Theory of Operating Systems

Week 1: Introduction to Operating Systems

Faye Chi Zhang Richard Cai Ping Ji

CUNY Hunter

January 28, 2026

Contact

- Faye Chi Zhang: `czhang2@gradcenter.cuny.edu`
- Richard Cai: `jcai@gradcenter.cuny.edu`
- Ping Ji: `Ping.Ji@hunter.cuny.edu`

Today's Outline

1. Course Overview and Policies
2. What is an Operating System?
3. OS History and Evolution
4. System Architecture and Kernel Designs
5. User Mode vs Kernel Mode
6. System Calls: The OS Interface
7. Hands-on: Tracing System Calls

Welcome: Why Study Operating Systems?

- **The Ultimate Software Challenge**: Building systems that are:
  - **Efficient**: Direct control over CPU, RAM, and storage
  - **Abstract**: Clean APIs hiding hardware complexity
  - **Robust**: One crash shouldn't bring down everything
- **Our Mission**: From API *Consumer* to System *Designer*

The OS Philosophy

An Operating System is a **government** for hardware resources—managing conflict, enforcing security, and providing essential services.

What We Will Master

Three Fundamental Concepts:

- **Virtualization**: Trick processes into thinking they own the CPU and all RAM
  - Processes, Scheduling, Memory Management
- **Concurrency**: Manage chaos when threads fight for shared data
  - Threads, Locks, Semaphores, Deadlocks
- **Persistence**: Store data reliably on failure-prone devices
  - File Systems, I/O, Storage

Course Policies: Grading

Undergraduate Students

Assignments (8)	40%
Weekly Quizzes (8–10)	20%
Midterm Exam	30%
Final Exam	*TBA*
Participation	5%

Note: Final exam weight ??.

Masters Students

Assignments (8)	35%
Weekly Quizzes (8–10)	15%
Midterm Exam	20%
Research Project	20%
Participation	5%

Technical Requirements

- **Platform**: All coding via the **Course Online IDE**
- **Zero Setup**: No local environment needed
- **Supported Languages**:
  - C / C++ (GCC 9.2.0)
  - Python (3.8.1)
- **Grading**: Automated with hidden test cases

Late Policy

Submissions accepted up to 48 hours late with 20% penalty per 24-hour period.

Course Schedule Overview

Week	Topic
1	Introduction to Operating Systems
2	Process Management
3	Process Scheduling Algorithms
4	Inter-Process Communication
5	Threads and Concurrency
6-7	Synchronization and Deadlocks
8-9	Memory Management (Midterm Week 9)
10-12	Virtual Memory and File Systems
13-16	I/O, Storage, Security, Modern Topics

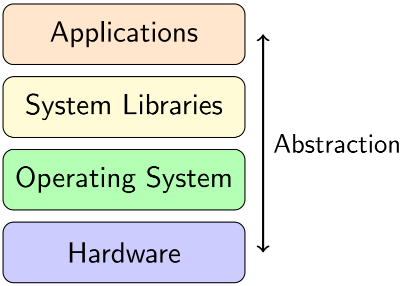
What is an Operating System?

Definition:

- Software layer between applications and hardware
- Manages and allocates system resources
- Provides abstractions for complex hardware

Key Perspective:

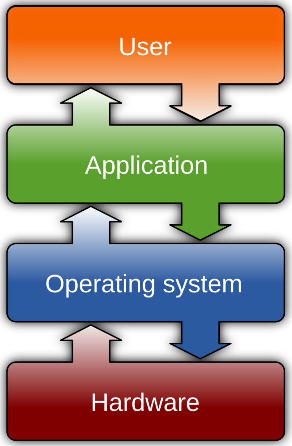
- Programs are **state machines**
- OS = **State machine manager**
- Controls transitions between states



Layered System Design:

- User applications at top
- OS kernel mediates access
- Hardware at the bottom
- Each layer provides services to layer above

Source: Wikimedia Commons (CC BY-SA)



Core Functions of an Operating System

Resource Management:

- **CPU**: Process scheduling
- **Memory**: Allocation, virtual memory
- **Storage**: File systems
- **I/O Devices**: Device drivers
- **Network**: Protocol stacks

Key Services:

- **Multiplexing**: Share hardware among apps
- **Isolation**: Protect apps from each other
- **Abstraction**: Hide hardware complexity
- **Security**: Access control
- **Communication**: IPC mechanisms

The OS as a Resource Manager

Problem: Multiple programs want to use limited resources simultaneously

- **CPU**: Only one instruction executes at a time (per core)
  - Solution: Time-sharing, scheduling algorithms
- **Memory**: Limited physical RAM
  - Solution: Virtual memory, paging
- **Disk**: Single read/write head
  - Solution: I/O scheduling, buffering
- **Network**: Shared bandwidth
  - Solution: Packet scheduling, QoS

The OS as an Abstraction Provider

Hardware is messy. OS provides clean interfaces.

