Efficiency: caching (continued); Multiprocessing: operating systems; limited direct execution; system calls

COSC 208, Introduction to Computer Systems, 2021-11-03

Announcements

- Project 2 Part B due date extended to Tues, Nov 9

Outline

- Warm-up
- Cache replacement
- OS overview
- Accessing hardware
- Limited direct execution
- System calls

Warm-up

- Q1: *Where are caches used in computer systems?*
    - Operating systems
    - Web browsers
    - Web servers
    - Domain Name System (DNS)
    - Databases
    - Central Processing Units (CPUs)
    - Graphics Processing Units (GPUs)
    - Hard Disk Drives (HDDs)
    - Solid State Drives (SSDs)

Operating systems (OS) overview

- Purpose of an OS
    - Make computer hardware easy to use—e.g., an OS knows how to load an application's executable code from persistent storage (e.g., solid state drive (SSD)) into main memory, initialize the process's address space (code, heap, stack), and make the CPU execute the application's instructions
    - Support multiprocessing—i.e., running multiple applications concurrently
        - Concurrently == switch between multiple tasks during a window of time—e.g., alternate between cooking and setting the table
        - Simultaneously == complete two tasks at the same time—e.g., listen to a podcast while cooking
    - Allocate and manage hardware resources—e.g., decide when/which applications can use the CPU, decide when/which memory applications can use, prevent applications from stealing/accessing another application's CPU time or memory
    - Many OSes also provide a user interface (UI)
- How does the OS fulfill its duties?
    - Mechanisms — fundamental approaches for managing/providing access to hardware resources
        - E.g., system calls, process abstraction
    - Policy — specific ways of employing an approach; different policies make different trade-offs (in terms of efficiency, performance, etc.)
        - E.g., process scheduler

Accessing hardware

- OS is responsible for allocating/managing the hardware
  - ⇒ applications should **not have unfettered access to hardware**
- *How should applications access the hardware?*
  - Ask the OS for access to the hardware
    - How do we ensure the OS does not "lose control" of the hardware?
  - Asks the OS to perform an action on the application's behalf
    - How do we ensure this doesn't substantially degrade performance?
- Example: execute an instruction on the CPU
  - Asking the OS to do this on behalf of an application is impractical—OS would need to execute multiple assembly instructions for each assembly instruction the application wants to execute
  - Alternative: allow the application to execute its own instructions on the CPU
    - This is risky—an application may execute an instruction that controls the hardware, e.g., `hlt` (halt) instruction pauses the CPU
  - Alternative: allow the application to execute "safe" instructions on its own on the CPU
- Example: accessing the solid state drive (SSD)
  - Allowing the application to access the SSD directly
    - This is risky—an application may read/write data it should not be able to access
  - Alternative: asking the OS to access the SSD on the application's behalf
    - Latency of accessing SSD (~1 million CPU cycles) far outweighs the extra instructions required for OS to perform the access on the application's behalf
- Mechanisms
  - Limited Direct Execution (LDE)
  - System calls

Limited Direct Execution

- CPU has two modes of execution: user mode & kernel mode
- *When does a CPU run in user mode?* — when executing application code
- *When does a CPU run in kernel mode?* — when executing OS code
- Allowable operations in user mode are restricted
  - Applications can...
    - Perform arithmetic/logic operations
    - Load/store values in its stack/heap
  - Applications must ask the OS to...
    - Start/terminate applications
    - Create/delete files
    - Display output on screen
    - Read input from user
  - Must transfer control to the OS to perform these operations — How?

System calls

- Invoked via a special assembly instruction: trap (generic) or svc (on ARM)
  - Prior to invoking trap, put system call number for desired operation into a register
- What does the hardware do when a trap instruction is executed?
  1. Save registers to the kernel stack
  2. Switch to kernel mode
  3. Use system call number to index into table of trap handlers — get memory address of first instruction of trap handler
  4. Branch and link to trap handler code (trap handler executes)
  5. Restore registers from the kernel stack
  6. Switch to user mode
  7. Branch to instruction referenced by program counter