COSC 315: Introduction to Operating Systems

History, Hardware Roots, and First Principles

1

Overview

- Why study OS history?
- Early computers and capabilities
- What breaks without an OS
- Core OS ideas

2

Computing Before Operating Systems

- Vacuum tube machines (1940s–50s)
- Room-sized, bespoke systems

Feature	ENIAC (1945)	UNIVAC I (1951)	IBM 701 (1952)
Program Storage	None (hard-wired)	Stored program	Stored program
Programming Method	Physical rewiring (plugboards, switches)	Machine code / early assembly	Assembly language
Reprogramming Time	Hours to days	Minutes	Minutes
Memory Type	Registers only (accumulators)	Mercury delay lines (sequential access)	CRT (Williams tubes, random access)
Input	Punch cards	Magnetic tape	Punch cards
Output	Punch cards	Magnetic tape	Tape, printer
Software Abstractions	None	Loaders, basic I/O routines	Subroutines, early monitors
Operating System	None	None (but OS need emerging)	Proto-OS concepts
Key Limitation	No software flexibility	Slow memory access	Hardware reliability
Historical Significance	Proved electronic computation	First practical stored-program system	Foundation of system software & OS ideas

3

Computing Before Operating Systems

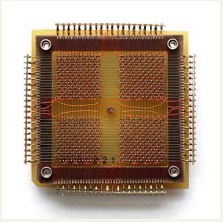
Problem / Capability	ENIAC	UNIVAC I	IBM 701
Program flexibility	None	Yes	Yes
Program reuse	No	Limited	Yes
Resource sharing	No	Minimal	Necessary
Execution control	Manual	Program-based	Software-assisted
Multiple programs in memory	No	No	No
Multiprogramming	No	No	No
Memory protection	Not applicable	None	None
Multi-user support	No	No	No

Early computers did not require operating systems until stored programs, increased software complexity, and shared hardware made manual control impractical.

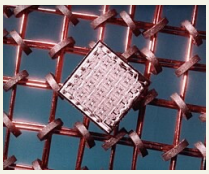
4

Core Memory

- Magnetic-core memory
  - Uses tiny **magnetized ferrite cores** arranged in a grid
  - Each core stores **one bit** (magnetized clockwise or counterclockwise)
  - Wires threaded through cores select and read/write individual bits
  - **Pros/Cons?**
  - Reliable random access
    - Any memory location can be accessed in **constant time**
    - No waiting for data to circulate (unlike delay lines or drums)
    - Random access makes *software structure* more important than *physical memory layout*.
  - Enabled larger, more complex programs
- Dynamic Random-access memory (DRAM) introduced in the the 1970's changed things
  - Initially around the same price as core, DRAM was smaller and simpler to use



https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/KL_CoreMemory.jpg/960px-KL_CoreMemory.jpg



https://upload.wikimedia.org/wikipedia/commons/thumb/c/c3/Electronic_Memory.jpg/500px-Electronic_Memory.jpg

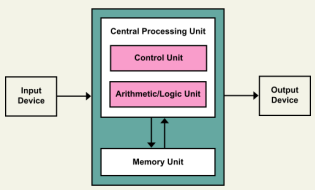
Even after magnetic-core memory was replaced by semiconductor memory, main memory was often still referred to as **core**

Question: What is a core dump?

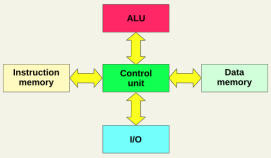
The Stored-Program Concept

- Early machines were ***configured***, not programmed
- Programs initially existed as:
  - Plugboards
  - Switch settings
  - Fixed wiring
- **Stored-program computers** place instructions *in memory*
- Code becomes:
  - Data
  - Loadable
  - Replaceable
  - Relocatable

Key: the Stored-Program Concept
von Neumann vs Harvard architecture



https://en.wikipedia.org/wiki/File:Von_Neumann_Architecture.svg



https://en.wikipedia.org/wiki/File:Harvard_architecture.svg

Life Without an OS

- **One program at a time**
  - **Early machines were single-job systems:** they executed *one program from start to finish before anything else could run*
- **Manual setup and execution**
  - For some early computers, programming literally meant plugging in cables, flipping switches, or loading physical media that determined the computation
  - Even after stored programs arrived, operators would still set up jobs manually by preparing input media and loading job decks
  - a human operator started and stopped executions rather than a control system.
- **No protection or scheduling**
  - Memory was fully accessible to the running program
  - there was no hardware or software to prevent a program from overwriting any part of memory because there was only one program resident at a time.
  - no risk of one program interfering with another → **protection was not needed**