Hardware Reality	OS Abstraction
CPU registers, instructions	Process (virtual CPU)
Physical RAM addresses	Virtual address space
Disk sectors, blocks	Files and directories
Network packets, buffers	Sockets and streams
Interrupts, I/O ports	Device-independent I/O

Key Insight

Programmers write to abstractions, not hardware. This enables portability and simplicity.

What's Inside an Operating System?

Core Components:

- **Process Manager**: Create, schedule, terminate
- **Memory Manager**: Allocate, map, protect
- **File System**: Organize, store, retrieve
- **I/O Manager**: Device drivers, buffering
- **Network Stack**: Protocols, sockets

Supporting Components:

- **Scheduler**: CPU allocation policy
- **Interrupt Handler**: Hardware events
- **System Call Interface**: User/kernel boundary
- **Security Module**: Access control
- **Clock/Timer**: Time management

The Computer Model: Von Neumann Architecture

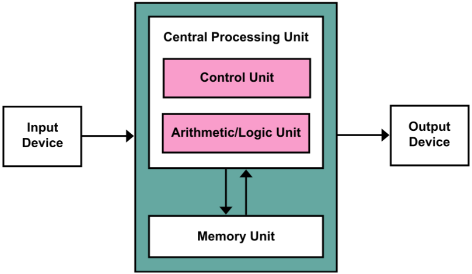
Key Components:

- **CPU**: Executes instructions
- **Memory**: Stores programs and data
- **Input/Output**: External communication
- **Bus**: Connects components

Key Insight:

- Programs stored in memory
- Same memory for code and data
- Sequential instruction execution

Source: Wikimedia Commons



OS History: Evolution of Computing

Era	Characteristics
1940s-50s	No OS. Manual operation, one job at a time
1960s	Batch processing, multiprogramming
1970s	Time-sharing, UNIX (1969), C language
1980s	Personal computers, DOS, early Mac OS
1990s	GUI explosion, Windows NT, Linux (1991)
2000s	Mobile OS (iOS, Android), virtualization
2010s+	Cloud computing, containers, microservices

Early Computing: No Operating System

1940s-1950s: The Dark Ages

- **Direct Hardware Access**: Programmers controlled everything
- **Single User**: One program at a time
- **Manual Operation**:
  - Load program via punch cards
  - Press buttons to start
  - Wait for output (often hours)
  - Debug by examining lights
- **Problem**: Expensive computers sat idle between jobs

Cost

A computer cost millions. Human time was cheap. Machine time was precious.

IBM Model 701 (Early 1950s): Before Modern OS

IBM Model 701 (Early 1950's)

Context: Expensive machine time, manual operation, and long turnaround.

Batch Processing Systems

1960s: First Operating Systems

Key Innovation:

- Operator batches similar jobs
- OS automatically loads next job
- Reduced setup time
- Better CPU utilization

Limitations:

- No interaction during execution
- Long turnaround (hours/days)
- CPU idle during I/O
- One job at a time in memory

Solution needed: Keep CPU busy during I/O waits

Batch Era Mainframes (Early 1960s)

IBM 7094 (Early 1960s)

IBM System/360 Console

Why OS mattered: reduce idle time, automate job sequencing, and manage I/O.

Multiprogramming

Key Insight: While one job waits for I/O, run another!

How It Works:

- Multiple jobs in memory
- When Job A blocks on I/O
- Switch to Job B
- CPU never idle (if jobs available)

New Challenges:

- Memory protection needed
- Job scheduling decisions
- Resource allocation
- Deadlock prevention

Impact

CPU utilization jumped from 30% to 80%+. This was revolutionary.

