The VMM runs in physical kernel mode, but the guest operating system runs in physical **user mode**. The guest still believes it has a user mode and a kernel mode, which are called **virtual user mode** and **virtual kernel mode**. When the guest kernel executes a privileged instruction, the CPU traps to the VMM, which emulates the instruction on behalf of the guest and returns control. Modern processors add hardware support for this (Intel VT-x, AMD-V), which makes virtualisation much faster.

Types of hypervisors

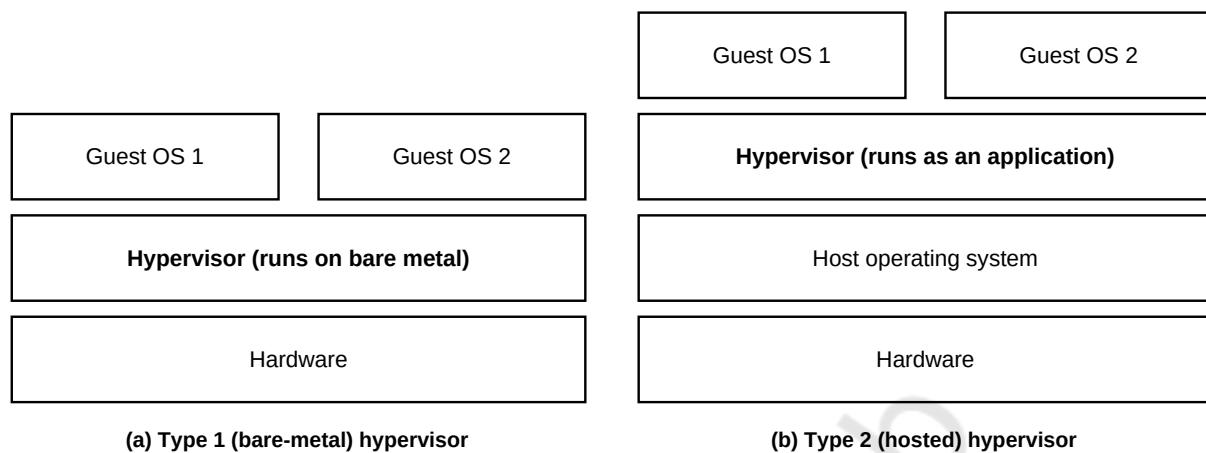


Fig 2.13: Type 1 and Type 2 hypervisors

Basis	Type 1 (bare-metal)	Type 2 (hosted)
Runs on	Directly on the hardware	As an application on a host operating system
Performance	Higher, less overhead	Lower, because of the extra host OS layer
Typical use	Data centres and servers	Desktops, learning and testing
Examples	VMware ESXi, Xen, Microsoft Hyper-V, KVM	Oracle VirtualBox, VMware Workstation, Parallels

Virtualisation techniques

Technique	Description
Full virtualisation	The guest OS runs unmodified; the VMM traps and emulates privileged operations (using binary translation or hardware support).
Paravirtualisation	The guest OS is modified to make calls (hypercalls) to the hypervisor instead of privileged instructions, giving better speed (for example, Xen).
Hardware-assisted virtualisation	The CPU provides special modes (Intel VT-x, AMD-V) that let the hypervisor run guests efficiently.
Emulation	The whole CPU of a different architecture is simulated in software (for example, QEMU). Slowest method.
OS-level virtualisation (containers)	Isolated user-space instances share the host kernel. They are lightweight (for example, Docker).
Programming-environment VM	A virtual machine designed for a language: the Java Virtual Machine runs the same bytecode on any computer ("write once, run anywhere").

Quick revision: 2.6

- Virtual machine = software illusion of several separate computers on one physical machine.
- Host = physical machine; guest = OS inside a VM; hypervisor/VMM creates and manages VMs.
- Type 1 hypervisor runs on hardware; Type 2 runs on a host OS.
- Guest kernel runs in physical user mode; privileged instructions trap to the VMM.
- Benefits: isolation, consolidation, testing, multiple OSes, cloud computing.

Chapter Summary

- The operating system is divided into components: process, main memory, file, I/O system and secondary storage management.
- It offers services to the user (UI, program execution, I/O, files, communication, error detection) and to the system (resource allocation, accounting, protection and security).
- Programs request kernel services through system calls, usually via an API; the call switches the CPU into kernel mode with a trap.
- System calls are of five main kinds: process control, file management, device management, information maintenance and communication.
- Structures: simple (MS-DOS), monolithic (UNIX, Linux), layered (THE) and microkernel (Mach).
- Booting loads the kernel using firmware, POST, boot loader and kernel initialisation.
- A virtual machine lets one physical computer run several isolated operating systems under a hypervisor.

Practice Questions

Part A: Multiple Choice Questions (15)

Choose the correct option. The answer key is given on the next page.

