CS162
Operating Systems and
Systems Programming
Lecture 13

Address Translation (con't)
Caches and TLBs

October 13, 2010
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Example of segment translation

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2</td>
<td>0x0F000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Let's simulate a bit of this code to see what happens (PC=0x240):
1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 $a0, Move PC+4

2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 $ra (return address!), Move 0x0360 PC

3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000 $v0, Move PC+4

4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850 $t0, Move PC+4

Review: Multi-level Translation

- What about a tree of tables?
  - Lowest level page table \(\rightarrow\) memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

```
Virtual Address:   Virtual Page #   Offset
-------------------  -------------------  -----------
        page #0  V, R            0
        page #1  V, R            4
        page #2  V, R, W        8
        page #3  V, R, W       12
        page #4  N             16
        page #5  V, R, W      20

Physical Page #   Offset
------------------------  ----------
        page #0            0
        page #1            4
        page #2            8
        page #3           12
        page #4           16
        page #5           20
```

- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

Review: Two-level page table

Virtual Address: Virtual P1 index Virtual P2 index Offset
Physical Address: Page Table Pointer

- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single Page Table Pointer register
- Sometimes, top-level page tables called "directories" (Intel)
- Each entry called a (surprise!) Page Table Entry (PTE)
Goals for Today

- Finish discussion of both Address Translation and Protection
- Caching and TLBs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

What is in a PTE?

- 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>O</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>S</th>
<th>G</th>
<th>E</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>
<td></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⇒4MB page (directory only).
  Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    » Region of address space is actually invalid or
    » Page/directory is just somewhere else than memory
  - Validity checked first
    » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives copy of parent address space to child
    » Address spaces disconnected after child created
  - How to do this cheaply?
    » Make copy of parent's page tables (point at same memory)
    » Mark entries in both sets of page tables as read-only
    » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

How is the translation accomplished?

- What, exactly happens inside MMU?
  - One possibility: Hardware Tree Traversal
    » For each virtual address, takes page table base pointer and 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!
Dual-Mode Operation

- Can Application Modify its own translation tables?
  - 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
- Intel processor actually has four “rings” of protection:
  - PL (Priviledge 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)

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

System Call Continued

- 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)
- 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!
User → Kernel (Exceptions: Traps and 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!
- 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

Additions to MIPS ISA to support Exceptions?

- Exception state is kept in "Coprocessor 0"
  - Use mfc0 read contents of these registers:
    - BadVAddr (register 8): contains memory address at which memory reference error occurred
    - Status (register 12): interrupt mask and enable bits
    - Cause (register 13): the cause of the exception
    - EPC (register 14): address of the affected instruction

\[
\begin{array}{cccccccc}
15 & 8 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{Status} & \text{Mask} & \begin{array}{c}
\text{\textbf{k}} \text{\textbf{e}} \text{\textbf{k}} \text{\textbf{e}} \text{\textbf{k}} \text{\textbf{e}} \\
\text{old prev cur}
\end{array}
\end{array}
\]

- Status Register fields:
  - Mask: Interrupt enable
    - 1 bit for each of 5 hardware and 3 software interrupts
  - k = kernel/user: 0 ⇒ kernel mode
  - e = interrupt enable: 0 ⇒ interrupts disabled
  - Exception ⇒ 6 LSB shifted left 2 bits, setting 2 LSB to 0:
    - run in kernel mode with interrupts disabled

Closing thought: Protection without Hardware

- 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 crash 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
- 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 next Monday:
  - Monday, 10/19, 6:00-9:00pm, 155 Dwinelle
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)
- No class on day of Midterm
  - Different Office Hours for me: Mon 11:00-12:30.
  - Office hours during Class time: 4:00-5:30
  - Unfortunately, 2:30-3:30 may be taken. Stay tuned.
- Midterm Topics
  - Topics: Everything up to Today
    - History, Concurrency, Multithreading, Synchronization, Protection/Address Spaces, TLBs
  - Make sure to fill out Group Evaluations!
- Project 2
  - Initial Design Document due Friday 10/15
  - Look at the lecture schedule to keep up with due dates!
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 of the techniques that are 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 \times Hit Time) + (Miss Rate \times Miss Time)

Why Bother with Caching?

- Why Does Caching Help? 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
  - 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”)
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

- On-Chip Cache
- Registers
- Control
- Datapath
- Second Level Cache (SRAM)
- Main Memory (DRAM)
- Secondary Storage (Disk)
- Tertiary Storage (Tape)

### Speed (ns):
- 1s
- 10s-100s
- 100s
- 10,000,000s (10s ms)
- 10,000,000,000s (10s sec)

### Size (bytes):
- 100s
- Ks-Ms
- Ms
- Gs
- Ts

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

- **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)

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

Review: Direct Mapped Cache

- 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

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ex: 0x01</td>
<td>Ex: 0x00</td>
<td></td>
</tr>
<tr>
<td>Valid Bit</td>
<td>Cache Tag</td>
<td>Cache Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 0</td>
</tr>
<tr>
<td></td>
<td>0x50</td>
<td>Byte 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 1023</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 992</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>32-Block Address Space:</th>
<th>Direct mapped:</th>
<th>Set associative:</th>
<th>Fully associative:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block no.</td>
<td>block 12 can go only into block 4 (12 mod 8)</td>
<td>block 12 can go anywhere in set 0 (12 mod 4)</td>
<td>block 12 can go anywhere</td>
</tr>
</tbody>
</table>

---

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)

<table>
<thead>
<tr>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU Random</td>
<td>LRU Random</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</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?

- 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
- 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!
What TLB organization makes sense?

• Needs to be really fast
  - Critical path of memory access
    » In simplest view: before the cache
    » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
• However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)

• Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    » First page of code, data, stack may map to same entry
    » Need 3-way associativity at least?
  - What if use high order bits as index?
    » TLB mostly unused for small programs

TLB organization: include protection

• How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
• TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
• What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a “TLB Slice”

Example for 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>

Example: R3000 pipeline includes TLB “stages”

MIPS R3000 Pipeline

- Inst Fetch
- Dcc/ Reg
- ALU / E.A
- Memory
- Write Reg
- TLB
- I-Cache
- RF
- Operation
- E.A.
- TLB
- D-Cache

Virtual Address Space

- ASID
- V. Page Number
- Offset

- 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

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:

Physical Address

- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early
Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

  - What if cache size is increased to 8KB?
    - Overlap not complete
    - Need to do something else. See CS152/252
  - Another option: Virtual Caches
    - Tags in cache are virtual addresses
    - Translation only happens on cache misses

Summary #1/2

- 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

Summary #2/2: Translation Caching (TLB)

- PTE: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, readable, dirty, user, etc)
- A cache of translations called a “Translation Lookaside Buffer” (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative (Since conflict misses expensive)
  - TLB entries contain PTE and optional process ID
- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault
- On context switch/change in page table
  - TLB entries must be invalidated somehow
- TLB is logically in front of cache
  - Thus, needs to be overlapped with cache access to be really fast