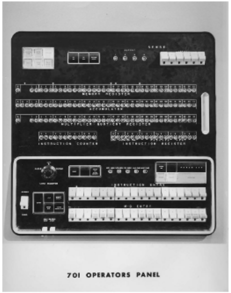
Operator Console: Early Human–Machine Interface

What the operator did:

- Load jobs / tapes
- Start/stop execution
- Monitor status lights
- Handle failures / I/O issues

Console shows how "interactive" early computing really was (for operators).

IBM 701 Console



Time-Sharing Systems

1970s: Interactive Computing

- **Problem**: Batch systems had no interactivity
- **Solution**: Give each user a time slice of CPU
- **Illusion**: Each user thinks they have the whole computer

UNIX (1969):

- Developed at Bell Labs by Thompson and Ritchie
- Written in C (portable!)
- Hierarchical file system
- Pipes for inter-process communication
- Foundation for modern OS design

Personal Computing Era

1980s-1990s: Computers for Everyone

MS-DOS (1981):

- Single-user, single-task
- Command-line interface
- No memory protection
- Direct hardware access

Windows/Mac:

- Graphical user interface
- Mouse-driven interaction
- Eventually: protected memory
- Multitasking support

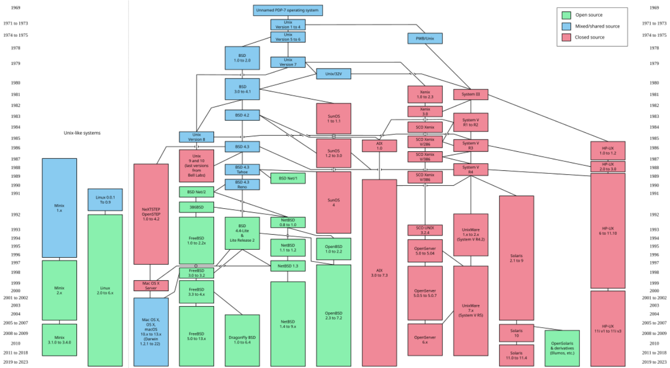
Linux (1991): Free, open-source UNIX-like OS by Linus Torvalds

UNIX Family Tree

UNIX Heritage:

- Original UNIX (1969)
- BSD (1977)
- System V (1983)
- Linux (1991)
- macOS (Darwin/BSD)
- Android (Linux kernel)

Source: Wikimedia Commons



Modern Operating Systems

2000s-Present

- **Mobile**: iOS (2007), Android (2008)
    - Touch interfaces, power management, app sandboxing
- **Virtualization**: VMware, Xen, KVM
    - Multiple OS on one machine
- **Containers**: Docker (2013), Kubernetes
    - Lightweight isolation, microservices
- **Cloud**: AWS, Azure, GCP
    - OS as a service, serverless computing

10-Minute Break

We'll continue with System Architecture

Kernel Architecture: Design Choices

Question: What should run in the privileged kernel?

- **More in kernel**: Faster, but harder to debug, less secure
- **Less in kernel**: Slower, but more modular, more reliable

Four Main Approaches:

1. Monolithic Kernel
2. Microkernel
3. Hybrid Kernel
4. Exokernel / Unikernel

Monolithic Kernel

Design:

- Entire OS runs in kernel space
- All services in one address space
- Direct function calls between components

Advantages:

- High performance
- No IPC overhead
- Simple design

Disadvantages:

- Large attack surface
- Bug in one module crashes all
- Hard to maintain (millions of lines)

Examples:

- Linux
- BSD (FreeBSD, OpenBSD)
- Traditional UNIX

Microkernel

Design:

- Minimal kernel: IPC, scheduling, memory
- Services run in user space
- Communication via message passing

Advantages:

- Small trusted computing base
- Fault isolation
- Easier to verify/prove correct

Disadvantages:

- IPC overhead
- More context switches
- Complex design

Examples:

- Minix
- seL4 (formally verified!)
- QNX (real-time)
- GNU Hurd

Hybrid Kernel and Alternatives

Hybrid Kernel:

- Combines monolithic and micro
- Some services in kernel for speed
- Some modularity preserved
- Examples: Windows NT, macOS

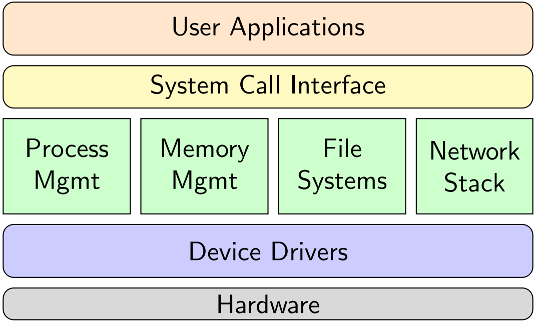
Unikernel:

