Lecture 2: Operating Systems

Yonghao Lee

March 31, 2025

What is a Kernel?

The kernel is the core component of an operating system. Think of it as the heart of the OS – the part that's always running and controls everything else. The kernel has several essential responsibilities:

- Managing the computer's hardware resources (CPU, memory, devices)
- Allowing programs to run and use these resources safely
- Ensuring different programs don't interfere with each other
- Providing basic services that all programs need

When you start your computer, the kernel is one of the first things that loads. It stays in memory the entire time your computer is running, acting as a bridge between your hardware and the software applications you use. Without a kernel, your applications would have no standardized way to interact with your computer's hardware.

Different operating systems have different types of kernels (monolithic, modular, micro, or hybrid), but they all serve this fundamental purpose of managing hardware and providing services to software.

Operating System Architecture

The Role of the Operating System

The operating system functions as a crucial intermediary layer between applications/users and the hardware. This architecture creates a controlled environment where resources are managed efficiently while providing isolation and protection. Information flows bidirectionally through the OS via two primary mechanisms:

- **System calls:** Applications request services from the hardware through the OS (e.g., when a user asks to print documents). This represents the flow from applications to hardware.

- **Interrupts:** Hardware events trigger responses in the software (e.g., when a user clicks the mouse and the computer reacts). This represents the flow from hardware to applications.

Privileged instructions are special commands that only the OS can execute, which enables it to safely manage these interactions between applications and hardware.

Processor Operation Modes

Modern operating systems implement a critical security boundary by operating in two distinct modes:

- **Kernel Mode:** Also called privileged mode or supervisor mode, this allows execution of all instructions, including both privileged and non-privileged instructions. In this mode, the OS has complete access to all hardware resources and can execute any assembly commands that directly control hardware.
- **User Mode:** In this restricted mode, only non-privileged instructions can be executed. Applications running in user mode cannot directly access hardware or manipulate critical system resources. This restriction helps maintain system stability and security.

The processor architecture distinguishes between two fundamental classes of instructions:

- **Non-privileged instructions:** These include all standard operations that user applications can execute, such as arithmetic operations (add, multiply), control flow (jump), and data manipulation instructions.
- **Privileged instructions:** These are special instructions reserved for the operating system. They provide direct access to hardware resources and sensitive system functions. Examples include instructions that activate disk I/O, manage memory allocation, or modify system control registers.

The distinction between these modes creates a protective barrier that prevents user applications from interfering with critical system operations. While the boundary between these modes may sometimes appear blurred in certain operating system designs, this dual-mode operation is fundamental to modern OS architecture.

Kernel Types and Designs

Operating system kernels can be classified into several architectural categories, each with distinct characteristics and trade-offs:

Monolithic Kernels

- All OS services run in kernel space as a single large binary
- Tightly coupled components with direct function calls between subsystems
- Examples: MS-DOS, early Unix systems
- Advantages: Fast execution due to direct function calls
- Disadvantages: Difficult to maintain, any bug can crash the entire system

Modular Monolithic Kernels

- Core kernel plus dynamically loadable modules
- Maintains monolithic performance while adding flexibility
- Examples: Linux, Windows, macOS, modern Unix systems
- Advantages: Allows extending kernel without recompilation, easier maintenance
- Disadvantages: Still has large portions of OS in kernel space

Kernel Design Trade-offs

The size and architecture of the kernel involve fundamental trade-offs that impact system performance, reliability, and functionality:

Advantages of Smaller Kernels

- **Performance:** Fewer traps (transitions between user and kernel mode) lead to less overhead and higher throughput
- **Memory efficiency:** Smaller memory footprint leaves more physical memory available for applications
- **Flexibility:** More functions outside the kernel mean no need to recompile the kernel when they change

Disadvantages of Smaller Kernels

- **Reduced protection:** If security features are moved out of the kernel, they might be less effective

System Calls

Purpose and Function

System calls serve as the bridge between user mode and kernel mode. They provide a controlled interface through which user-level applications can request services from the operating system kernel.

