

UNIT 2

Operating System Structure

(2 Hrs) — Operating System (CACS251)

2.1 Introduction

An operating system is a large and complex piece of software. Its **structure** describes how its components are organised and interconnected. A good design makes the OS easier to write, debug, maintain and extend. The main structural approaches are the simple (monolithic) structure, the layered approach, the microkernel, the client-server model and virtual machines.

2.2 Layered System

In the layered approach the OS is divided into a number of **layers**. The bottom layer (layer 0) is the hardware and the highest layer is the user interface. Each layer is built on top of the layer below and uses *only* the services of lower layers. This was first used in the THE operating system (Dijkstra, 1968).

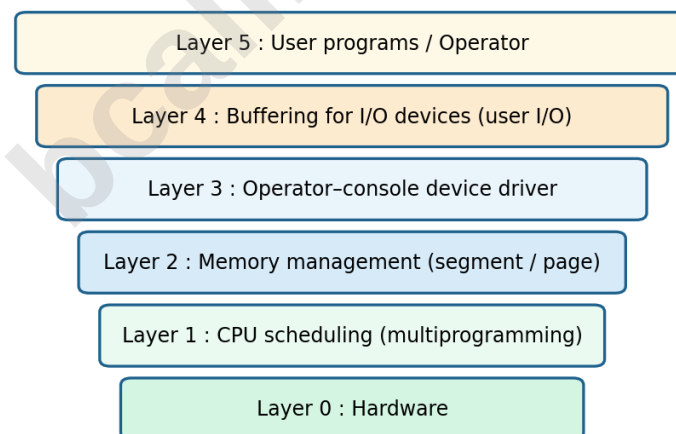


Figure 2.1 — Layered structure of an operating system (THE system)

Advantages	Disadvantages
Simplicity of construction and debugging — each layer tested independently	Careful definition of layers is difficult (a layer can only use lower layers)
Modularity and information hiding	Less efficient — a request passes through many layers, adding overhead
Easy to modify one layer without affecting others	Rigid ordering limits flexibility

2.3 Kernel

The **kernel** is the core part of the operating system. It is loaded first at boot time, always stays in main memory, runs in privileged (kernel) mode and has complete control over the hardware. Its responsibilities include process scheduling, memory management, device drivers, inter-process communication and system-call handling.

2.3.1 Monolithic (Macro) Kernel

In a **monolithic kernel** the entire operating system — scheduler, memory manager, file system, drivers, networking — runs as one large program in kernel space. All components can call each other directly with simple procedure calls, which makes it **fast**. However, a bug in any component (for instance a faulty driver) can crash the whole system, and the code is hard to maintain. Examples: traditional UNIX, Linux (modular monolithic), MS-DOS.

2.3.2 Microkernel

A **microkernel** keeps only the minimum in kernel mode — low-level address-space management, basic scheduling and inter-process communication (IPC). Everything else (file system, device drivers, networking) runs as ordinary **user-space server processes** that communicate with each other through message passing. Examples: Mach, QNX, MINIX 3, L4.

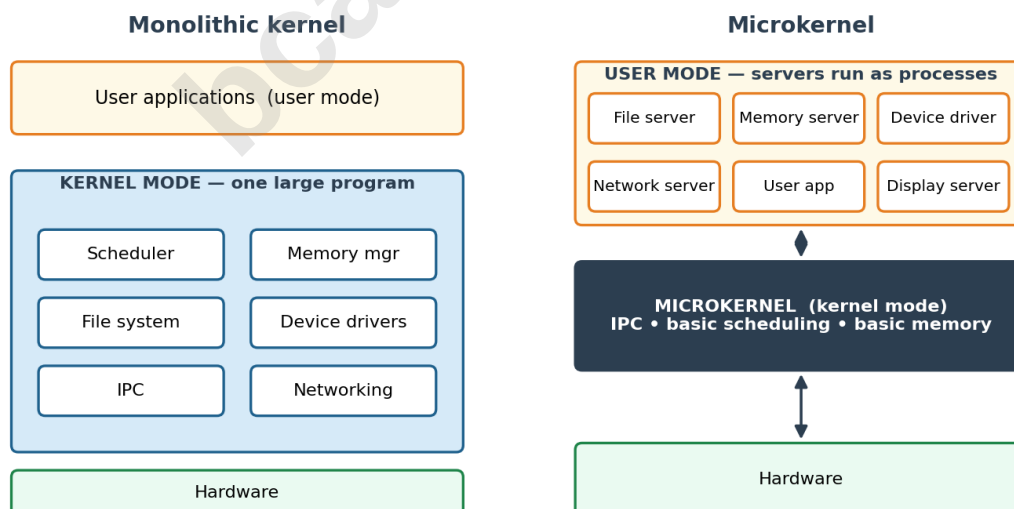


Figure 2.2 — Monolithic kernel versus microkernel

Feature	Monolithic kernel	Microkernel
Size	Large	Very small
Speed	Fast (direct function calls)	Slower (message-passing overhead)

