Efficiency: locality, caching; operating systems;

COSC 208, Introduction to Computer Systems, 2022-04-11

Announcements

- Project 3 due Thursday

Outline

- Warm-up
- Memory hierarchy
- Locality
- Cache replacement
- OS overview
- Accessing hardware
- Limited direct execution (LDE)

Warm-up: reducing data movement

- Q1: *Cross-out unnecessary loads and stores for each of the following snippets of assembly code*

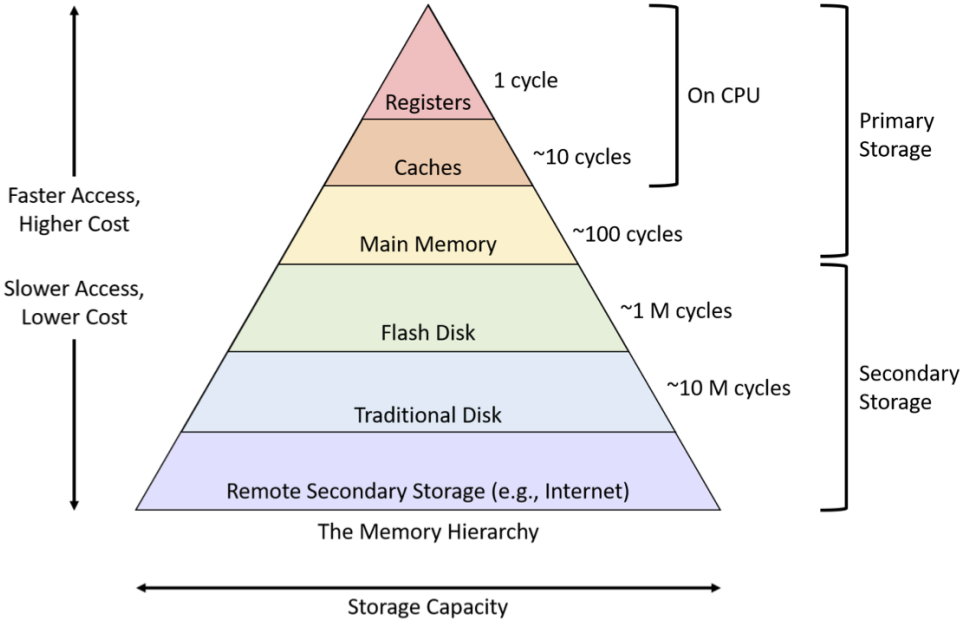
```
000000000000088c <interest_due>:
    88c:    sub    sp, sp, #0x20
    890:    str    w0, [sp, #12]    XXXXX
    894:    str    w1, [sp, #8]     XXXXX
    898:    ldr    w0, [sp, #12]    XXXXX
    89c:    ldr    w1, [sp, #8]     XXXXX
    8a0:    mul    w0, w1, w0
    8a4:    str    w0, [sp, #20]
    8a8:    mov    w0, #0x4b0
    8ac:    str    w0, [sp, #24]    XXXXX
    8b0:    ldr    w1, [sp, #20]
    8b4:    ldr    w0, [sp, #24]    XXXXX
    8b8:    sdiv   w0, w1, w0
    8bc:    str    w0, [sp, #28]    XXXXX
    8c0:    ldr    w0, [sp, #28]    XXXXX
    8c4:    add    sp, sp, #0x20
    8c8:    ret
```