1. A program in execution is called a
 - (a) Process
 - (b) Thread
 - (c) Compiler
 - (d) Directory
2. Which of the following is NOT a responsibility of the process-management component of an operating system?
 - (a) Creating and deleting processes
 - (b) Suspending and resuming processes
 - (c) Providing mechanisms for process synchronisation
 - (d) Managing free space on the disk
3. Keeping track of which parts of main memory are in use and by whom is the function of
 - (a) File management
 - (b) Main memory management
 - (c) I/O management
 - (d) Secondary storage management
4. Holding the output for a slow device such as a printer on disk so that the CPU can continue with other work is called
 - (a) Buffering
 - (b) Caching
 - (c) Spooling
 - (d) Paging
5. Free-space management, storage allocation and disk scheduling are functions of
 - (a) Process management
 - (b) Main memory management
 - (c) Protection system
 - (d) Secondary storage management
6. Which of the following services mainly ensures the efficient operation of the system rather than directly helping the user?
 - (a) Program execution
 - (b) Accounting
 - (c) File-system manipulation
 - (d) Communications
7. Which mechanism is normally used to transfer control from a user program to the operating system kernel during a system call?
 - (a) Trap (software interrupt)
 - (b) Ordinary subroutine call
 - (c) Macro expansion
 - (d) Direct jump into kernel memory

8. In UNIX, the system call used to create a new process is
- (a) exec()
 - (b) fork()
 - (c) wait()
 - (d) pipe()
9. When there are more parameters than registers, parameters are generally passed to the operating system by
- (a) Writing them into the program counter
 - (b) Saving them in the cache only
 - (c) Storing them in a block in memory and passing the block address in a register
 - (d) Placing them in the disk boot sector
10. In which structure does the whole operating system run as one large program in kernel mode in a single address space?
- (a) Microkernel
 - (b) Monolithic
 - (c) Layered
 - (d) Virtual machine
11. MS-DOS is an example of which operating system structure?
- (a) Simple structure
 - (b) Layered structure
 - (c) Microkernel
 - (d) Virtual machine
12. The main disadvantage of the layered approach is
- (a) Lack of modularity
 - (b) Debugging is impossible
 - (c) Lower efficiency because a request passes through many layers
 - (d) It cannot use the hardware
13. In a microkernel-based operating system, services such as the file server and device drivers run
- (a) Inside the kernel binary in kernel mode
 - (b) In ROM firmware
 - (c) In the BIOS
 - (d) In user space as separate server processes
14. Which sequence correctly shows the start-up process of a computer?
- (a) Boot loader, POST, kernel, BIOS
 - (b) POST, BIOS, kernel, boot loader
 - (c) Firmware with POST, boot loader, kernel, init and services
 - (d) Kernel, boot loader, POST, shell
15. Which of the following is an example of a Type 2 (hosted) hypervisor?
- (a) Oracle VirtualBox
 - (b) VMware ESXi
 - (c) Xen
 - (d) IBM VM/370

Answer Key for Part A

Q. No.	Answer	Explanation
1	(a) Process	A process is a program in execution (an active entity).
2	(d) Managing free space on the disk	Free-space management belongs to secondary storage management.
3	(b) Main memory management	This is a main memory management task.
4	(c) Spooling	Spooling = Simultaneous Peripheral Operations On-Line.
5	(d) Secondary storage management	These are the three tasks of secondary storage management.
6	(b) Accounting	Accounting (with resource allocation and protection) is a system-efficiency service.
7	(a) Trap (software interrupt)	A trap switches the CPU from user mode to kernel mode.
8	(b) fork()	fork() creates a child process; exec() replaces its program.
9	(c) Storing them in a block in memory and passing the block address in a register	The block (table) method places no limit on the number of parameters.
10	(b) Monolithic	Traditional UNIX and Linux are monolithic.
11	(a) Simple structure	MS-DOS has no well-separated modules, so it is a simple structure.
12	(c) Lower efficiency because a request passes through many layers	Each layer adds overhead to a request.
13	(d) In user space as separate server processes	Only minimal functions remain in the microkernel.
14	(c) Firmware with POST, boot loader, kernel, init and services	Firmware and POST run first, then the boot loader loads the kernel, which starts init.
15	(a) Oracle VirtualBox	VirtualBox runs as an application on a host OS; the others are bare-metal.

Quick key: 1-a, 2-d, 3-b, 4-c, 5-d, 6-b, 7-a, 8-b, 9-c, 10-b, 11-a, 12-c, 13-d, 14-c, 15-a

Part B: Broad (Long Answer) Questions (10)

Answer in detail, with diagrams and tables wherever required.

1. List and explain the major components of an operating system. Describe process management and main memory management in detail, with the process state diagram.
2. Explain the role of the operating system in file management, I/O system management and secondary storage management. Differentiate between buffering, caching and spooling.
3. What services does an operating system provide? Explain the services helpful to the user and the services that ensure efficient operation of the system.
4. What is a system call? Explain how a system call is executed with a neat diagram. Describe the methods of passing parameters to the operating system and the difference between a system call and an API.
5. Describe the different types of system calls (process control, file management, device management, information maintenance and communication) with suitable Windows and UNIX examples.
6. Explain the simple structure and the monolithic structure of an operating system with diagrams. Write their advantages and disadvantages.

7. Explain the layered approach of operating system design with a neat diagram. Discuss its merits and demerits.
8. What is a microkernel? Explain its architecture, advantages and disadvantages, and compare it with a monolithic kernel.
9. What is booting? Explain the booting process step by step with a diagram. Differentiate between BIOS and UEFI and between cold boot and warm boot.
10. What is a virtual machine? Explain its benefits, implementation (virtual user mode and virtual kernel mode) and the types of hypervisors with diagrams.

Poly Notes Hub