Feature	Monolithic kernel	Microkernel
Reliability	One faulty module can crash the OS	A failed server can be restarted without crashing kernel
Security	Whole kernel is trusted code	Smaller trusted computing base
Extensibility	Difficult — kernel recompilation	Easy — add a new user-space server
Examples	UNIX, Linux, MS-DOS	Mach, QNX, MINIX 3

2.3.3 Exokernel

An **exokernel** (exo = “outside”) goes further than the microkernel. It does not provide abstractions such as files or processes at all. Instead it only **securely multiplexes the raw hardware resources** (e.g., allocating physical disk blocks and memory pages to applications). Application-level *library operating systems* then build whatever abstractions they need on top. This gives applications maximum performance and flexibility and removes a layer of mapping between virtual and physical resources. Example: MIT Aegis / Xok.

Note: Hybrid kernels (Windows NT, macOS XNU) combine ideas from both monolithic and microkernel designs.

2.4 Client–Server Model

A variation of the microkernel idea is the **client–server model**. The operating system’s services are implemented by **server processes** (file server, memory server, process server) running in user mode. A **client** process — a user program or another server — requests a service by sending a message to the appropriate server and waits for the reply. The kernel’s only job is to handle communication between clients and servers.

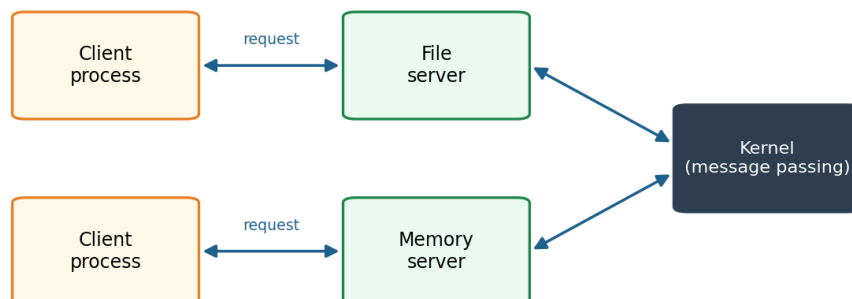


Figure 2.3 — Client–server model within an operating system

- Servers run in user mode, so they cannot directly access the hardware — a bug in a server does not crash the whole system.

- The model adapts naturally to **distributed systems**: a client neither knows nor cares whether the request is served locally or on a remote machine.
- Drawback: message passing is slower than direct procedure calls.

2.5 Virtual Machines

A **virtual machine (VM)** takes the layered approach to its logical conclusion. A **virtual machine monitor (hypervisor)** runs directly on the bare hardware and creates several *virtual copies* of the hardware. Each virtual machine can run its own guest operating system, which believes it has a whole physical computer to itself. The first widely used VM system was IBM VM/370.

Type	Description	Examples
Type 1 (bare-metal) hypervisor	Runs directly on the hardware; guest OSes run above it	Xen, VMware ESXi, Microsoft Hyper-V
Type 2 (hosted) hypervisor	Runs as an application on top of a host OS	VirtualBox, VMware Workstation
Language VM	Executes portable byte-code on a runtime	Java Virtual Machine (JVM)

Benefits: complete isolation and protection between VMs, ability to run several different operating systems on one computer, safe testing and OS development, server consolidation and easy backup/migration. **Drawback:** some performance overhead and sharing of files between VMs is harder.

2.6 Shell

The **shell** is the **command interpreter** — the program that lets the user interact with the operating system. It is *not* part of the kernel, but it uses many kernel features. The shell reads commands typed by the user, interprets them and asks the kernel to execute them through system calls.

- **Command-line shell (CLI):** text commands, e.g., bash, sh, csh, PowerShell, cmd.exe.
- **Graphical shell (GUI):** windows, icons and menus, e.g., Windows Explorer, GNOME, KDE.

Example — Steps in a shell loop

1. Display the prompt (\$ or C:\>).
2. Read the command line typed by the user.
3. Parse it into a program name and arguments.
4. Create a child process (fork) and load the program (exec).
5. Wait for it to finish (or run it in the background), then go back to step 1.

Kernel	Shell
Core of the OS, always in memory	Outer layer / user interface
Runs in kernel (privileged) mode	Runs in user mode
Interacts directly with hardware	Interacts with the kernel through system calls
Only one per system	Many different shells can exist

Key Takeaways

- ✓ OS structures: monolithic, layered, microkernel, client–server and virtual machine.
- ✓ The **layered** approach gives modularity but adds overhead; each layer uses only lower layers.
- ✓ **Monolithic kernel** is fast but less reliable; **microkernel** is small, reliable, but slower due to IPC; **exokernel** only multiplexes raw hardware.
- ✓ **Client–server** model moves OS services into user-mode servers communicating by messages.
- ✓ A **virtual machine** (via a hypervisor) lets several operating systems share one physical machine.
- ✓ The **shell** is the command interpreter — the user’s interface to the kernel.

— End of Unit 2 —