Key characteristics of system calls:

- They initiate a switch from user mode to kernel mode, allowing temporary access to privileged operations
- They are not regular function calls, as they involve a special processor mechanism
- They provide the only legitimate way for user programs to access or modify internal OS data structures
- They include validation checks to ensure parameters are valid before executing privileged operations
- They offer a portable interface, allowing applications to work across different hardware implementations
- They establish a protection layer, preventing malicious or buggy programs from compromising system integrity

System Call Mechanism

The transition from user to kernel mode occurs through a specialized hardware mechanism:

- When an application makes a system call (e.g., `open()` in C), it's actually invoking a library function provided by the operating system
- This library function prepares for the kernel transition by setting up parameters and system call numbers in specific registers
- The function then executes a "trap" instruction, which is a special processor instruction that changes the mode bit from user mode (1) to kernel mode (0)
- Once in kernel mode, the system can execute privileged instructions
- After completing the requested operation, a return mechanism changes the mode bit back to user mode (1)

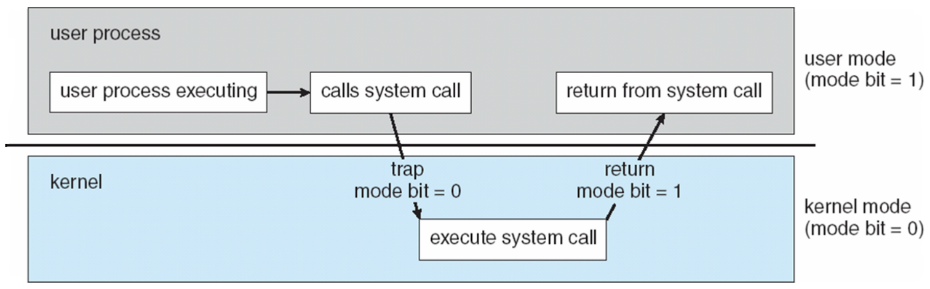


Figure 1: Transition from User to Kernel Mode

It's important to note that the trap mechanism is implemented at the processor hardware level, not by the operating system itself. Modern processors are designed with these mechanisms to support operating system requirements.

Detailed Trap Process

From User to Kernel: The Trap Mechanism

1. A user application invokes a library function (e.g., `open()`)
2. The library function:
   - Places the system call number in a designated register
   - Arranges parameters according to the calling convention
   - Executes a special instruction to trigger the trap
3. The trap causes a privilege level change from user mode to kernel mode
4. Control transfers to the system call entry point
5. The entry point dispatcher examines the system call number
6. Based on the number, it calls the appropriate implementation function, passing the parameters

The Return Path: From Kernel Back to User

1. The system call implementation (e.g., `sys_open()`) completes its operation
2. When the system call ends it returns to the entry point function
3. The entry point function:
   - Stores the return value in a designated register for user space access
   - Prepares for the transition back to user mode
4. A special instruction returns execution to user mode at the instruction after the trap

5. The library function retrieves the return value from the designated register
6. It returns this value to the calling application

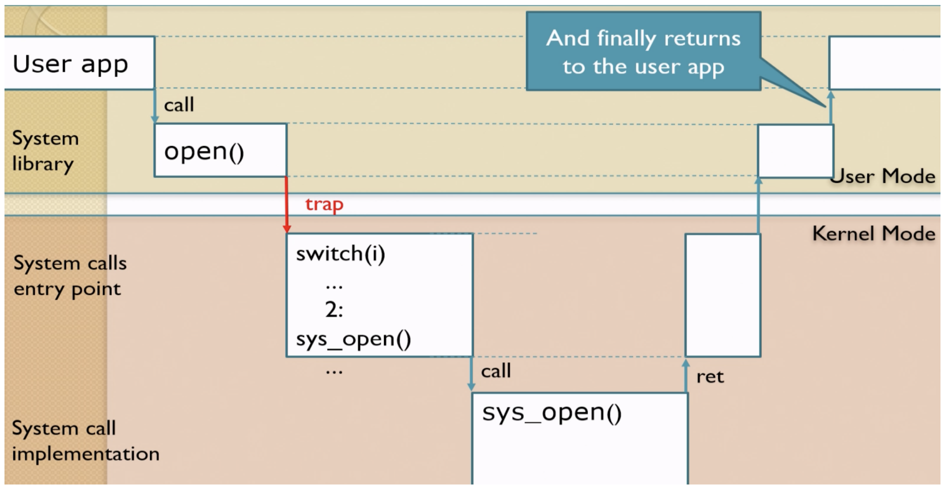


Figure 2: Detailed Trap Process

This process introduces significant overhead, which has led to optimizations in modern systems.

