Review: 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:
  - 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

Review: Multi-level Translation

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

  Virtual Address:  
  - Segment # Page # Offset
  - Lowest level page table = memory still allocated with bitmap
  - Higher levels often segmented
  - Could have any number of levels. Example (top segment):
  - Table fixed size
  - On context-switch: save single PageTablePtr register
  - Sometimes, top-level page tables called “directories” (Intel)
  - Each entry called a (surprise!) Page Table Entry (PTE)

Review: Two-level page table

- Virtual Address:  
  - Segment # Page # Offset
  - Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
  - Sometimes, top-level page tables called “directories” (Intel)
  - Each entry called a (surprise!) Page Table Entry (PTE)

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

- Finish discussion of Address Translation
- Caching and TLBs

---

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!

- Really starts to be a problem for 64-bit address space:
  - How big is virtual memory space vs physical memory?

---

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>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</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

---

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

Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    » Much of process space may be out on disk or not in use
- Answer: use a hash table
  - Called an “Inverted Page Table”
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

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 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
- 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

- Still grading exam
  - Will announce results as soon as possible
  - Also will get solutions up very soon!
- Project 2 is started!
  - We moved the design document due date to Wednesday (10/18) at 11:59pm
  - Always keep up with the project schedule by looking on the “Lectures” page
- Make sure to come to sections!
  - There will be a lot of information about the projects that I cannot cover in class
  - Also supplemental information and detail that we don’t have time for in class
**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 x Hit Time) + (Miss Rate x Miss Time)

**Why Bother with Caching?**

- 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")

**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
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

![Memory Hierarchy Diagram]

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

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: 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

Review: Set Associative Cache

- **N-way set associative**: N entries per Cache Index
  - N direct mapped caches operate 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: 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)
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
      processor not held up on writes
    » **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>0xFA000</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**

```
<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcc/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></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

---

Reducing translation time further

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

**Physical Address**

```
<table>
<thead>
<tr>
<th>V page no.</th>
<th>offset</th>
</tr>
</thead>
</table>
```

**TLB Lookup**

```
<table>
<thead>
<tr>
<th>V</th>
<th>Access Rights</th>
<th>PA</th>
</tr>
</thead>
</table>
```

- 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:
  - Overlap not complete
  - Need to do something else. See CS152/252

• 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, writeable, 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