CS 162: Operating Systems and Systems Programming

Lecture 21: TLB, Buffer Cache, Demand Paging

July 31, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Logistics

• Proj 3 Due on August 12
  • Design Doc Due Thursday

• HW3 Due on August 13
  • Last Tuesday of the class
Recall: Paging Tricks

1. Demand Paging: Swapping for pages
   • Keep only active pages in memory
   • Remember: not common on modern systems, except perhaps when first loading program
   • Response: Load in page from disk, retry operation

2. Copy on Write (remember fork?)
   • Temporarily mark pages as read-only
   • Allocate new pages when OS receives protection fault

3. Zero-Fill on Demand
   • Slow to overwrite new pages with all zeros
   • Page starts as invalid, zero it out when it's actually used
Making Exceptions Transparent

- Exception prompts OS to fix something, e.g. finally load copy a page from disk
- Can we return just like with a system call?
  - Not Quite
    - Need to repeat instruction that caused exception, not advance past it
Precise Exceptions

• **Definition**: Machine's state is as if the program executed up to the offending instruction

• Hardware has to complete previous instructions
  • Remember pipelining
  • May need to revert side effects if instruction was partially executed

• OS relies on hardware to enforce this property
Recall: Caching

- **Cache**: Repository for copies that can be accessed more quickly than the originals

- **Goal**: Improve **performance**

- We'll see lots of applications!
  - Memory
  - Address translations
  - File contents, file name to number mappings
  - DNS records (name to IP addr mappings)
Recall: Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology
Recall: Locality

- **Temporal Locality** *(Locality in Time)*:
  - Keep recently accessed data items closer to processor

- **Spatial Locality** *(Locality in Space)*:
  - Move contiguous blocks to the upper levels
How is a Block found in a Cache?

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Block** is minimum unit of caching (recall spatial locality)
  - Data Select: Which part of block to retrieve
- **Index** Used to Lookup Candidates in Cache
  - Index identifies the set
- **Tag** used to identify actual copy
  - If no candidates match, then declare cache miss
Example: Block 12 placed in 8 block cache

32-Block Address Space:

Direct mapped: block 12 can go only into block 4 (12 mod 8)

Set associative: block 12 can go anywhere in set 0 (12 mod 4)

Fully associative: block 12 can go anywhere
Caching Applied to Address Translation

- CPU
  - Virtual Address
- TLB
  - Cached?
    - Yes
    - Translate (MMU)
    - Physical Address
  - No
    - Save Result
    - Physical Memory
- Data Read or Write (untranslated)
Caching Address Translations

- Locality in page accesses?

- Yes: Spatial locality, just like CPU cache

- TLB: Cache of page table entries
TLB and Context Switches

• Do nothing upon context switch?
  • New process could use old process's address space!

• Option 1: Invalidate ("flush") entire TLB
  • Simple, but very poor performance

• Option 2: TLB entries store a process ID
  • Called tagged TLB
  • Requires additional hardware
TLB and Page Table Changes

• Think about what happens we we use `fork`
• OS marks all pages in address space as read-only
• After parent returns from `fork`, it tries to write to its stack
• Triggers a protection fault. OS makes a copy of the stack page, updates page table.
• Restarts instruction.
TLB and Page Table Changes

What if TLB had cached the old read/write page table entry for the stack page?

• OS marks all pages in address space as read-only
• After parent returns from fork, it tries to write to its stack
• Triggers a protection fault. OS makes a copy of the stack page, updates page table.
• Restarts instruction.
How do we invalidate TLB entries?

• Hardware could keep track of where page table for each entry is and monitor that memory for updates…
• Very complicated!
• Especially for multi-level page tables and tagged TLBs

• Instead: The OS must invalidate TLB entries
• So TLB is not entirely *transparent* to OS
TLB and Page Table Changes

• Think about what happens when we use `fork`
• OS marks all pages in address space as read-only and tells MMU to clear those TLB entries.
• After parent returns from `fork`, it tries to write to its stack
• Triggers a protection fault. OS makes a copy of the stack page, updates page table. Also tells MMU to clear that TLB entry.
• Restarts instruction.
Designing a TLB