Categories of System Calls

System calls can be categorized into several groups:

- **Process Control:** Create, terminate, load, execute processes; get/set process attributes
- **File Management:** Create, delete, open, close, read, write files; get/set file attributes
- **Device Management:** Request/release devices; read/write device registers
- **Information Maintenance:** Get/set time, date, system data; get process/file/device attributes
- **Communication:** Create/delete communication connections; send/receive messages
- **Protection:** Set/get permissions; allow/deny access to resources

Modern System Call Optimization: SYSCALL

Traditional trap-based system calls involve significant overhead due to the interaction with the interrupt vector table. Modern processors provide a more optimized instruction called SYSCALL (on x86-64 architecture) to handle system calls more efficiently:

- SYSCALL bypasses the interrupt vector table
- It provides a more direct path to kernel mode
- It reduces the number of instructions required for the transition
- It preserves only essential processor state
- Performance is approximately doubled compared to the traditional mechanism

Interrupts and Exceptions

Transition Mechanisms

While system calls represent a voluntary transfer of control from user to kernel mode, both interrupts and exceptions represent involuntary transfers:

- **Exceptions:** Occur when something goes wrong during program execution, such as:
  - Division by zero
  - Illegal memory access (segmentation violation)
  - Attempt to execute invalid or privileged instructions
- **Interrupts:** Generated by hardware devices to signal events requiring OS attention

Exception handling is built into the CPU architecture and transfers control to the OS for handling the exceptional condition.

Interrupt Mechanism

Interrupt Basics

Devices use interrupts to signal that they need service, which can be:

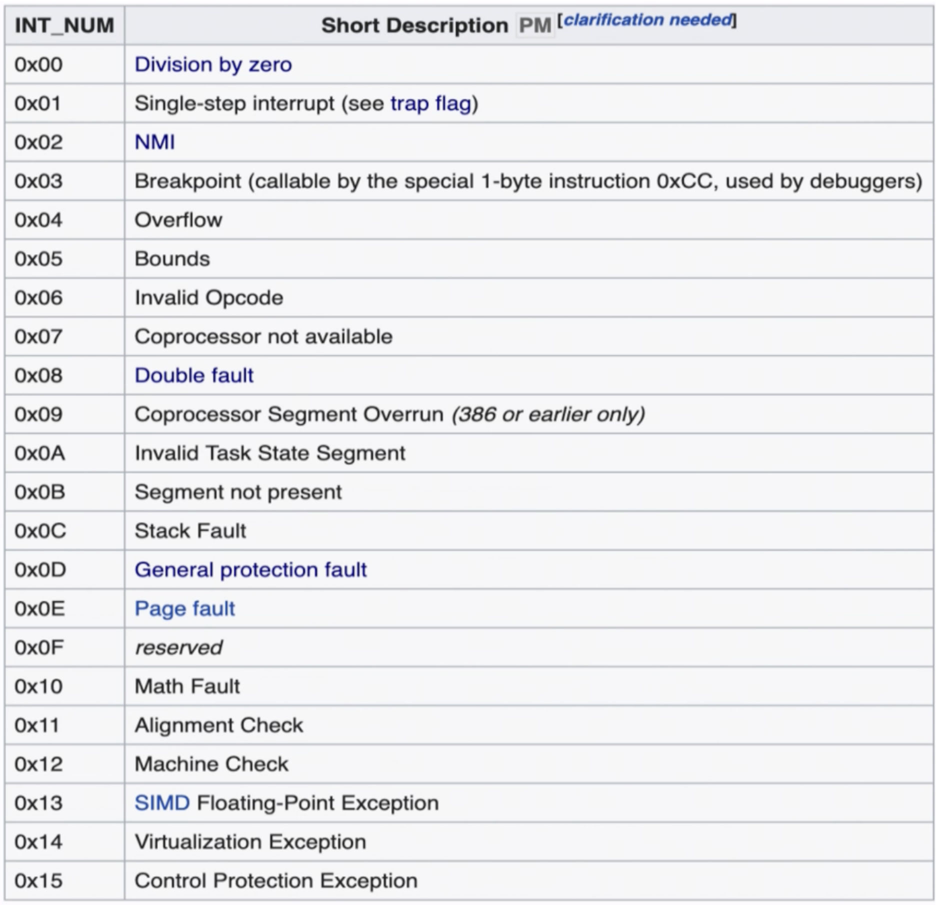
- Completion of an I/O operation
- Detection of input from an external source
- Timer expiration for scheduling
- And more

