What is an OS?

CS 450: Operating Systems
Michael Lee <lee@iit.edu>
Agenda

- Road to the modern OS
- OS responsibilities
- OS privileges
- OS organization
- Summary
Road to the modern OS
1950s: Batch processing

- A program is completely defined by a “batch” of punchcards
- Batches are manually fed into mainframes, which execute a single batch at a time (a “job”)
- Programmer defines any and all routines needed for the job
  - E.g., for controlling and accessing specific I/O devices
1950s-1960s: Support libraries

- Useful, reusable routines (e.g., for math, I/O) distributed as collections of punchcards
- These routines can be “linked” (manually) into programs without much modification
- First support libraries = the original OSes
  - No standardization!
1960s: Automatic batch processing

- To keep up with faster processors, reading and starting/transitioning between jobs require automation
- “Monitor” programs also keep track of usage, resources expended, etc.
- Grew to become runtime libraries that automatically manage the execution of multiple batches of jobs (in sequence)
Pros/Cons of Batch processing

- Pros
  - Full use of hardware
  - No worrying about other jobs during execution

- Cons
  - No interactivity
  - No live debugging
  - No feedback loop
  - Poor hardware utilization
  - Do everything yourself!
1970s: Rise of Timesharing

- To let many users share a computer *concurrently*, software is needed to *automatically save/restore context* between jobs

- Resources (e.g., CPU & memory) are *virtualized*

- Jobs are *isolated* and *protected* from each other

- Resulted in a lot of system overhead, but offset by interactivity and improved hardware utilization

- Development and availability of UNIX on mainframes and “minicomputers”
1980s: Era of (some) bad ideas

- Personal computers (microcomputers) become widely available
- Underpowered compared to systems that ran contemporary timesharing OSes such as UNIX
- PC OSes (e.g., MS-DOS, Mac OS) were dumbed down in many ways
  - Lack of memory protection
  - Cooperative multitasking vs. preemptive multitasking
  - Poor system stability, and chaos for developers!
1990s-Present: Modern OSes

- More powerful PCs make *preemptively multitasked* OSes generally viable
- High degrees of *virtualization*, *isolation*, and *concurrency*
- Exploding market for varied I/O devices and peripherals
  - OS support for “plug and play” third-party device drivers
- Large, sophisticated system call interfaces
  - Standards are created for portability across OSes (e.g., POSIX)
OS responsibilities
operating system

noun
the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.

New Oxford American Dictionary
Breaking down the definition

- “tasks and applications” = running programs
  = processes
- instructions & data stored in memory; executed and fetched on the CPU
- “peripherals” = I/O devices (hardware)
- OS raison d’être: facilitate process execution and access to hardware
Resource management

- CPU, Memory, I/O devices are *limited resources*

  - i.e., possible for num processes > num CPU cores, total memory required > physical RAM, file accesses > disk read/write heads

- OS acts as a high level *resource manager*

  - So processes can focus on their own tasks, the OS ideally manages and allocates resources in an unobtrusive, transparent way
Virtualization

- A powerful model for resource allocation is \textit{virtualization}:
  - Each process behaves as though it is accessing its own private CPU(s), address space, I/O device, etc.
  - Behind the scenes, the OS maintains this illusion by allocating and multiplexing resources across all concurrently executing processes
  - Effectively creates an idealized machine for each process
Concurrency

- Concurrency presents its own hurdles and techniques for dealing with them.
- Concurrent processes must be protected/isolated from each other.
- Nondeterministic execution and requests/access to resources creates *race conditions* within the OS and between processes.
- Dealing with these issues requires special tools and techniques.
Persistence

- CPU and Memory state are volatile
- I/O devices provide support for persistent storage
  - Presents a host of new issues:
    - How to namespace persistent data?
    - What APIs are needed for accessing persistent data?
    - How to efficiently manage and access data on slow HDDs?
    - If processes crash when updating persistent store, how to guarantee consistency?
How to achieve these?