- Critical path of every memory access
  - Caches use physical addresses (Why?)
  - Very high miss penalty

- Needs to be fast. Low Associativity?
  - Problem: Conflict Misses
  - Index bits could easily coincide
    - Code: 0x0040 0000
    - Data: 0x1000 0000
    - Stack: 0x7FFF 0FFC
Designing a TLB

- Critical path of **every** memory access
  - Caches use physical addresses (Why?)
  - **Very** high miss penalty

- Needs to be fast. Low Associativity?
  - More associativity: we pay a cost for each cache lookup
  - **But this is worth it because misses are so slow**
Typical TLB Organization

• 128-1024 Entries
  • Modern CPUs: Multi-level TLBs, just like caches
  • Maybe 128, then 1024 entries in L1, L2 TLBs
  • 2048 entries in brand new Intel CPUs

• Often Highly Associative

• Contains: Virtual to Physical Address Mapping
  • Other Info: Tagged TLB – Process ID
  • Aspects of page table entry: permission bits, etc.
Example: MIPS R3000

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W</td>
<td>34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among 64 user processes without TLB flush
Overlapping TLB & Cache Access

• Idea: Cache access shouldn’t need physical page number immediately

  virtual address
  | Virtual Page # | Offset |
  |
  physical address
  | tag / page # | index | byte |

• Cache access starts off with index + byte
• In parallel, check TLB for physical page number
• Then we can check tag bits last
Overlapping TLB & Cache Access

- What if cache is larger?
  - Need something more sophisticated (CS 152)
- Alternative: Just use virtual addresses in caches
Exercise

• Assume we have a machine with 4KB pages and want to overlap the cache and TLB lookups

• What is the maximum amount of data we can store in a direct-mapped cache?

• What about a 2-way set associative cache?

• 4-way set associative cache?
Putting Everything Together: Address Translation

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)
- Page Table Pointer
- Physical Address:
  - Physical Page #
  - Offset

Page Table (2nd level)

Physical Memory:

Diagram shows the process of address translation from virtual to physical memory, involving page tables at different levels.
Putting Everything Together: TLB

Virtual Address:

Virtual P1 index Virtual P2 index Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:

Physical Page # Offset

Physical Memory:
Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Memory:

Physical Address:
- Physical Page #
- Offset

Cache:
- tag
- index
- byte

Putting Everything Together: Cache
More Caching Coming Up

• Paged Virtual Memory (memory as cache for disk)

• File Systems (Buffer Cache)
Impact of Caches on OS

• Need to deal with consequences of having a cache
  • Maintaining correctness (e.g., clear entries from TLB if the underlying page table changes)

• Process and Thread Scheduling
  • Rapid interleaving may degrade cache performance

• Designing operating system data structures to account for caching (want good performance)
Break
Buffer Cache

• Copies of disk blocks cached in main memory for the kernel's use

• Could be data blocks or contain inodes, directory contents, etc.

• Possibly dirty (modified and not written back)
Buffer Cache Replacement Policy

• Replacement Policy: Least Recently Used
  • Can afford to record and maintain timestamp for each disk block (already going through a read/write op)
  • Keep in kernel memory
  • This is in contrast to demand paging

• LRU Weakness: Scan through filesystem, e.g. to search for file by name – no block reuse

• Alternative option: application gives a hint, "this file is only used once"
Prefetching/Read Ahead

• Goal: Avoid *compulsory misses* in buffer cache
• Think of spatial locality in traditional mem cache

• **Idea:** Guess that files/directories read together
• If process reads part of a file, fetch and cache the rest of its blocks *before the process asks for them*

• "Almost" Free: Sequential Read
Delayed Writes

• Writes not immediately sent to disk
  • So buffer cache is a write back cache

• `write` copies data from user space to kernel buffer
  • Other apps **read data from cache** instead of disk
  • Cache is `transparent` to user programs

• Flushed to disk periodically

• In Linux: kernel threads flush buffer cache very 30 sec. in default setup
Delayed Writes

• Delay block allocation: May be able to allocate multiple blocks at same time for file, keep them contiguous

• Some files never actually make it all the way to disk
  • Remember all those short-lived files we discussed when studying real file system usage patterns?