An interrupt causes the CPU to suspend its current execution and transfer control to a specific part of the operating system called an **interrupt handler**. This mechanism is architecturally similar to exceptions and system calls, but with a key difference: interrupts originate from hardware devices rather than from software execution.

Interrupt Vector Table

When an interrupt occurs, the CPU needs to quickly locate and execute the appropriate interrupt handler. The solution is the **interrupt vector table** (IVT):

- A region of memory containing addresses of interrupt handlers
- Each entry corresponds to a specific interrupt type or device
- The location of this table is defined by the processor architecture
- For example, in traditional x86 systems, the first 1KB of memory was reserved for this purpose



INT_NUM	Short Description PM [*clarification needed*]
0x00	Division by zero
0x01	Single-step interrupt (see trap flag)
0x02	NMI
0x03	Breakpoint (callable by the special 1-byte instruction 0xCC, used by debuggers)
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor Segment Overrun *(386 or earlier only)*
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Fault
0x0D	General protection fault
0x0E	Page fault
0x0F	*reserved*
0x10	Math Fault
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception

Figure 3: Examples of Interrupts

Interrupt Handling Process

The complete interrupt handling sequence:

1. A device generates an interrupt signal on the system bus

2. The interrupt controller prioritizes and forwards the interrupt to the CPU
3. The CPU completes its current instruction and saves its current state
4. The CPU determines the interrupt number, which serves as an index into the interrupt vector table
5. The CPU loads the address from the corresponding entry in the interrupt vector table
6. Control transfers to the interrupt handler at that address
7. After handling the interrupt, the system restores the CPU state and resumes the interrupted process

Hardware Response to Interrupts

When an interrupt signal is received, the hardware automatically performs two critical operations:

- **Blind Address Loading:** The CPU mechanically retrieves and follows the address stored in the interrupt vector table without evaluation or processing.
- **Privilege Escalation:** The CPU sets the processor mode to kernel mode, ensuring the interrupt handler has sufficient privileges to interact with hardware and access protected memory.

Classifications of Interrupts

Interrupts can be classified in several ways:

- **Software vs. Hardware**
  - Software: Generated by software operations (including traps and exceptions)
  - Hardware: Generated by external devices
- **Internal vs. External**
  - Internal: Generated within the CPU (generally equivalent to software interrupts)
  - External: Generated by external devices (generally equivalent to hardware interrupts)
- **Synchronous vs. Asynchronous**
  - Synchronous: Occur in direct response to instruction execution
  - Asynchronous: Occur independently of the instruction stream (like device interrupts)
- **Maskable vs. Non-Maskable**
  - Maskable: Can be temporarily disabled/ignored by the CPU
  - Non-Maskable: Critical interrupts that cannot be disabled

Device Management

Device Controllers

I/O devices have specialized hardware components called **controllers** that manage their operation:

- Controllers contain firmware that implements device-specific logic
- They act as an interface between the main system and the actual I/O device
- Controllers have registers that can be accessed by the CPU through specific bus addresses