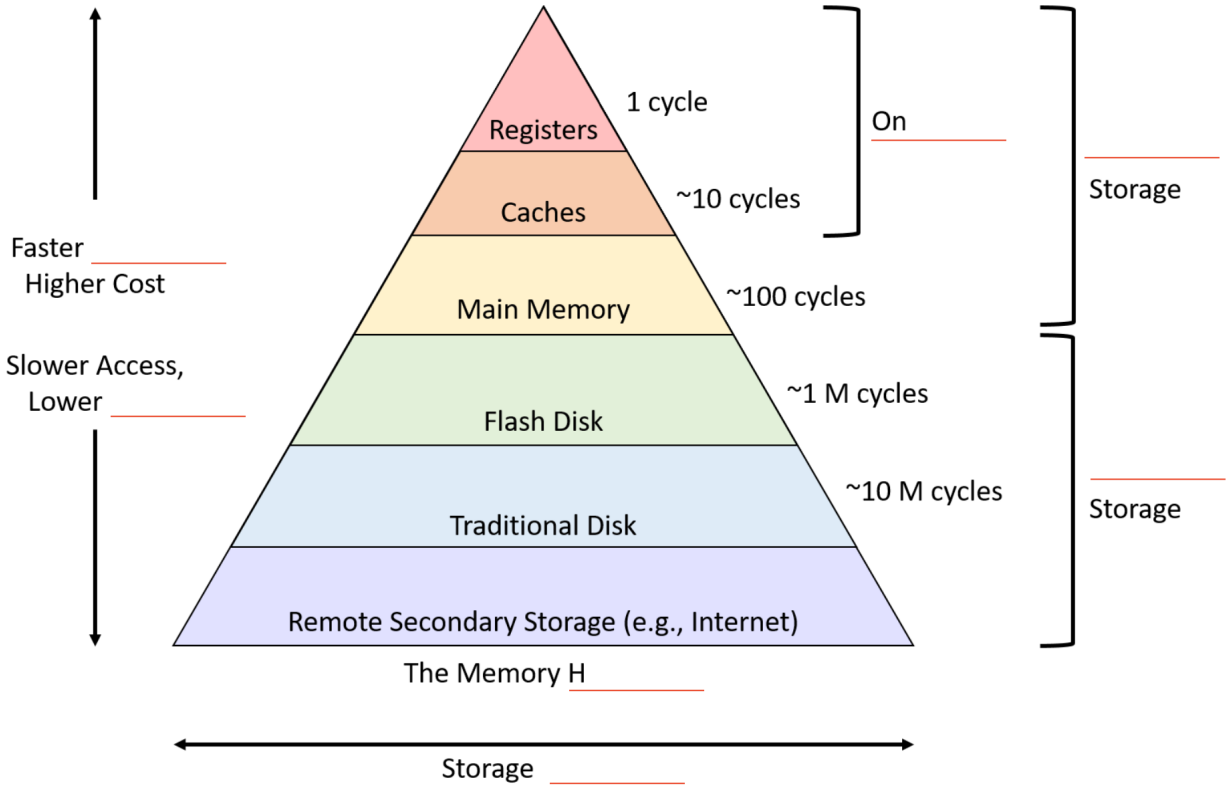
- Doing even more

```
000000000000088c <interest_due>:
    88c:    sub    sp, sp, #0x20
    8a0:    mul    w0, w1, w0
    8a4:    str    w0, [sp, #20]
    8a8:    mov    w0, #0x4b0
    8b0:    ldr    w1, [sp, #20]
    8b8:    sdiv   w0, w1, w0
    8c4:    add    sp, sp, #0x20
    8c8:    ret
```

- Update register assignments to eliminate loads and stores
  - Must preserve calling conventions
    - Parameters are stored in w/x0, w/x1, ...
    - Return value is stored in w/x0
    - Caller must store register values into caller's stack frame before `bl` to callee — actually only needed if values in registers are needed by caller after `bl` and callee overwrites the values in those registers
  - Example

```
000000000000088c <interest_due>:
    88c:    sub    sp, sp, #0x20    XXXXX
    8a0:    mul    w0, w1, w0
    8a4:    str    w0, [sp, #20]    XXXXX
    8a8:    mov    w0, #0x4b0       // mov w1 #0x4b0
    8b0:    ldr    w1, [sp, #20]    XXXXX
    8b8:    sdiv   w0, w1, w0       // sdiv w0, w0, w1
    8c4:    add    sp, sp, #0x20    XXXXX
    8c8:    ret
```

