Overview of the OS

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

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Agenda

- what is an operating system?
  - what are its main responsibilities?
    - how does it achieve them?
- how is an operating system organized?
  - what is an operating system *kernel*?

§What is an OS?

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**

OS duties revolve around aiding and abetting user processes

- setting up a consistent view of system (e.g., virtual memory)
- simplifying access to disparate devices (e.g., open/close/read/write API)

Problem: there's never enough hardware to go around

- OS *multiplexes* hardware (time/space)
- must also *isolate* processes from each other (and the OS itself)

primary OS services:

isolation, *h.w. abstraction* and *concurrency*

(and another, arising from first: *interaction*)

How to enforce 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 prevention mechanisms:

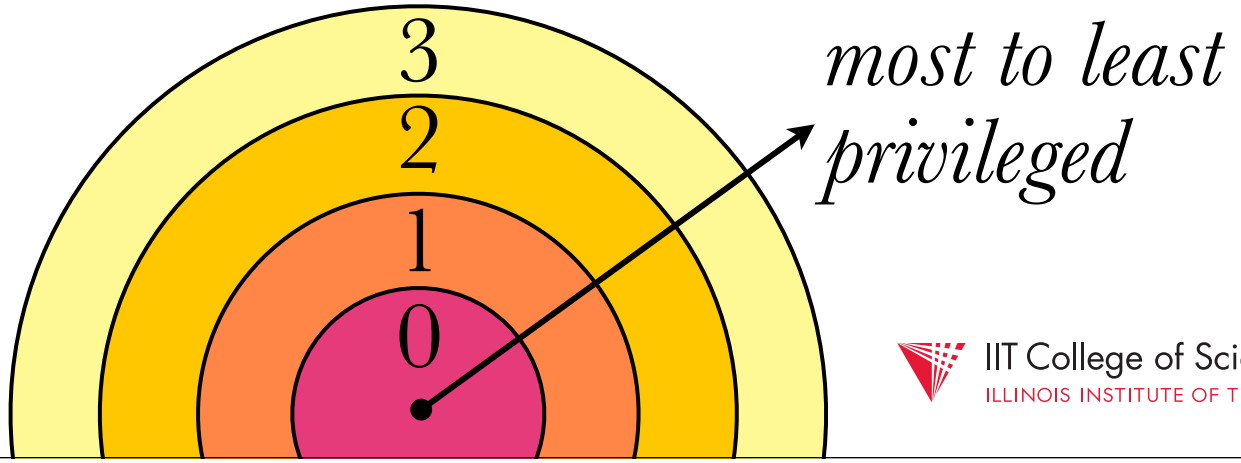
- static verification (e.g., type-checking) — programs must "pass" to be run
- run-time tools (e.g., garbage collection, exception handling)

Is isolation possible solely via software?

- maybe — but difficult/impractical
- the popular approach (all commercial OSes) is to rely on hardware support

e.g., Intel x86 architecture provides a 2-bit *current privilege level* (CPL) flag

- implements 4 *protection ring* levels



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 invoking OS?

A: No! User can set CPL=0 and run arbitrary code before calling OS

Q: What about combining CPL "set" instruction with "jump" instruction to force instruction pointer (`eip`) change?

A: Bad! User can set CPL=0 and jump to user code to masquerade as OS.

Q: What about combining CPL “set” instruction with “jump” instruction that must target OS codespace?

A: Not good enough. User code may jump to delicate location in OS.

Solution: x86 provides `int` instruction:

- sets CPL=0
- loads a pre-defined OS entry point from *interrupt descriptor table* (IDT)
- IDT base address can only be set when CPL=0 (by privileged `lidt` instr)

Privileged instruction & hardware access prevented, but how is memory protected?

- Each segment/page of memory in x86 is associated with a minimum CPL
- Only permit current process to access its own segments/pages

Finally, how can OS regain control from unruly user process? (E.g., running in tight loop, never executing `int`)

- hardware sends periodic *clock interrupt*
  - *preempts* user; summons OS

Isolation accomplished.

