Introduction to Operating Systems

Advanced Operating Systems — Week 1

Hunter College, CUNY

Faye
czhang2@gradcenter.cuny.edu

January 2026

Contents

1 What is an Operating System? 2

2 Core Functions 2
2.1 Resource Manager . . . 2
2.2 Abstraction Provider . . . 2

3 OS History and Evolution 3
3.1 Batch Processing (1950s) . . . 3
3.2 Multiprogramming (1960s) . . . 3
3.3 Time-Sharing and UNIX (1970s) . . . 3
3.4 Personal Computing (1980s–1990s) . . . 4
3.5 Modern OS Landscape (2000s–present) . . . 4

4 Kernel Architecture 4
4.1 Monolithic Kernel . . . 5
4.2 Microkernel . . . 5
4.3 Hybrid Kernel . . . 5
4.4 Exokernel . . . 5

5 User Mode vs Kernel Mode 5
5.1 Kernel Mode . . . 6
5.2 Mode Switching . . . 6
5.3 Cost of Mode Switching . . . 6

6 System Calls 7
6.1 Categories of System Calls . . . 7
6.2 Example: The `write()` System Call . . . 7
6.3 Tracing System Calls with `strace` . . . 7

7 Key Takeaways 8

1 What is an Operating System?

An **operating system (OS)** is a layer of software that manages computer hardware resources and provides an interface between the user/application programs and the hardware. It acts as a *government*: it makes the system convenient to use while ensuring fair and efficient use of resources.

Three Fundamental Concepts

1. **Virtualization** — The OS takes a physical resource (CPU, memory, disk) and transforms it into a more general, powerful, and easy-to-use *virtual* form. Each process believes it has its own CPU and its own private address space.
2. **Concurrency** — The OS must correctly handle events that occur simultaneously. Threads, locks, semaphores, and condition variables are the key abstractions for managing concurrent execution.
3. **Persistence** — Data must survive power failures and system crashes. The OS provides a file system that stores data reliably on disk, handling writes carefully through journaling or copy-on-write techniques.

> *"The OS is the one program running at all times on the computer — usually called the **kernel**. Everything else is either a system program or an application program."*
> *— Silberschatz, Galvin, Gagne*

2 Core Functions

2.1 Resource Manager

The OS arbitrates access to shared hardware resources:

- **CPU scheduling** — decides which process runs and for how long.
- **Memory management** — allocates/deallocates memory, implements virtual memory.
- **I/O management** — mediates access to disks, network, and peripherals.
- **Protection and security** — isolates processes from one another and from the kernel.

2.2 Abstraction Provider

The OS hides messy hardware details behind clean abstractions.

Hardware Reality	OS Abstraction
CPU (one or more cores)	Process / Thread
RAM (physical DRAM chips)	Virtual address space
Disk (sectors, platters)	Files and directories
Network (packets, NIC)	Sockets
Interrupts (IRQs)	Device-independent I/O

3 OS History and Evolution

Era	Key Development
1940s	No OS — programs ran directly on hardware (vacuum tubes, plugboards).
1950s	Batch processing systems; jobs submitted on punch cards.
1960s	Multiprogramming; multiple jobs in memory simultaneously (IBM OS/360).
1970s	Time-sharing (UNIX 1969, Multics); interactive terminals.
1980s	Personal computing; DOS, early Windows, Macintosh.
1990s	Modern GUI-based OS; Windows NT, Linux (1991), network-centric design.
2000s	Multi-core, 64-bit, virtualization (VMware, Xen).
2010s	Mobile OS (Android, iOS), containers (Docker), cloud-native OS.

3.1 Batch Processing (1950s)

In batch processing, the operator collected jobs (card decks), grouped similar ones, and fed them to the computer sequentially. A **resident monitor** automatically loaded and ran the next job.

Key Innovation

Automatic job sequencing eliminated idle time between jobs. The monitor used a **job control language (JCL)** to specify resource requirements.

Limitations:

- No interaction — users submitted jobs and waited hours or days.
- CPU idle during I/O — only one job in memory at a time.
- Debugging was painful; turnaround time was very long.

3.2 Multiprogramming (1960s)

Multiple jobs are kept in memory simultaneously. When one job waits for I/O, the CPU switches to another. This dramatically improved CPU utilization. The OS needed new capabilities: **memory protection**, **job scheduling**, and **interrupt handling**.

