CPSC313: Computer Hardware and Operating Systems

Unit 5: Process Isolation & Virtual Memory

(5.2) Transferring Control to the Operating System

Today

- Learning Outcomes
  - Explain how the operating system is “special” relative to regular applications.
  - Explain what supervisor mode and privilege mean.
  - Explain how and when the operating system gets to run.
  - Define:
    - Trap
    - Exception
    - Interrupt
- Reading:
  - 8.1-8.3

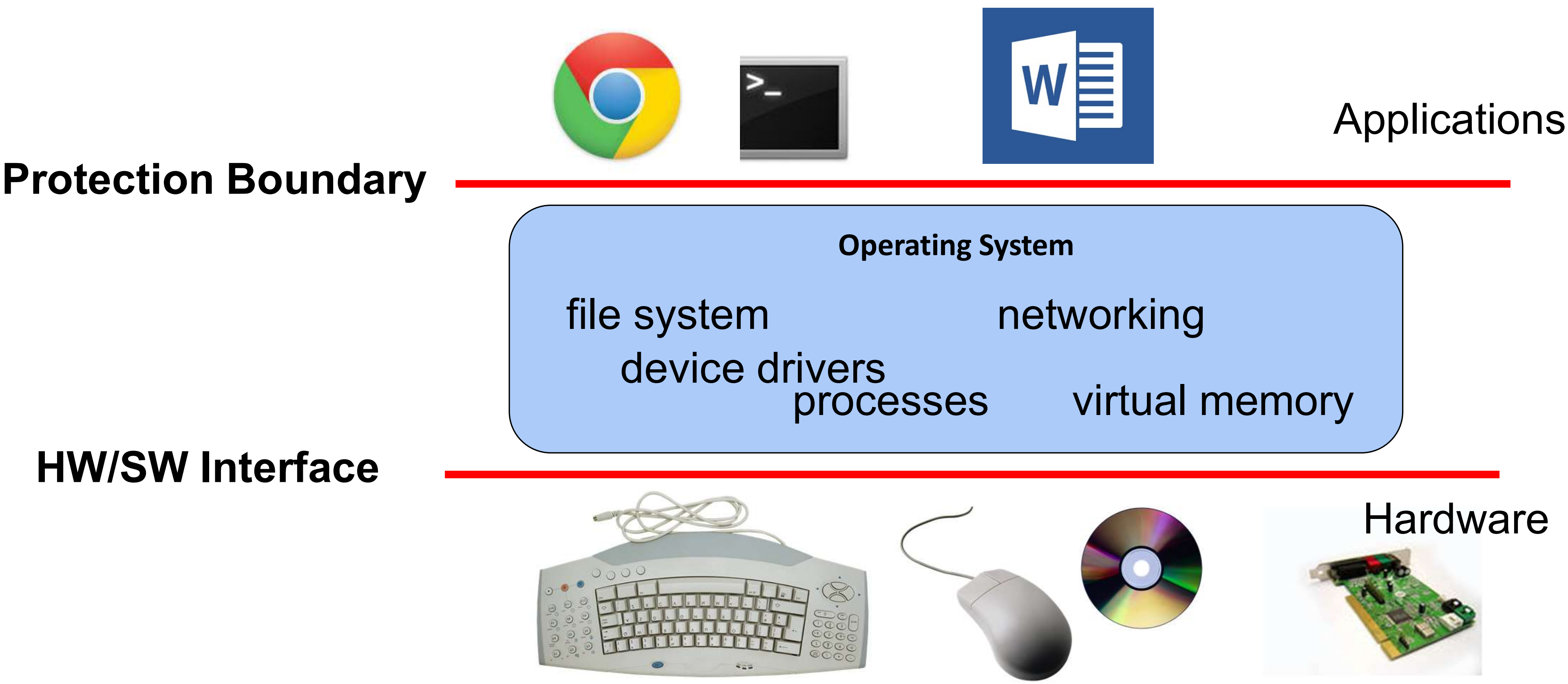
Admin

- Quiz 4 viewings/retakes this week
- Quiz 5 (last quiz!) next week; no retake, and so no reserved viewings
- Lab 10 due Sunday (last lab!)

