Address Translation, and Caching

March 8th, 2017
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu

Physical Address Offset
Recall: Paging
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
  - Physical page # copied from table into physical address
    - Check Page Table bounds and permissions

Summary: Paging

What happens if stack grows to 1110 0000?

Virtual memory view
Page Table
Physical memory view

Virtual memory view
Page Table
Physical memory view
### Summary: Paging

#### Virtual memory view
- stack
- heap
- data

#### Physical memory view
- code

---

### Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- Analysis
  - Pros
    - Simple memory allocation
    - Easy to share
  - Cons:
    - What if address space is sparse?
      - E.g., on UNIX, code starts at 0, stack starts at \((2^{31}-1)\)
      - With 1K pages, need 2 million page table entries!
    - Con: What if table really big?
      - Not all pages used all the time ⇒ would be nice to have working set of page table in memory

- How about combining paging and segmentation?

---

### Fix for sparse address space: The two-level page table

- Tables fixed size (1024 entries)
  - On context-switch, save single PageTablePtr register

- Valid bits on Page Table Entries
  - Don’t need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

---

### Summary: Two-Level Paging

- Stack
- Heap
- Data
### Summary: Two-Level Paging

**Virtual memory view**
- `stack`
- `heap`
- `data`
- `code`

**Page Tables (level 2)**
- `stack`
- `heap`
- `data`

**Physical memory view**
- `stack`
- `heap`
- `data`

**Virtual memory view**
- `100` (0x90)

**Physical memory view**
- `1000 0000`

### Multi-level Translation: Segments + Pages

- **Pros:**
  - Only need to allocate as many page table entries as we need for application
  - In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)

- **Cons:**
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - Seems very expensive!

---

### What about Sharing (Complete Segment)?

**Process A**
- `Base0` Limit0
- `Base1` Limit1
- `Base2` Limit2

**Process B**
- `Base3` Limit3
- `Base4` Limit4
- `Base5` Limit5

**Shared Segment**
- `Base6` Limit6
- `Base7` Limit7

---

### Multi-level Translation Analysis

- **Pros:**
  - Only need to allocate as many page table entries as we need for application
    - In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)

- **Cons:**
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - Seems very expensive!
What is in a Page Table Entry

• What is in a Page Table Entry (or PTE)?
  – Pointer to next-level page table or to actual page
  – Permission bits: valid, read-only, read-write, write-only

• Example: Intel x86 architecture PTE:
  – Address same format previous slide (10, 10, 12-bit offset)
  – Intermediate page tables called “Directories”

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>C</th>
<th>S</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>

P: Present (same as “valid” bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: \( L=1 \rightarrow 4 \text{MB page} \) (directory only).
Bottom 22 bits of virtual address serve as offset

How is the Translation Accomplished?

• What, exactly happens inside MMU?
  • One possibility: Hardware Tree Traversal
    – For each virtual address traverses the page table in hardware
    – Generates a “Page Fault” if it encounters invalid PTE
      » Fault handler will decide what to do
      » More on this next lecture
    – Pros: Relatively fast (but still many memory accesses!)
    – Cons: Inflexible, Complex hardware
  • Another possibility: Software
    – Each traversal done in software
    – Pros: Very flexible
    – Cons: Every translation must invoke Fault!
  • In fact, need way to cache translations for either case!

Recall: Dual-Mode Operation (1/2)

• Can a process modify its own translation tables?
  – NO!
  – If it could, could get access to all of physical memory
  – Has to be restricted somehow

• To Assist with Protection, hardware provides at least two modes (Dual-Mode Operation):
  – “Kernel” mode (or “supervisor” or “protected”)
  – “User” mode (Normal program mode)
  – Mode set with bits in special control register only accessible in kernel-mode

Recall: Dual-Mode Operation (2/2)