3.3 Time-Sharing and UNIX (1970s)

Time-sharing extended multiprogramming by rapidly switching between users, giving each the illusion of a dedicated machine.

- **UNIX** was created in 1969 at Bell Labs by Ken Thompson and Dennis Ritchie.
- It was rewritten in **C** in 1973 — one of the first OS written in a high-level language.
- Introduced the concept of **pipes** (`cmd1 | cmd2`), enabling modular composition of small programs.
- "Everything is a file" philosophy unified device and file access.

3.4 Personal Computing (1980s–1990s)

- **MS-DOS** (1981) — single-user, single-tasking, command-line interface.
- **Windows** evolved from a DOS shell to a full OS (Windows NT, 1993).
- **Linux** (1991) — Linus Torvalds released a free UNIX-like kernel; combined with GNU tools, it became a full OS.

3.5 Modern OS Landscape (2000s–present)

- **Mobile OS** — Android (Linux-based) and iOS dominate billions of devices.
- **Virtualization** — hypervisors (VMware, KVM, Xen) allow multiple OS instances on one machine.
- **Containers** — Docker, Kubernetes provide lightweight isolation using OS-level virtualization (cgroups, namespaces).
- **Cloud computing** — OS designed for data centers; thousands of machines managed as one.

4 Kernel Architecture

Type	Design	Pros	Cons	Examples
Monolithic	All services in kernel space	Fast (no IPC overhead); simple communication	Large attack surface; one bug can crash entire system	Linux, FreeBSD
Microkernel	Minimal kernel; services in user space	Reliable; modular; easy to extend	IPC overhead; complex design	Minix, L4, QNX
Hybrid	Mix of monolithic and micro	Balanced performance and modularity	Complexity; "worst of both worlds" risk	Windows NT, macOS (XNU)
Exokernel	Exposes hardware directly; library OS	Maximum flexibility and performance	Very complex for app developers	MIT Exokernel

4.1 Monolithic Kernel

The entire OS runs as a single program in kernel mode. All kernel services — scheduling, file systems, device drivers, networking — share the same address space and can call each other directly. Linux is the most prominent example.

Linux kernel structure:

- **System call interface** — entry point from user space (about 450 syscalls).
- **Process management** — scheduler (CFS), fork/exec, signals.
- **Memory management** — virtual memory, page tables, slab allocator.
- **VFS (Virtual File System)** — unified interface to ext4, XFS, Btrfs, NFS, etc.
- **Network stack** — TCP/IP, socket layer, netfilter.
- **Device drivers** — largest part of the kernel codebase (over 70%).
- **Loadable Kernel Modules (LKMs)** — drivers can be loaded/unloaded at runtime without rebooting, giving some microkernel-like flexibility.

4.2 Microkernel

Only the most essential services run in kernel mode: IPC, basic scheduling, and address space management. Everything else (file systems, device drivers, networking) runs as user-space server processes. Communication happens via message passing (IPC), which introduces overhead but provides strong fault isolation.

4.3 Hybrid Kernel

Combines aspects of both: keeps performance-critical services in kernel space while allowing some modularity. Windows NT keeps the window manager and graphics driver in kernel space for performance. macOS uses the XNU kernel, which pairs a Mach microkernel with BSD components.

4.4 Exokernel

The kernel provides minimal abstractions; instead, it *securely exports* hardware resources to user-level library operating systems. Applications can implement their own file systems, networking stacks, and scheduling policies. This yields extreme performance but at the cost of application complexity.

5 User Mode vs Kernel Mode

Modern CPUs provide **hardware protection rings** to separate privileged and unprivileged code:

- **Ring 0 (Kernel Mode)** — full access to all hardware and instructions.
- **Ring 3 (User Mode)** — restricted; cannot execute privileged instructions.
- Rings 1 and 2 exist in x86 architecture but are rarely used by modern OSes.

User Mode Restrictions

A process running in user mode **cannot**:

- Access hardware directly (disk, NIC, GPU registers).
- Execute privileged instructions (`HLT`, `LGDT`, `MOV` to control registers, `IN`/`OUT`).
- Access kernel memory (pages marked supervisor-only in page tables).
- Disable or mask interrupts.
- Change the CPU mode bit directly.