CPU Utilization Problem

- **CPU idle during I/O**
  - Early computers spent time waiting for slow I/O devices (card readers, printer) to complete operations
  - I/O was orders of magnitude slower than CPU execution -> CPU often sat idle until that I/O finished
  - Idle time represented wasted potential
    - CPU could been doing useful computation while I/O completed if there were other work available
- **Humans slower than machines**
  - Operators manually sequenced work, loaded programs, managed I/O media, and initiated runs
  - Pace of human setup and job transitions was much slower than machine execution
  - **Expensive CPU time was under-utilized**
- **Wasted computation**
  - Inefficiency of CPU waiting led to two major operating system developments:
    - **Batch scheduling** and monitor programs to automate job sequencing
    - **Multiprogramming:** *Something keeps* multiple jobs in memory and switches the CPU to another job when one is blocked, reducing idle time

Proto-Operating Systems: Resident Monitors & Batch Systems

- Early response to growing system complexity was the Resident Monitor
  - Small control program permanently resident in memory
  - Primary responsibilities
    - Load the next job into memory
    - Initialize execution and transfer control
    - Handle basic I/O and job termination
  - Batch processing model
    - Jobs prepared offline on cards or tape
      - Submitted and executed sequentially
  - Automation of operator tasks, replaced manual setup, sequencing, and job control
- Reduced idle CPU time between jobs

9

The BIG problems

Because the CPU was idle waiting for slow I/O and human setup, early systems wasted significant computation

A core motivation for OS features like multiprogramming and scheduling BUT…

1. **Need for Memory Management**
2. **Scheduling Complexity**
3. **Starvation**
4. **Priority Inversion (high priority waiting for low)**
5. **Context-Switching Overhead**
6. **Resource Contention and Deadlock**
7. **Privilege levels**
8. **Fairness vs. Efficiency Trade-offs**

The operating system exists to coordinate shared hardware safely and efficiently.

10

Memory Problems

- Programs overwrite memory
  - In early systems all programs shared the same flat memory space (**no isolation**)
  - There was nothing to stop a program from writing anywhere in memory (**crashes halt entire system**)
  - A buggy pointer, buffer overflow, or incorrect memory access could overwrite another program's code or data
  - Need Isolation and protection so modern OSes use hardware support and virtual memory to prevent them
  - Separates address spaces so one program's writes cannot corrupt another's memory.
- Lack of isolation + unchecked memory access leads to unstable systems
- Modern operating systems use memory protection and **user** vs. **kernel** mode to contain faults
  - If a program violates access rules, the OS can terminate that program rather than crashing the entire system

11

Reloading and Context Switching

- Early CPU Behavior
  - Processors could execute only one task at a time on a uniprocessor system
  - Meant jobs ran sequentially rather than concurrently
- Before modern OS loaders, transitioning between jobs required manual or simple loader intervention
  - Replaced the running program with another, effectively restarting execution from the beginning of the next job
- In a multiprogramming OS with a single core, a scheduler can switch the CPU from one process to another by saving the current process state (its context) and loading the next process's state.
  - Allows multiple processes to share the CPU over time, giving the illusion of concurrent execution even though the CPU runs one at a time
  - Overhead? Challenges?

Context switching enables true multitasking where the OS can pause one task and resume it later without restarting it, improving CPU utilization and responsiveness

12

Reloading Code

- Manual loading
  - Required an operator to manually load programs into memory and start execution
- Static memory layout
  - Programs were built to run at **fixed memory addresses**, meaning the code and data were placed at predetermined locations in memory and could not be easily relocated at load time
- Restart from beginning
  - if a program needed to be rerun or recovered from an error, it had to be **restarted from the very beginning** of its execution

Single CPU: Multiple Tasks, Scheduling and the Illusion of Concurrency

- Single CPU Reality
  - A single CPU can execute **only one task at any given moment**;
  - it cannot truly run multiple tasks simultaneously.
- Multiprogramming & Context Switching
  - Operating systems divide CPU time into **small slices** and rapidly switch between tasks.
  - Each switch saves the state of the current task and loads the state of another (context switching).
- Illusion of Concurrency
  - Because these switches happen so quickly, users perceive that multiple tasks are running at the same time
- Scheduling (Who gets to run?)
  - Mechanism the OS uses to assign CPU time to processes that are ready to run
  - Decides which process should use the CPU next and for how long

This illusion improves responsiveness and utilization, enabling modern multitasking even on single-core systems

Privilege Levels

- With multi-user/multi-program, access needs to be restricted
- Kernel mode (the one ring…)
  - **Privileged mode** with **unrestricted access to all hardware and memory**
  - Only trusted code (the kernel) can execute privileged instructions directly
- User mode
  - Regular applications run in **restricted mode** with limited access to system resources
  - **Cannot directly access hardware or critical memory** and must request services via system calls
- Hardware enforcement
  - CPU maintains a **mode bit (privilege level)**
  - (Should) Prevents privileged instructions or direct hardware access in user mode
    - Fault occurs if user code attempts restricted operations, transferring control safely to the OS
    - Example?

