What is an OS?

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

- Why do we need an OS?
- What does an OS do?
- What are some secondary concerns?
- How is an OS organized?
§ Why do we need an OS?
Discussion: What would using/programming a computer be like *without* an OS?

e.g., consider a simple “hello world” program
consider a more complex program (requiring a compiler)
consider a situation where we need multiple programs
consider a situation where we must perform complex I/O
(really, really, really, really, really, really, really painful)
§ What does an OS do?
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*
tasks & applications = running programs
= Processes

peripherals = I/O devices (hardware)
Process = *Program in execution*

- instructions & data stored in **Memory**
- fetched and executed on the **CPU**
OS raison d’être: run processes and access hardware
- processes — i.e., concurrent execution of \( \geq 1 \) program
- goal is to make this seamless, efficient, and robust!
Central problem: CPU, Memory, I/O devices are limited resources
i.e., possible for num processes > num execution cores,
total memory required > physical RAM,
file accesses > disk read/write heads
For CPU and Memory, OS solution is to *virtualize* them

- each process behaves as though it is accessing its own private CPU(s) and memory address space
- behind the scenes, OS *allocates* and *multiplexes* CPU time and memory across all processes using available hardware
- pros/cons?
For I/O devices, OS may mediate access via abstract interfaces rather than provide virtualization

- simplify and secure I/O operations
- e.g., uniform API for reading/writing
  memory mapped access to “block” devices
I/O devices also enable *persistence* (i.e., non-volatile storage)
- filesystems serve as namespaces for storing and accessing data
- manipulating persistent data from dynamic, volatile processes can be tricky! (e.g., what happens if crash during write?)
- requires mechanisms to guarantee robustness
primary OS responsibilities:
- virtualization
- concurrency
- persistence
§ What are some secondary concerns?
Thanks to virtualization and concurrency, we can have many processes running at the same time while sharing adjacent system resources.

- but now they can potentially mess with each other!

- processes must be protected from each other; and isolated from the rest of the system

- security is a related concern
How to enforce protection/isolation?
Two routes: software / hardware
Is isolation possible solely via software?

I.e., can you write a program (the OS) to execute other (user) programs, and guarantee separation & robustness without hardware support?
Some software attack vectors:

- address fabrication (e.g., integer-to-address cast for cross-space pointers)
- buffer overruns (e.g., on syscalls)
- run-time errors (e.g., intentional/accidental stack overflows)
Software mechanisms:

- static verification (e.g., type-checking) — programs must “pass” to be run
- run-time tools (e.g., garbage collection, exception handling)
Is isolation possible solely via software?

- *maybe* — but difficult/impractical

- the popular approach (almost all OSes) is to rely on hardware support
Common approach: “rings” of protection
- e.g., on x86, indicated by current privilege level (CPL) flag

most to least privileged
CPL=3 → “user” mode
CPL=0 → “supervisor/kernel” mode
- access to special instructions & hardware
How to modify CPL?

Q: Ok to allow user to directly modify CPL before calling OS?

A: No! User can set CPL=0 and run arbitrary code.
Q: What about combining CPL “set” instruction with “jump” instruction to force instruction pointer change?
A: Bad! User can still set CPL=0 and jump to non-OS code.
Q: What about combining CPL “set” instruction with “jump” instruction that must target OS codespace?

A: Not safe. User code may jump to delicate location in OS.
Solution on x86: `int` instruction

- sets CPL=0
- loads pre-defined OS entry point from *interrupt descriptor table*
- IDT base address can only be set when CPL=0 (by *privileged lidi_t instr*) — this happens during the boot process

(more on this later when we go over interrupts/syscalls in detail!)
Tight integration of software & hardware permits safe, controlled jumps between running user/OS code
  - example of an application binary interface (“ABI”)
  - contrast with “API”
So … isolation is possible, but what code is run in user mode? what code is run in kernel mode?

- longstanding debate in OS design/organization
§ How is an OS organized?
i.e., what are the *top-level modules* of an OS, and which must run in privileged/kernel mode?
“Standard” OS modules:
- virtual memory
- scheduler
- device drivers
- file system
- IPC
privileged modules constitute the “core” of the operating system; i.e. the OS kernel
traditional approach: all are privileged
- i.e., entire “OS” runs in kernel mode
  - known as monolithic kernel
- pros/cons?
alternative approach: *minimum* privileged

- i.e., have a “*microkernel*” with minimal set of privileged services
  - everything else runs in user mode
    - microkernel relays requests
  - pros/cons?
Monolithic Kernel based Operating System

Microkernel based Operating System

Application

System Call

VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware

Application
UNIX Server
Device Driver
File Server

Basic IPC, Virtual Memory, Scheduling
Hardware

courtesy of Wikimedia Commons
… 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)
Yet another route: why not just implement OS as a low-level library?

- loss of isolation, but big efficiency gain (and flexibility in using h.w. directly)

- used by many embedded systems
And finally, what about hosting multiple OSes on a single machine? (Useful/feasible on large, 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
Summary

- What are some secondary concerns?
  - Protection, Isolation, Security — implemented through a combination of software/hardware mechanisms

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