Attempting any of these triggers a **general protection fault** (trap to kernel).

5.1 Kernel Mode

Code running in kernel mode has **full, unrestricted access** to:

- All physical memory and I/O ports.
- All CPU instructions, including privileged ones.
- All hardware registers and interrupt controllers.
- The ability to modify page tables and control registers.

5.2 Mode Switching

When a user process needs OS services, a **mode switch** occurs:

1. User process executes a **trap instruction** (e.g., `syscall` on x86-64, `int 0x80` on older x86).
2. CPU saves the current program counter and status register to the kernel stack.
3. CPU switches the mode bit from user (ring 3) to kernel (ring 0).
4. CPU jumps to the pre-configured **trap handler** (system call dispatch table).
5. Kernel executes the requested service.
6. Kernel executes `sysret`/`iret` to return; CPU restores registers and mode bit.

5.3 Cost of Mode Switching

`Performance Impact`

A mode switch typically costs **1–10 µs** and involves:

- Saving and restoring registers (general-purpose, FPU/SSE/AVX state).
- Switching page tables (updating CR3 on x86).
- Flushing the TLB (Translation Lookaside Buffer), though tagged TLBs (PCID) mitigate this.
- Cache pollution — kernel code/data may evict user cache lines.

This is why minimizing system calls is important for high-performance applications.

6 System Calls

A **system call** is the *only* programmatic way for a user-space process to request services from the kernel. System calls provide a controlled entry point into the kernel, maintaining protection while enabling functionality.

6.1 Categories of System Calls

Category	Example System Calls
Process Control	`fork()`, `exec()`, `exit()`, `wait()`, `kill()`, `getpid()`
File Management	`open()`, `read()`, `write()`, `close()`, `lseek()`, `stat()`
Device Management	`ioctl()`, `read()`, `write()`, `mmap()`
Information	`getpid()`, `alarm()`, `time()`, `uname()`, `sysinfo()`
Communication	`pipe()`, `socket()`, `send()`, `recv()`, `shmget()`, `mmap()`
Memory Management	`brk()`, `mmap()`, `munmap()`, `mprotect()`

6.2 Example: The `write()` System Call

```c
#include <unistd.h>

int main(void) {
    const char *msg = "Hello, OS!\n";
    // write(fd, buffer, count)
    //   fd = 1 (stdout)
    //   buffer = pointer to the string
    //   count = 11 bytes
    write(1, msg, 11);
    return 0;
}
```

What happens inside `write(1, msg, 11)`

1. The C library wrapper loads the syscall number for `write` (1 on x86-64 Linux) into `%rax`.
2. Arguments are placed in registers: `%rdi` = 1, `%rsi` = `msg`, `%rdx` = 11.
3. The `syscall` instruction traps into the kernel.
4. The kernel validates the file descriptor, copies data from user buffer to kernel buffer, and writes to the device.
5. The kernel returns the number of bytes written (or -1 on error) in `%rax`.

6.3 Tracing System Calls with `strace`

`strace` intercepts and logs every system call made by a process:

```
# Trace all syscalls of a command
strace ./my_program

# Count syscalls and show summary
strace -c ./my_program

# Trace only file-related syscalls
```

```
strace -e trace=file ./my_program

# Trace a running process by PID
strace -p 12345

# Show timestamps for each syscall
strace -t ./my_program
```

7 Key Takeaways

Week 1 Summary

1. An OS **virtualizes** hardware (CPU → processes, memory → address spaces, disk → files), manages **concurrency**, and ensures **persistence**.
2. OS architecture has evolved from no OS (1940s) through batch, multiprogramming, time-sharing, to modern cloud-native and containerized systems.
3. Kernel designs range from **monolithic** (Linux) to **microkernel** (L4, QNX) to **hybrid** (Windows, macOS), each with distinct trade-offs in performance, reliability, and complexity.
4. The CPU enforces **dual-mode operation** (user mode vs. kernel mode) via hardware protection rings, ensuring that user programs cannot corrupt the OS or other processes.
5. **System calls** are the only controlled gateway from user space to kernel space; understanding their mechanism and overhead is essential for systems programming.