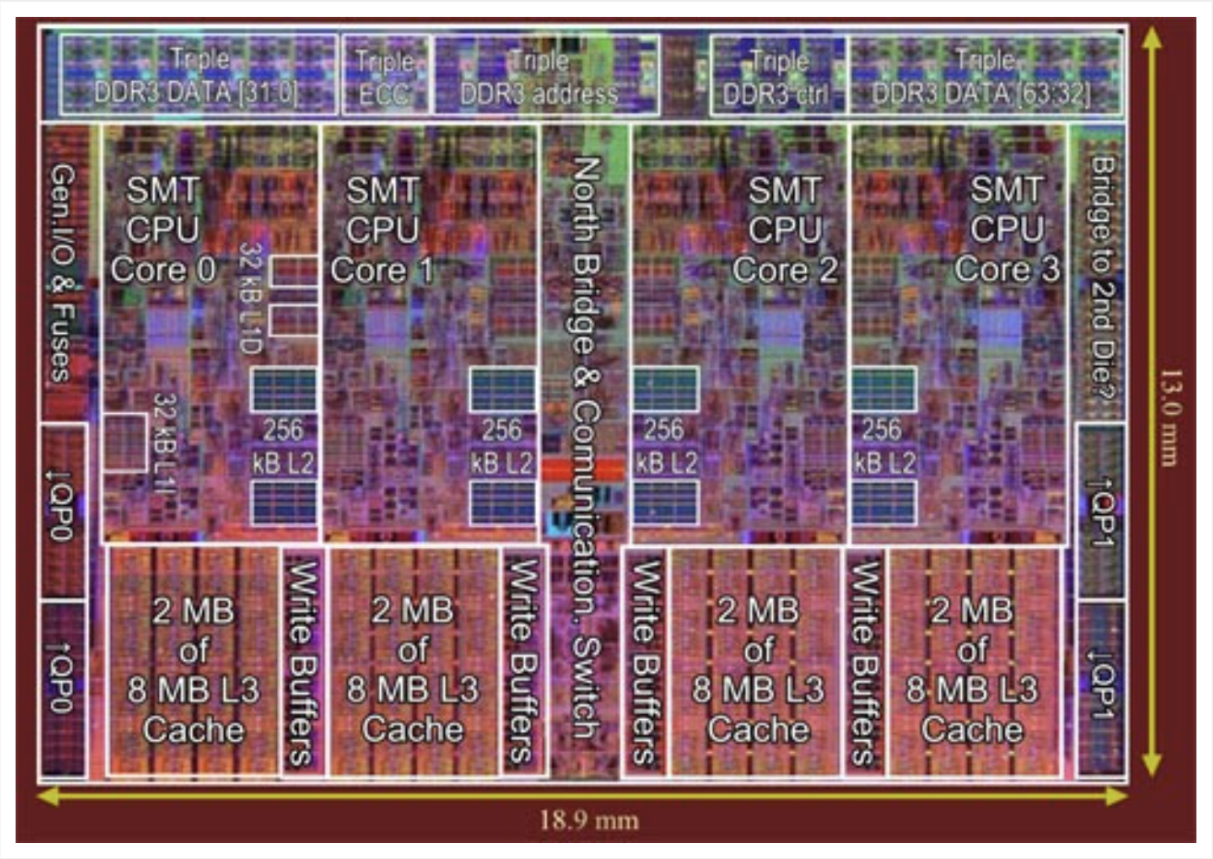




Memory hierarchy

- Compares various forms of storage in terms of
  - Access latency
  - Capacity
  - Cost
  - Volatility
- Access latency
  - Let's consider a 1hz CPU, which means 1 cycle = 1 second
  - Registers — 1 cycle = 1 second
  - Caches — ~10 cycles = ~10 seconds
  - Main memory — ~100 cycles = ~2 minutes
  - Solid-state drive — ~1 million cycles = ~11.5 days
  - Hard (i.e., traditional) disk drive — ~10 million cycles = ~115 days
  - Remote (i.e., network) storage — ~20ms = ~2 years
- Storage capacity
  - Let's assume 1 byte = 1mL
  - Registers — 30 * 8B = ~250mL = ~1 cup
  - Caches (Core i7 in MacBook Pro)
    - L1 — 32KB + 32KB = 64L = ~1 tank of gas
    - L2 — 512KB * 4 cores = 2048L = ~7 bathtubs
  - Main memory = 32GB (in MacBook Pro) = ~13 olympic swimming pools
  - SSD = 1TB (in MacBook Pro) = ~Lake Moraine
- Cost
  - 2 x 16GB DRAM = ~$100 = $3.12 per GB
  - 1TB SSD = $80 = $0.08 per GB
  - 2TB HDD = $60 = $0.03 per GB
- Volatility
  - Primary storage (registers, caches, and main memory) — volatile (i.e., data is lost if power is lost)
  - Secondary storage (SSD, HDD, network storage) — non-volatile (i.e., data is preserved if power is lost)

Intel Nehalem CPU die picture.



efficient, easy, near instant retrieval

Instances of caching

- CPU caches
  - *Why do we have caches on the CPU?* — accessing main memory is ~100x slower than accessing a register
  - Store instructions and data (stack, heap, etc.) from main memory
  - Three levels --- L1, L2, and L3 (last-level cache)
  - Range in size from a few KB to a few MB
  - Cache line (i.e., cache entry) is typically larger than a word — e.g., 128 bytes
    - *Why?* — spatial locality
  - What happens when we write to memory?
    - Write through cache — write to the cache and main memory
    - Write back cache — initially write to the cache; write to main memory when the entry is evicted from the cache
    - What are the advantages of each approach?
      - Write through cache ensures consistency between CPU cores
      - Write back cache only incurs the overhead of accessing main memory when absolutely necessary
- Web browser caches
  - *Why do web browsers have caches?*
