CS153: Memory Management 3

Chengyu Song

Slides modified from Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Recap

- Memory required to hold page table can be significant
  - Multi-level page tables
- Memory reference overhead
  - TLB - caching address translation
Memory hierarchies

• Some fundamental and enduring properties of hardware and software:
  • Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  • Well-written programs tend to exhibit good locality.
• These fundamental properties complement each other beautifully
  • Suggest an approach for organizing memory and storage systems known as a memory hierarchy.
Examples of speed and price gap

- **SRAM (Static Random Access Memory)**
  - Latency: 0.5-2.5 ns, cost: ~$5000 per GB
- **DRAM (Dynamic Random Access Memory)**
  - Latency: 50-70 ns, cost: ~$20 - $50 per GB
- **SSD / NVM (Non-Volatile Memory)**
  - Latency: 70-150 ns, cost: ~$4 - $12 per GB
- **Magnetic disk**
  - Latency: 5-20 ms, cost: ~$0.02 - $2 per GB
Caches

• A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

• Fundamental idea of a memory hierarchy
  • For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

• Why do memory hierarchies work?
  • Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
Locality

• **Principle of Locality**
  • Programs tend to use data and instructions with addresses near or equal to those they have used recently

• **Temporal locality**
  • Recently referenced items are likely to be referenced again in the near future

• **Spatial locality**
  • Items with nearby addresses tend to be referenced close together
General cache concepts

Cache: Smaller, faster, more expensive memory caches a subset of the blocks

Memory: Larger, slower, cheaper memory viewed as partitioned into “blocks”

Data is copied in block-sized transfer units
Cache hit

Cache

Request: 14

Data in block b is needed

Block b is in cache: Hit!
Cache miss

Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache
• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)
Types of cache miss

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.

- Conflict miss
  - When mapping a larger set of blocks at level $k+1$ to a smaller subset of blocks at level $k$, multiple data objects will map to the same block.

- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
An example memory hierarchy

- **L0:** Registers
  - CPU registers hold words retrieved from L1 cache
- **L1:** L1 cache (SRAM)
  - L1 cache holds cache lines retrieved from L2 cache
- **L2:** L2 cache (SRAM)
  - L2 cache holds cache lines retrieved from main memory
- **L3:** Main memory (DRAM)
  - Main memory holds disk blocks retrieved from local disks
- **L4:** Local secondary storage (local disks)
  - Local disks hold files retrieved from disks on remote network servers
- **L5:** Remote secondary storage (tapes, distributed file systems, Web servers)

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte
Examples of caching in the hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-8 bytes words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>On/Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
Virtual memory

- Memory that is not backed by physical memory but swapped to disk
  - Pages are evicted to disk when memory is full
  - Pages loaded from disk when referenced again
- Mechanism
  - Page translation - V bit in the PTE
- Replacement policy
  - Who get evicted
Page eviction

- When physical memory is fully utilized or the process has used its physical memory quota, and
  - More memory is required
  - A paged out memory is accessed and needs to be page in, or
- How?
  1. Transfer evicted page to the disk
  2. Update the corresponding PTE
  3. Recycle the page frame
Evicting the best page

- The goal of the replacement algorithm is to reduce the fault rate
- The best page to evict is the one never touched again
- Never is a long time, so picking the page closest to "never" is the next best thing
  - Evicting the page that won't be used for the longest period of time
  - Proved by Bélády
- We're now going to survey various replacement algorithms, starting with Belady's
Belady's algorithm

• Belady's algorithm is known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream.
  • **Idea:** replace the page that will not be used for the longest time in the future.
  • **Problem:** how to predict the future.
Belady's algorithm (cont.)

• Why is Belady's still useful?
  • Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
    • If optimal is not much better, then algorithm is pretty good
    • If optimal is much better, then algorithm could use some work
      • Random replacement is often the lower bound
First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago is not being used
- Why might this be bad?
  - Then again, maybe it’s not
  - We don't have any info to say one way or the other
Belady's Anomaly

• When we have more physical memory, we are expecting the fault rate to **decrease**

• When the fault rate actually **increases** when the algorithm is given more memory (**very bad**), we call it the "Belady's Anomaly"
  
  • FIFO suffers from Belady's Anomaly
Example of Belady's Anomaly
Least Recently Used (LRU)

• LRU uses reference information to make a more informed replacement decision
  • **Idea:** We can't predict the future, but we can make a guess based upon past experience
  • On replacement, evict the page that has not been used for the longest time in the past (Belady's: future)
  • When does LRU do well? When does LRU do poorly?
Approximating LRU

• To be true LRU, need to time stamp every reference (or maintain a stack) – much too costly

• So we need to approximate it, using the PTE reference bit
  • Keep a counter for each page
  • At regular intervals, for every page do:
    • If ref bit == 0, increment counter; else, zero the counter
    • Zero the reference bit
  • The page with the largest counter is the least recently used
LRU Clock (Not Recently Used)

- Not Recently Used (NRU) – Used by Unix
  - Replace page that is "old enough"
  - Arrange all of physical page frames in a big circle (clock)
  - A clock hand is used to select a good LRU candidate
    - Sweep through the pages in circular order like a clock
    - If the ref bit is off, it hasn't been used recently
    - If the ref bit is on, turn it off and go to next page
  - Arm moves quickly when pages are needed
Example: gcc page replace

![Graph showing the comparison of different page replacement algorithms](image)

- Optimal
- LRU
- Clock
- FIFO
- LIFO
- LFU
- Random
Working set model

- A working set of a process is used to model the dynamic locality of its memory usage
- Defined by Peter Denning in 60s
  - $WS(t,w) = \{\text{set of pages P, such that every page in P was referenced in the time interval (t, t-w)}\}$
  - $t$ – time, $w$ – working set window (measured in page refs)
- A page is in the working set (WS) only if it was referenced in the last $w$ references
Example: gcc working set
Working set problems

• Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  • Denning: don't run a process unless working set is in memory
    (scheduling)
• Problems
  • How do we determine w?
  • How do we know when the working set changes?
• However, it is still used as an abstraction
Thrashing

- When most of the time is spent by the OS in paging data back and forth
  - No time spent doing useful work (making progress)
- In this situation, the system is **overcommitted**
  - No idea which pages should be in memory to reduce faults
  - Could just be that there isn't enough physical memory
- Possible solutions
  - Swapping – write out all pages of a process
  - **Buy more memory**
Summary

• Memory hierarchy
  • Speed and price gap
  • Program locality -> caching

• Virtual memory
  • Cache replacement policies (apply to all caches)
  • Working set
  • Trashing
Next class ...

- More on paging
- Textbook
  - Module 20