• Intel processor actually has four “rings” of protection:
  – PL (Privilege Level) from 0 – 3
    » PL0 has full access, PL3 has least
  – Privilege Level set in code segment descriptor (CS)
  – Mirrored “IOPL” bits in condition register gives permission to programs to use the I/O instructions
  – Typical OS kernels on Intel processors only use PL0 (“kernel”) and PL3 (“user”)
How to get from Kernel→User

• What does the kernel do to create a new user process?
  – Allocate and initialize address-space control block
  – Read program off disk and store in memory
  – Allocate and initialize translation table
    » Point at code in memory so program can execute
    » Possibly point at statically initialized data
  – Run Program:
    » Set machine registers
    » Set hardware pointer to translation table
    » Set processor status word for user mode
    » Jump to start of program

• How does kernel switch between processes?
  – Same saving/restoring of registers as before
  – Save/restore PSL (hardware pointer to translation table)

Recall: User→Kernel (System Call)

• Can’t let inmate (user) get out of padded cell on own
  – Would defeat purpose of protection!
  – So, how does the user program get back into kernel?

  • System call: Voluntary procedure call into kernel
    – Hardware for controlled User→Kernel transition
    – Can any kernel routine be called?
      » No! Only specific ones.
    – System call ID encoded into system call instruction
      » Index forces well-defined interface with kernel

Recall: System Call Continued (1/2)

• What are some system calls?
  – I/O: open, close, read, write, lseek
  – Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  – Process: fork, exit, wait (like join)
  – Network: socket create, set options

• Are system calls constant across operating systems?
  – Not entirely, but there are lots of commonalities
  – Also some standardization attempts (POSIX)

Recall: System Call Continued (2/2)

• What happens at beginning of system call?
  » On entry to kernel, sets system to kernel mode
  » Handler address fetched from table/Handler started

• System call argument passing:
  – In registers (not very much can be passed)
  – Write into user memory, kernel copies into kernel mem
    » User addresses must be translated!
      » Kernel has different view of memory than user
    – Every argument must be explicitly checked!
Recall: User→Kernel (Exceptions: Traps & Interrupts)

• A system call instruction causes a synchronous exception (or “trap”)
  – In fact, often called a software “trap” instruction
• Other sources of Synchronous Exceptions (“Trap”):
  – Divide by zero, Illegal instruction, Bus error (bad address, e.g., unaligned access)
  – Segmentation Fault (address out of range)
  – Page Fault (for illusion of infinite-sized memory)
• Interrupts are Asynchronous Exceptions
  – Examples: timer, disk ready, network, etc….
  – Interrupts can be disabled, traps cannot!

Closing thought: Protection without Hardware (1/2)

• Does protection require hardware support for translation and dual-mode behavior?
  – No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
• Protection via Strong Typing
  – Restrict programming language so that you can't express program that would trash another program
  – Loader needs to make sure that program produced by valid compiler or all bets are off
  – Example languages: LISP, Ada, Modula-3 and Java

Recall: User→Kernel (Exceptions: Traps & Interrupts)

• On system call, exception, or interrupt:
  – Hardware enters kernel mode with interrupts disabled
  – Saves PC, then jumps to appropriate handler in kernel
  – For some processors (x86), processor also saves registers, changes stack, etc.
• Actual handler typically saves registers, other CPU state, and switches to kernel stack

Closing thought: Protection without Hardware (2/2)

• Protection via software fault isolation:
  – Language independent approach: have compiler generate object code that provably can’t step out of bounds
    » Compiler puts in checks for every “dangerous” operation (loads, stores, etc). Again, need special loader.
    » Alternative, compiler generates “proof” that code cannot do certain things (Proof Carrying Code)
  – Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)
Administrivia

- Midterm #1 regrades deadline, tomorrow (Thursday) at 11:59PM
- 2nd project out
  - You’ll learn virtual memory, how to load and run the program, how to use the file system, etc
- Midterm 2 coming up on Tue 3/21 6:30-8PM
  - All topics up to and including Lecture 15
    » Focus will be on Lectures 11 – 15 and associated readings
    » Projects 1 and 2
    » Homework 0 – 2
  - Closed book
  - 2 pages hand-written notes both sides

Caching Concept

- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Why Bother with Caching?