    - Accessing remote network storage is >50x slower than accessing a solid state drive (SSD)
    - Spatial locality — many aspects of a web page are also used with other pages on the same site: e.g., images, Cascading Style Sheets (CSS), JavaScript (JS)
    - Temporal locality — users often visit the same web page repeatedly: e.g., Google
    - Internet Service Provider (ISP) may limit amount of data downloaded/uploaded per month
  - Store static content (e.g., images, CSS, JS)
  - Web browser caches are read-only

- Domain Name System (DNS) caches
    - DNS is used to translate domain names (e.g., `portal.colgate.edu`) into Internet Protocol (IP) addresses (e.g., `149.43.134.29`)
    - DNS entries—i.e., mappings from domain names to IP addresses—can be cached by a web browser, an operating system, and a recursive resolver
        - A recursive resolver is a DNS server within a network that receives DNS queries from clients and queries other DNS servers on the client's behalf in order to locate the desired DNS entry
    - Why are DNS entries cached?
        - Contacting a recursive resolver (and other DNS servers) is much slower than access a solid state drive (SSD)
        - Spatial locality — users often visit subdomains of a domain, e.g., `portal.colgate.edu`, `moodle.colgate.edu`, and `cs.colgate.edu` are subdomains of `colgate.edu`
        - Temporal locality — users often visit the same domain repeatedly
        - Mappings from domain names to IP addresses change infrequently —> DNS entries can be cached for hours or days
- Content distribution networks (CDNs)
    - Collection of geographically distributed servers that delivery content (e.g., streaming videos) to users
    - User's computers contact a server that is "nearby"
        - Ideally measured in terms of latency, which is a function of geographic distance, network routes, and network load
        - Analogy: time it takes to drive somewhere is a function of geographic distance, the route you take, and the amount of traffic on the road
    - CDN servers fetch and cache content from origin servers
    - Popular content (e.g., image from the front page of the NY Times) is more likely to already be cached