Recall: VM: A Hardware/Software Partnership

- We need hardware support to provide virtual memory. Why?
    - We need virtual-to-physical address translation.
      Software (specifically, the OS) is too slow. So, we need the HW.
    - Questions you should be asking:
        - Why does it have to be the OS?
        - Why would that be slow?
- Software
    - Sets up the mappings that the hardware will execute
    - Manages the allocation of physical memory
    - Implements policies about how memory is shared.

What is special about the operating system?



Protection Boundaries

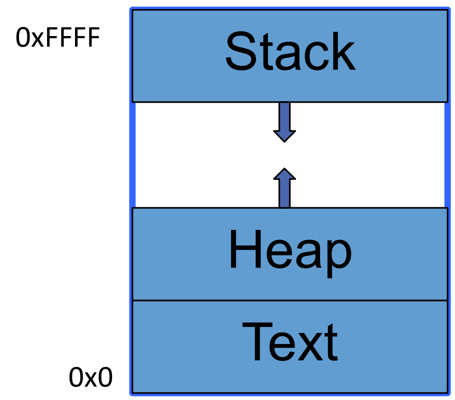
- Modern hardware has multiple privilege levels or modes.
- Different software can run with different privilege.
- Processors typically provide two or more different modes of operation:
    - User mode: how all “regular programs” run.
    - **Kernel mode or supervisor mode: how the OS runs.**

    (Mostly two; x86 has four; some older machines had eight!)
- The mode in which a piece of software is running determines:
    - What instructions may be executed.
    - How addresses are translated.
    - What memory locations may be accessed (enforced through translation).

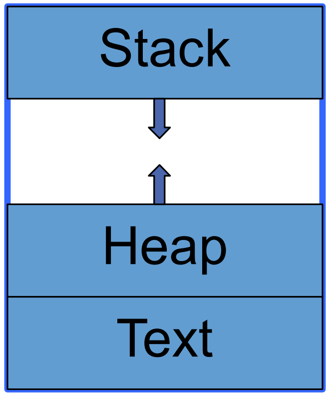
What are some things the *kernel* should be allowed to do that the *user* cannot?

Constraining the Mapping

- We want the OS to be allowed to do things that normal processes cannot: interact with devices, read/write any process’s memory, etc.
    - The OS should have access to some things that are inaccessible to regular processes.
    - Our mapping information has to distinguish between mapping user processes and the OS.
- Different parts of an address space support different operations:
    - Read-only text/data: cannot be modified
        - Mappings should disallow writes to some parts of memory.
    - Data should not be executed:
        - Mappings should disallow execution of some parts of memory.



Putting it all Together: The Hardware



- We ask the hardware to map a triple of:
  - Virtual address
  - Type of access (read, write, execute)
  - Privilege level

  to a physical address
- The hardware will either:
  - Produce a physical address
  - Fault (which is not necessarily "bad"!) due to:
    - The virtual address is not valid (i.e., there is no mapping for it).
    - The type of access is not allowed to the memory requested (e.g., writing to read-only memory).
    - The process requesting access does not have the appropriate privilege level to access the memory (e.g., a user process wants to read memory that requires supervisor mode privilege).
    - The process is allowed to access the address, but *the OS needs this fault to force it to find it first*.
- When the hardware faults: Software (the OS) takes over.

Example: MIPS

- Has two privilege modes:
  - User mode:
    - access to CPU/FPU (floating point unit) registers
    - access to *flat, uniform virtual memory address space*
  - Kernel mode:
    - access to everything accessible in user mode
    - access to memory mapping hardware and special registers
    - can issue privileged instructions

Example: Intel x86

- Four protection levels
  - Ring 0: Most privileged: OS (usually) runs here
  - Rings 1 & 2: often ignored; can run less privileged code (e.g., third party device drivers); use of these rings/levels changes in a virtualized environment
  - Ring 3: Application code
- Memory is described in chunks called *segments*
  - Each segment also has a privilege level (0 through 3)
  - Processor maintains a "current protection level" (CPL) - usually the protection level of the segment containing the currently executing instruction
  - Program can read/write data in segments of *less (or equal) privilege* than CPL
    - Less privileged CPL means higher protection level (and so less access)
  - Program cannot *directly* call code in segments with more privilege

So, how do we change the privilege level?