- Processor-DRAM Memory Gap (latency)
  - "Moore's Law" (really Joy's Law)
  - "Less' Law?"
  - Processor-Memory Performance Gap:
    - (grows 50% / year)
    - DRAM 9%/yr:
      - (2X/10 yrs)
• Cannot afford to translate on every access
  – At least three DRAM accesses per actual DRAM access
  – Or: perhaps I/O if page table partially on disk!
• Even worse: What if we are using caching to make memory access faster than DRAM access?
• Solution? Cache translations!
  – Translation Cache: TLB (“Translation Lookaside Buffer”)

Another Major Reason to Deal with Caching

Why Does Caching Help? Locality!

Memory Hierarchy of a Modern Computer System

• 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

A Summary on Sources of Cache Misses

• Compulsory (cold start or process migration, first reference): first access to a block
  – “Cold” fact of life: not a whole lot you can do about it
  – Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant
• Capacity:
  – Cache cannot contain all blocks access by the program
  – Solution: increase cache size
• Conflict (collision):
  – Multiple memory locations mapped to the same cache location
  – Solution 1: increase cache size
  – Solution 2: increase associativity
• Coherence (Invalidation): other process (e.g., I/O) updates memory
How is a Block found in a Cache?

- 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
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don’t have data select field

Review: Direct Mapped Cache

- Direct Mapped 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

Review: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

Review: Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block
Where does a Block Get Placed in a Cache?

- 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

Review: Which block should be replaced on a miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

- Miss rates for a workload:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way LRU</th>
<th>2-way Random</th>
<th>4-way LRU</th>
<th>4-way Random</th>
<th>8-way LRU</th>
<th>8-way Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
<td>5.3%</td>
<td>4.4%</td>
<td>5.0%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
<td>1.7%</td>
<td>1.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>

Review: What happens on a write?

- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    » PRO: read misses cannot result in writes
    » CON: Processor held up on writes unless writes buffered
  - WB:
    » PRO: repeated writes not sent to DRAM
    » CON: More complex
    Read miss may require writeback of dirty data

Caching Applied to Address Translation

- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds
What Actually Happens on a TLB Miss? (1/2)

• Hardware traversed page tables:
  – On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    » If PTE valid, hardware fills TLB and processor never knows
    » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards

• Software traversed Page tables (like MIPS)
  – On TLB miss, processor receives TLB fault
  – Kernel traverses page table to find PTE
    » If PTE valid, fills TLB and returns from fault
    » If PTE marked as invalid, internally calls Page Fault handler

What Actually Happens on a TLB Miss? (2/2)

• Most chip sets provide hardware traversal
  – Modern operating systems tend to have more TLB faults since they use translation for many things
  – Examples:
    » shared segments
    » user-level portions of an operating system

What happens on a Context Switch?

• Need to do something, since TLBs map virtual addresses to physical addresses
  – Address Space just changed, so TLB entries no longer valid!
• Options?
  – Invalidate TLB: simple but might be expensive
    » What if switching frequently between processes?
  – Include ProcessID in TLB
    » This is an architectural solution: needs hardware
• What if translation tables change?
  – For example, to move page from memory to disk or vice versa…
    » Must invalidate TLB entry!
    » Otherwise, might think that page is still in memory!

Summary

• The Principle of Locality:
  – Program likely to access a relatively small portion of the address space at any instant of time.
    » Temporal Locality: Locality in Time
    » Spatial Locality: Locality in Space

• Three (+1) Major Categories of Cache Misses:
  – Compulsory Misses: sad facts of life. Example: cold start misses.
  – Conflict Misses: increase cache size and/or associativity
  – Capacity Misses: increase cache size
  – Coherence Misses: Caused by external processors or I/O devices

• Cache Organizations:
  – Direct Mapped: single block per set
  – Set associative: more than one block per set
  – Fully associative: all entries equivalent