- Single-purpose, library OS
- Application + OS compiled together
- Minimal footprint
- Examples: MirageOS, IncludeOS

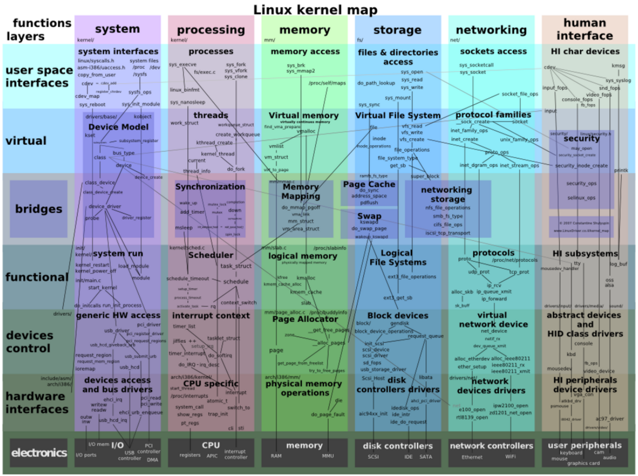
Exokernel:

- Minimal abstraction
- Apps manage own resources
- Maximum flexibility

Case Study: Linux Kernel Structure



Linux Kernel: Interactive Map



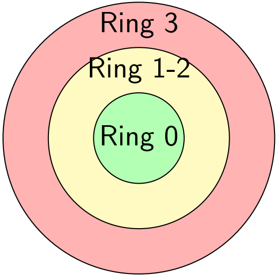
Source: Wikimedia Commons - Linux Kernel Map (CC BY-SA)

The Protection Boundary

Why do we need protection?

- Prevent buggy apps from crashing the system
- Prevent malicious apps from stealing data
- Ensure fair resource sharing

Hardware Support: Protection Rings (x86)



User applications

(rarely used)

Kernel (most privileged)

x86 Protection Rings Architecture

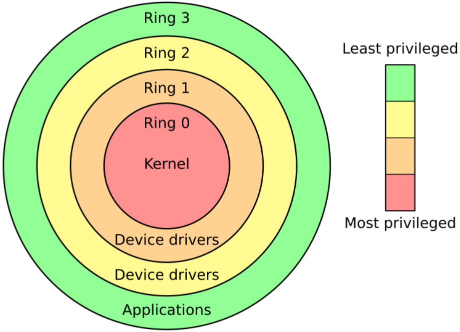
Intel x86 Privilege Levels:

- **Ring 0**: Kernel/OS (full access)
- **Ring 1-2**: Device drivers (rarely used in modern OS)
- **Ring 3**: User applications (restricted)

Modern Usage:

- Most OS use only Ring 0 and Ring 3
- Hypervisors may use Ring -1 (VMX root)

Source: Wikimedia Commons (CC BY-SA)



User Mode (Ring 3)

Restricted Environment for Applications

Cannot:

- Access hardware directly
- Execute privileged instructions
- Access kernel memory
- Modify page tables
- Disable interrupts

Can:

- Execute normal instructions
- Access own memory
- Make system calls
- Use CPU registers
- Perform computations

Violation

Attempting privileged operations triggers a **protection fault** — the OS terminates the process.

Kernel Mode (Ring 0)

Full Privileges for the Operating System

Full Access To:

- All memory (physical and virtual)
- All CPU instructions
- All I/O ports
- All hardware devices
- Interrupt handling

Responsibilities:

- Process management
- Memory management
- Device drivers
- File system operations
- Network stack
- Security enforcement

Trust

Kernel code is trusted. A bug here affects the entire system.

Switching Between Modes

How does a user program request kernel services?

1. **System Call**: User program invokes syscall instruction
2. **Trap**: Hardware switches to kernel mode
3. **Handler**: Kernel executes the requested service
4. **Return**: Kernel returns result, switches back to user mode

Other ways to enter kernel mode:

- **Interrupt**: Hardware device needs attention
- **Exception**: Error (divide by zero, page fault)

The Cost of Mode Switching

Mode switch is expensive!