• But what if system crashes before buffer cache block is flushed to disk?

• And what if this was for a directory file?
  • Lose pointer to inode

• As we've seen: file systems have recovery mechanisms
Demand Paging

- Main memory as a cache for SSD/Disks

- Idea: Present *illusion* we have the speed of RAM and the capacity of disk at program’s disposal
Illusion of Infinite Memory

- Transparent?
  - Performance: No, disk is much slower
  - Correctness: Yes
Memory as a Cache for Disk

• **Block Size:** 1 Page

• **Associativity:** Fully Associative
  • Any page can go in any frame

• **Write Policy:** Write Back (disk writes are slow)

• **Replacement Policy:** LRU Approximation
32-bit x86 Page Table Entry

- 10/10/12 Split of Virtual Address
- Top-level page tables called *directories*

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Unused (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **A**: Set by hardware when page is first *accessed*
- **D**: Page marked *dirty* if it is ever modified

*Page Table entry tells OS about usage patterns*
Why Might this Approach Work?  
Working Set Model

• Theory: Programs transition through sequence of “working sets” – subsets of their address spaces.
Cache Behavior under WS model

- Hit Rate vs. Cache Size
- Graph shows the relationship between cache size and hit rate, with a new working set fitting at a certain cache size
Another model: Zipf/Power Law

- "Heavy-Tailed" distribution
- Lots of rare items – lots of unavoidable misses
- A few very common items – caching very helpful

\[ P(\text{access(rank)}) = \frac{1}{\text{rank}} \]
Demand Paging: Page Replacement

1. Process access page on disk, PTE indicates invalid
2. MMU traps to OS – Page Fault
3. OS chooses a page to replace
   • If dirty bit set, must write back to disk
4. OS loads new page into memory
5. OS updates page table entries (for new and evicted pages) and invalidates relevant TLB entries
6. OS restarts process from just before access
   • Precise Exceptions!
Summary: Page Fault

1. Reference
2. Trap
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

Load M

Operating System

Page Table

Free Frame

Physical Memory
Aside: Swapping in Modern Sys?

• Recall: We've already said that swapping performance is unacceptable in a modern system

• Hardware has changed since the idea of demand paging was introduced
  • CPUs have gotten *much* faster
  • Disks/SSDs have gotten faster, but still several *orders of magnitude* slower than CPU
  • Capacity of RAM has gotten larger

• Modern OSs only swap out something *very* idle or when under memory pressure
So why talk about it?

• Shows power of indirection – OS can play tricks with memory, all unseen by user program

• Could imagine lazily loading an executable
  • "Compulsory Misses" in memory prompt read from disk

• As an instance of caching
  • Very expensive miss time
  • Cache Eviction Policy => Page Replacement Policy
Page Replacement

• Where does a free page frame come from?
• Hopefully: **Free List**
  • Clear out space in background, ahead of time
  • Maintain pool of free pages for future use

• Bad case: Have to evict a page from physical memory on demand at time of replacement
How bad is a miss?

• Recall: Expected Access Time
  \[ \text{EAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time} \]
  \[ = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \]

• Reasonable Numbers:
  • Hit Time = Memory Access Time = 200ns
  • Miss Penalty = Page Fault Service Time = 8 ms

• EAT = 200 ns + Miss Rate \times 8 ms

• Miss Rate of 0.1\%: EAT = 8200 ns

• Want EAT 220ns: Miss Rate = \textbf{0.00025\%}
How do we limit misses?

• Compulsory Misses: Prefetching
• Capacity Misses: Can't quickly add more RAM
• Conflict Misses: Don't technically occur, RAM acts as fully associative cache

• Comes down to having a good page replacement policy
Summary: Applications of Caching

1. **TLB: Cache address mappings**
   - Crucial for reasonable memory access times
   - Not transparent: OS does some management

2. **Buffer Cache: Cache file system blocks**
   - LRU Replacement Policy: Have necessary space and time
   - Read ahead to avoid compulsory misses
   - Delay writes

3. **Demand Paging: Memory behaves as disk cache**
   - Handle page fault by loading necessary page into mem
   - A miss is so expensive, it must be highly unlikely
   - Replacement policy important (more to come)