Device Drivers

Each device has a corresponding **device driver**, which is software that runs as part of the operating system:

- Drivers contain code that knows how to communicate with specific device controllers
- They translate high-level OS requests into device-specific commands
- Drivers often execute in kernel mode with privileged access

I/O Operation Process

When an application needs to perform I/O (e.g., writing to a hard disk):

1. The application makes a system call to the OS
2. The OS invokes the appropriate device driver
3. The driver writes specific commands to the controller's registers
4. The controller interprets these commands and initiates the actual I/O operation
5. Upon completion, the controller generates an interrupt
6. The OS handles the interrupt and notifies the application

System Components and Layers

System Architecture Layers

Modern operating systems are structured in layers:

User Applications:
Programs running in user mode that utilize OS services

System Programs:	Utilities and tools that provide interfaces to OS functionality
User Interface:	Making OS mechanisms available to users through shells or GUIs
Libraries:	Language-specific abstractions that simplify access to OS services
OS Abstractions:	Lower-level abstractions for storage, computation, and communication
Hardware Drivers:	Interfaces to physical hardware components
Hardware:	Physical components including CPU, memory, and peripherals

System Libraries vs. System Calls

An important distinction exists between system libraries and system calls:

- System libraries (like C library functions) run in user mode
- They provide a convenient programming interface
- They may call multiple system calls to implement their functionality
- Only actual system calls trigger the transition to kernel mode

For example, the `echo` system program uses library functions like `fputs` and `putchar`, which ultimately rely on the `write` system call to output bytes to the screen.

Protection Rings

Modern processors support more nuanced privilege separation than just user/kernel modes:

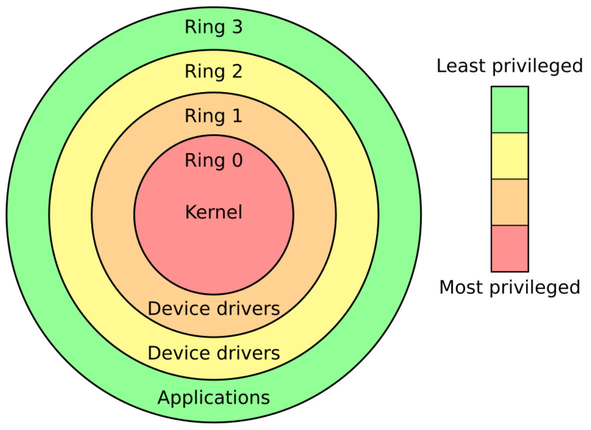


Figure 4: Protection rings for privilege separation in modern CPUs

These protection rings allow for more granular control:

- Ring 0: Core kernel functionality (highest privilege)
- Ring 1: Device drivers and kernel extensions
- Ring 2: Privileged system services
- Ring 3: User applications (lowest privilege)

While these rings exist in hardware, most operating systems primarily use Ring 0 (kernel mode) and Ring 3 (user mode) due to performance considerations.

Hardware Communication

Connecting I/O Devices

I/O devices connect to the system through buses, which are specialized communication pathways:

- **Advantages:**
  - Versatility: New devices can be added easily
  - Modularity: Components can be moved between computer systems
  - Cost-effectiveness: Standardized interfaces reduce manufacturing costs
- **Disadvantages:**
  - Communication bottleneck: Bus bandwidth limits I/O throughput
  - Physical constraints: Maximum bus speed limited by physical length

Throughput Considerations

Bus design must account for:

- Total bandwidth requirements of all connected devices
- Conflict prevention when multiple devices attempt simultaneous communication
- Support for devices with varying latencies and data transfer rates

Summary: Communication Pathways

The operating system interacts with hardware and applications through three primary mechanisms:

- **System Calls:** Voluntary transitions from user to kernel mode for requesting OS services
- **Interrupts:** Hardware-initiated signals for attention from the OS
- **Exceptions:** CPU-detected error conditions that transfer control to the OS

These mechanisms, sometimes collectively referred to as ”interrupts” in a broader sense, form the foundation of the operating system's ability to maintain control while providing services to applications.