- We must answer two fundamental questions:
  - **How** do we transfer control between applications and the kernel?
  - **When** do we transfer control between applications and the kernel?
- Proposed mechanism: Two new instructions: *raise* privilege level (or leave it at kernel) and *lower* it (or leave it at user).
  - Who do we let run the “raise” instruction? User? Supervisor? Discuss!
  - Who do we let run the “lower” instruction? Discuss!

So, how do we change the privilege level?

- We must answer two fundamental questions:
  - **How** do we transfer control between applications and the kernel?
  - **When** do we transfer control between applications and the kernel?
- How: To transfer control from less privileged to more privileged code, we use a *trap*.
  - How does the trap get around the issue from the previous page to raise privilege? The short version: Carefully specifying what code gets to execute as a result of the trap. The long version: In the next pre-class video!

Kinds of traps

- *System calls*: An application asks the OS to act on its behalf.
- *Exceptions*: An application unintentionally does something requiring OS assistance (e.g., divides by 0, reads a page not in memory).
  - This does not *necessarily* indicate an error.
  - This is **not** your programming language's "throw/catch" exception!
- *Interrupts*: An asynchronous event (e.g., I/O completion).

How do system calls and exceptions differ from interrupts?

So, *when* do we change the privilege level?

- *System calls*: An application asks the OS to act on its behalf.
- *Exceptions*: An application unintentionally does something requiring OS assistance (e.g., divides by 0, reads a page not in memory).
  - This does not *necessarily* indicate an error.
  - This is **not** your programming language's "throw/catch" exception!
- *Interrupts*: An asynchronous event (e.g., I/O completion).

When a Process Does Something

So, *when* do we change the privilege level?

- *System calls*: An application asks the OS to act on its behalf.
- *Exceptions*: An application unintentionally does something requiring OS assistance (e.g., divides by 0, reads a page not in memory).
  - This does not *necessarily* indicate an error.
  - This is **not** your programming language's "throw/catch" exception!
- *Interrupts*: An asynchronous event (e.g., I/O completion).

When a device does something (independently)!

But what if a process never does one of those things?

- *System calls*: An application asks the OS to act on its behalf.
- *Exceptions*: An application unintentionally does something requiring OS assistance (e.g., divides by 0, reads a page not in memory).

We don't want to "starve" the OS of time.
Why not? Why might this be a problem?

The OS must guarantee that it eventually runs

- Processors have timers.
- Timers have the property that they either count up to some value or down to zero and then …

… generate an interrupt!

- To ensure it gets to run, the OS schedules a timer interrupt;
  if nothing *else* causes the OS to run before the timer interrupt,
  the OS knows it will get to run when the timer expires.

How long do we make the timer?

- Selecting a timer interval is part of the scheduling problem.
- An OS must make many decisions around scheduling:
  - Which process should get to run first?
  - What policy should it use to share the processor among multiple processes?
  - How long should a process run?
- The timer is a mechanism to transfer control to the OS
  so it can implement a policy.
  - We need a timer interval: *short* enough that the system is responsive;
    long enough to keep the fraction of our time handling timer interrupts small.
  - This interval is typically called a quantum.

Recap

- The OS is just a bunch of code sitting around waiting for something to do (e.g., help a user process, respond to a HW device, process a timer interrupt, etc.)
  - It sits around *in your process's address space*, but access to it is privileged
- The OS runs in privileged (or kernel, or supervisor) mode.
- HW provides a mechanism to transfer control from one privilege level to another.
- We call a mechanism that transfers control into the OS a trap.
- There are different kinds of traps:
  - System calls: explicit requests from a program to the OS (synchronous with respect to the program)
  - Exceptions (software interrupts; synchronous with respect to the program)
  - Interrupts (caused by hardware; asynchronous)

In-class activity – interacting with processes