How to achieve *h.w. abstraction* & *concurrency*?

h.w. abstraction = user traps to OS (via `int`) with service request; OS carries out task and returns result — "syscall"

i.e., hardware (e.g., NIC) is exposed as a software stack (e.g., TCP/IP)

concurrency = clock interrupt drives *context switches* and *hardware multiplexing*, carried out by OS scheduler (and others)

enables *multitasking* on limited hardware (compare to *parallelism*)

Different approaches to multitasking:

- *cooperative*: processes voluntarily control
- *preemptive*: OS periodically interrupts
- *real-time*: more stringent requirements

§How is an OS *organized*?

i.e., what are the *top-level modules* of an OS, and which must run in privileged mode (e.g., CPL=0)?

some modules:

- virtual memory
- scheduler
- device drivers
- file system
- IPC

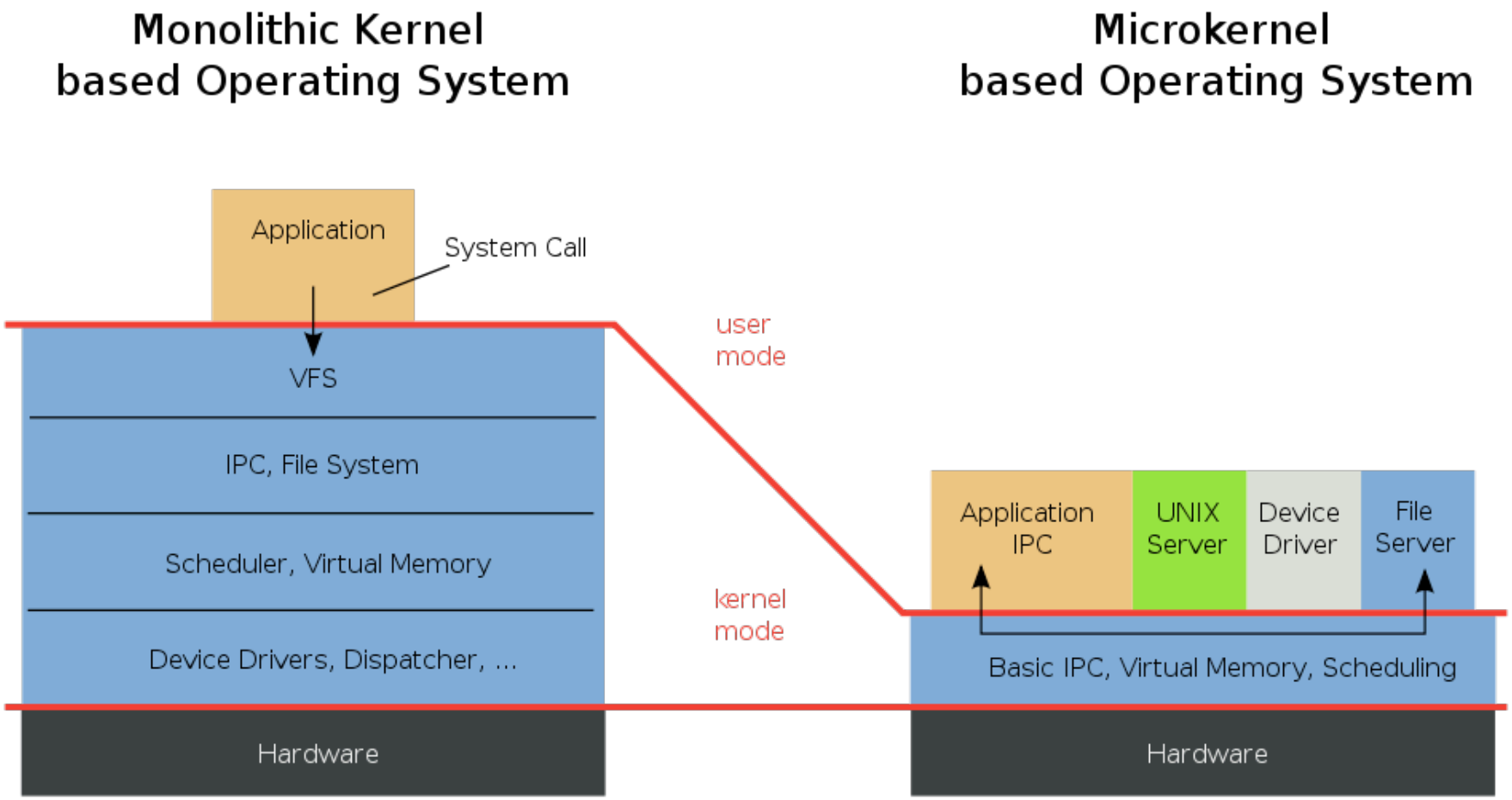
privileged modules constitute the "core" of the operating system; i.e. the OS *kernel*

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?



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)

your opinion?

→ assignment 1 (paper)

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!