Separation between user and kernel modes is critical so that only trusted OS code can perform critical operations, improving system security and stability

15

The Operating System: an Illusion Machine

- Hardware is limited:
  - One CPU
  - Shared memory
  - Shared devices
- The OS creates useful illusions:
  - Illusion of concurrency → each process appears to run simultaneously
  - Illusion of private memory → each process appears to own its own address space
  - Illusion of ownership → processes appear to control devices and resources
- These illusions are enforced by:
  - Scheduling
  - Memory protection
  - Privilege levels

An operating system is a carefully engineered set of lies that make shared hardware usable

16

Interrupts

- Hardware regains control
  - CPU normally runs user code until **hardware signals an event** that needs immediate attention
  - CPU **saves its state and transfers control to an interrupt handler**, allowing the OS to regain control without polling
- Timer interrupts
  - Periodic hardware-generated interrupt triggered by a programmable timer at regular intervals
  - **Pre-empts the running process** to perform bookkeeping, and make scheduling decisions (needed for preemptive multitasking)
- Device interrupts
  - Hardware signals from I/O devices (e.g., disk, keyboard) indicating an event or completion of data transfer.
  - OS stops the current task and runs an **interrupt service routine (ISR)** to handle the event, enabling efficient event-driven communication

Interrupts enable the OS to regain control from running for scheduling and device signals for responsive I/O.

17

Hardware Abstraction

- Hide device details
  - Provides a **hardware abstraction layer (HAL)**
  - Hides the low-level specifics of devices (e.g., differing disk, keyboard, or network hardware), so applications and users don't need to know how each device works internally
  - Simplifies development and usage because complex hardware interactions are encapsulated behind uniform interfaces
- Portability
  - Allows for **standardized OS interfaces**
    - Software can be written once and run on **different hardware platforms with little or no modification** (hopefully)
- Simplified interfaces
  - Expose high-level, simplified interfaces (like file reads/writes or network APIs) instead of exposing raw hardware commands
  - Enables programmers to build complex applications without managing device specifics, thereby reducing complexity and potential errors

18

Multi-User Support

- **Multi-user operating systems**
  - Allows **multiple users to interact with the same computer at the same time** often from different terminals
  - OS managing resources and isolation among users
- Grew out of **time-sharing systems** developed in the 1960s (e.g., Compatible Time-Sharing System at MIT)
  - Rapid CPU scheduling gave each logged-in user a small time slice so the system *felt* dedicated even though the machine was shared

In a multi-user OS, the scheduler, memory protection, and access controls work together so that each user's processes run safely and without interfering with others, making interactive use by several people practical and efficient.

Starvation, Contention, Deadlock, and Fairness

- Operating systems manage shared resources
  - CPU time
  - Memory
  - I/O devices
- Sharing introduces unavoidable problems:
  - Starvation → a process may never get the resources it needs
  - Resource contention → multiple processes compete for limited resources
  - Deadlock → processes wait indefinitely on each other
  - Fairness vs. efficiency → maximizing throughput can disadvantage some tasks

These are not bugs; they are design trade-offs in the design of a system

Once you allow many things to run at once, deciding who gets what becomes the real problem

Why OSes Exist…

- Early computers were powerful but unmanaged
- Stored programs and shared hardware increased complexity
- Manual control did not scale
- Operating systems emerged to:
  - Coordinate execution
  - Manage memory
  - Control devices
  - Handle failure

Core OS mechanisms evolved in response to real constraints

21

Key Takeaways

- The OS is a coordination layer, not just software
- OSes create useful illusions:
  - Concurrency on a single CPU
  - Private memory on shared hardware
  - Safe ownership of resources
- These illusions require:
  - Scheduling
  - Memory protection
  - Privilege separation
- Sharing resources introduces unavoidable trade-offs:
  - Starvation
  - Contention
  - Deadlock
  - Fairness vs. efficiency

An operating system does not eliminate complexity, but it manages it, hides it, and decides who pays for it.

22