Cache replacement

- If a cache is full, then a cache entry must be removed so different data can be placed in the cache
- Cache replacement policy governs which data is removed
- *What should a good cache replacement policy do?*
  — maximize the number of cache hits (or minimize the number of cache misses)
    - Evaluation metric: Hit ratio = number of hits / total number of memory accesses
- *How do we determine which cache entry to replace?* different policies with tradeoffs.
- **Optimal** replacement policy: policy that replaces the entry that will be accessed furthest in the future
    - Impractical because we don't know data access patterns a priori
- **Least Frequently Used** (LFU) and **Least Recently Used** (LRU)
    - Based on the principle of locality
    - LFU policy replaces entry that was accessed the least frequently in the past assumes a item that is accessed many times will be accessed many more times
    - LRU policy replaces entry that was accessed furthest in the past assumes a item that was accessed recently will be accessed again soon
    - Downside: lots of *overhead* to implement
      — need to store an ordered list of items and move a item up in the list whenever it's accessed
    - Where does this go wrong? — when working-set size (i.e., number of repeatedly accessed entries) is (slightly) greater than size of the cache

- **First-in First-out** (FIFO)
    - Simple to implement
    - Doesn't consider the importance of a cache entry
- **Random**
    - Even simpler to implement
    - Doesn't consider the importance of a cache entry

Example:

- Q2: *Assume a cache can hold 3 entries and the following 15 data accesses occur: 3, 4, 4, 5, 3, 2, 3, 4, 1, 4, 4, 2, 5, 2, 4. Assuming the cache is initially empty, what is the hit ratio for each of the following algorithms?*
    - *FIFO* — +3, +4, H4, +5, H3, -3/+2, -4/+3, -5/+4, -2/+1, H4, H4, -3/+2, -4/+5, H2, -1/+4 — 5/15 - 33%
    - *LRU* — +3, +4, H4, +5, H3, -4/+2, H3, -5/+4, -2/+1, H4, H4, -3/+2, -1/+5, H2, H4 — 7/15 = 47%
    - *LFU* — +3, +4, H4, +5, H3, -5/+2, H3, H4, -2/+1, H4, H4, -1/+2, -2/+5, -5/+2, H4 — 7/15 = 47%
    - *Optimal* — +3, +4, H4, +5, H3, -5/+2, H3, H4, -3/+1, H4, H4, H2, -1/+5, H2, H4 — 9/15 = 60%

Temporal vs. spatial locality

- *What is temporal locality?*
    - Access the same data repeatedly
    - E.g., for loop variable
- *What is spatial locality?*
    - Access data with a similar scope
    - E.g., next item in array
    - E.g., local variables/parameters, which are stored in the same stack frame
- Analogies for temporal and spatial locality
    - Book storage (Dive Into Systems Section 11.3.2)
        - Temporal locality — store most frequently used books at your desk, less frequently used books on your bookshelf, and least frequently used books at the library
        - Spatial locality — checkout books on the same/nearby subjects when you go to the library
    - Groceries (pre-class questions 3 & 4)
        - Temporal locality — you store food you eat frequently in the front of the refrigerator, while you store food you eat infrequently in the back of the refrigerator
        - Spatial locality — you organize the items on your grocery list based on the aisle in which they are located
    - Breakout groups: *Develop your own analogy for temporal and spatial locality*

Q3: *For each of the following instances of caching, indicate whether the caching is motivated by temporal or spatial locality.*

- A CPU caches the first 32 instructions of a function when the function is called — spatial
- A CPU caches all of the instructions for a frequently called function — temporal
- A web browser caches the Moodle pages for your courses, which you view multiple times per week — temporal
- A content distribution network (CDN) caches a video that has gone viral — temporal
- A content distribution network (CDN) caches "recommended videos" related to a popular video — spatial

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