- **Save user state**: Registers, program counter
- **Switch page tables**: Different address space
- **Flush TLB**: Translation cache invalidated
- **Cache pollution**: Kernel code displaces user code

Typical cost: 1-10 microseconds

Design Implication

Minimize system calls in performance-critical code. Batch operations when possible.

System Calls: The OS Interface

Definition: The programmatic interface to OS services

- **Only way** for user programs to access kernel services
- Controlled entry points into the kernel
- Each syscall has a unique number
- Parameters passed via registers

Analogy

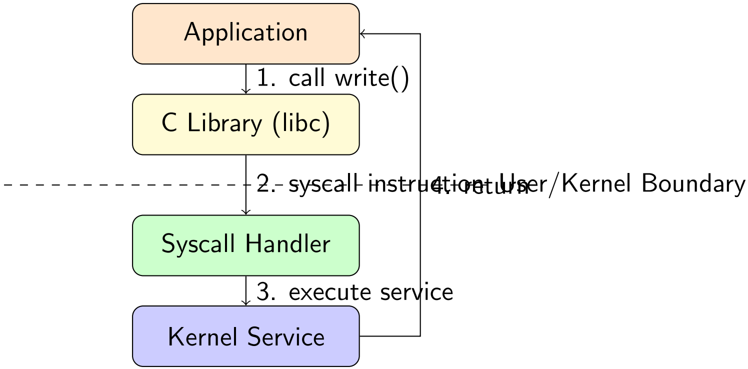
System calls are like a restaurant menu. You can only order what's on the menu — you can't go into the kitchen yourself.

Categories of System Calls

Category	Examples
Process Control	fork, exec, exit, wait, kill
File Management	open, read, write, close, stat
Device Management	ioctl, read, write
Information	getpid, time, uname
Communication	pipe, socket, send, recv
Memory	mmap, brk, mprotect

Linux has ˜400 system calls. Windows has ˜2000+.

How System Calls Work



Example: The write() System Call

```c
#include <unistd.h>

int main() {
    char *msg = "Hello, OS!\n";

    // System call: write to stdout
    write(1, msg, 11);

    return 0;
}
```

write() parameters:

- `fd = 1`: File descriptor (stdout)
- `buf = msg`: Data to write
- `count = 11`: Number of bytes

Returns: Number of bytes written, or -1 on error

Essential System Calls

Process:

- `fork()`: Create child process
- `exec()`: Replace process image
- `exit()`: Terminate process
- `wait()`: Wait for child
- `getpid()`: Get process ID

File I/O:

- `open()`: Open file
- `read()`: Read from file
- `write()`: Write to file
- `close()`: Close file
- `lseek()`: Move file pointer

Key Insight

These simple calls combine to build complex programs. UNIX philosophy: simple tools, powerful combinations.

Hands-on: Tracing System Calls

Use `strace` to see what syscalls a program makes:

```
$ strace ./hello
execve("./hello", ["./hello"], ...) = 0
brk(NULL)                           = 0x55a8c8d000
mmap(NULL, 8192, ...)               = 0x7f2a8c000000
...
write(1, "Hello, OS!\n", 11)        = 11
exit_group(0)                       = ?
```

What we learn:

- Every I/O operation is a system call
- Even simple programs make many syscalls
- The kernel does a lot of work behind the scenes

More strace Examples

Count syscalls by type:

```
$ strace -c ls
% time      calls   syscall
------   -------   ----------------
 25.00         10   read
 20.00          8   write
 15.00         12   openat
 10.00         12   close
  ...
```

Useful strace options:

- `-e trace=file`: Only file-related syscalls
- `-p PID`: Attach to running process
- `-t`: Show timestamps
- `-f`: Follow child processes

Key Takeaways

1. **OS Definition**: Resource manager + abstraction provider
2. **History**: From batch processing to cloud computing
3. **Kernel Designs**: Monolithic vs microkernel trade-offs
4. **Protection**: User mode vs kernel mode boundary
5. **System Calls**: The only way to access OS services

Core Principle

The OS creates the illusion that each program has the machine to itself, while safely sharing resources among all programs.

This Week's Tasks

- **Quiz 1**: Introduction concepts (due before next class)
- **Reading**: Textbook Chapters 1-2
- **Hands-on**: Try `strace` on various programs
- **Next Week**: Process Management
  - Process lifecycle
  - Process Control Block
  - fork() and exec()
  - Context switching

Questions?