- To implement virtualization, concurrency, persistence (and other goals), the OS relies on hardware assistance
- All modern ISAs have built-in mechanisms to support OS tasks
- Of paramount importance: hardware features that allow the OS to maintain exclusive access to privileged operations and structures
- To prevent accidental/malicious process behavior from interfering with other processes or the OS itself — i.e., ensuring robustness & isolation
OS privileges
Can we do this without HW?

- I.e., can you write a program (the OS) to execute other (user) programs, and guarantee isolation and robustness without hardware support?

- Consider some common (local) security vulnerabilities:
  - address fabrication
  - code injection
  - return-oriented programming
Software mitigation

- Software mechanisms:
  - Static verification (e.g., type-checking)
  - Run-time tools (e.g., garbage collection, exception handling, VM)
- Very hard to guard against all security vulnerabilities in software alone!
- Basic issue: once untrusted/dangerous code starts running on the processor, how can we prevent it from doing whatever it wants?
Hardware support

- All modern CPUs support, at minimum, two “modes” of operation
  - Privileged/Supervisor mode: all features accessible
    - Including special operations and access to I/O devices, control registers, and all of memory
  - User mode: only “safe” operations and process-local data accessible
- System boots to OS in privileged mode, which runs processes in user mode
  - Mode switches must be rigorously enforced!
Mode transitions

- A common mechanism for switching between privileged & user modes is the *interrupt* (either software or hardware triggered)

- E.g., system call / trap (starting in user mode):
  1. Process executes special `int` instruction with interrupt # as argument
  2. Hardware looks up associated OS entry point from interrupt table
     - Interrupt table is managed by OS
  3. Hardware switches to privileged mode before running OS handler

- Implements a hardware-assisted *application binary interface* (ABI)
OS exists to serve

- OS carries out privileged operations on behalf of user processes
- Also keeps up the illusion of various abstractions that simply process execution (e.g., concurrency and non-overlapping address spaces)
- Important: privileged/supervisor mode is only for the OS!
- Administrator / “root” user do not run processes in privileged mode
- All processes run in user mode!
- But does all of the OS?
OS organization
What is privileged?

- Which portions/modules of an OS will be run in “privileged” mode?
- “Standard OS modules”:
  - virtual memory
  - scheduler
  - device drivers
  - file system
  - IPC
The “Kernel”

- Privileged modules constitute the “kernel” of the operating system
  - First program loaded into memory, and always memory-resident
  - Handles all privileged operations
    - Hardware access
    - Updating special/control registers
    - Running special instructions
  - Works in close concert with architecture features (e.g., clock interrupt)
Monolithic architecture

- *All primary modules and I/O device drivers* run in privileged mode
- Relatively large, permanent memory footprint
- No mode transitions when jumping between different pieces of the OS
  - Very little system overhead
- Because the privileged codebase is very large, harder to verify and guarantee system robustness!
  - If one piece of the OS crashes, all of it does
Microkernel architecture

- Only essential services are privileged; everything else runs in user mode
- Relatively small memory footprint
- Microkernel functions in part as a messenger between different modules running in user mode
  - Jumping between different OS modules may require mode switch
  - Higher system overhead (though clever optimizations exist)
- Easier to verify and guarantee robustness
  - If a user-level OS module crashes, just restart it
... suffice it to say that among the people who actually design operating systems, the debate is essentially over. **Microkernels have won**

- Andrew Tanenbaum
  (noted OS researcher)
The whole “microkernels are simpler” argument is just bull, and it is clearly shown to be bull by the fact that whenever you compare the speed of development of a microkernel and a traditional kernel, the traditional kernel wins. By a huge amount, too.

- Linus Torvalds
  (chief architect, Linux)
Beyond the debate

- Yet another route: why not just implement OS as a low-level library?
  - Loss of isolation, but big efficiency gain
  - Used by many embedded systems
- And what about hosting multiple OSes on a single machine?
  - Useful/feasible on modern multi-core machines
  - Hypervisors provide low-level virtual machines to guest OSes
  - Yet another layer of isolation!
Summary

- Why do we need an OS?
  - To facilitate process execution and simplify/control access to hardware

- What does an OS do?
  - Provide virtualization, concurrency, and persistence

- How is an OS organized?
  - Separation of kernel (privileged) and user modules — architecture of kernel is an exercise in tradeoffs!