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: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x0000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x0400</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0x0F00</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Review: Implementation of Multi-Segment Model

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: mov [es:bx],ax
- What is "V/N"?
  - Can mark segments as invalid; requires check as well

Virtual Address Format

- Might be shared
- Space for Other Apps
- Shared with Other Apps

Virtual Address Space

Physical Address Space

Physical Address

Segment Map

Error
Review: How to Implement 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
  - Check Page Table bounds and permissions

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):

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

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?
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.
    - Alternatively, 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 I on Wednesday, 3/8:
  - TBA, Room TBA
  - 1½ hour exam
  - Closed book, one page of hand-written notes (both sides), no calculators/PDAs/…
  - Bring ID
- Midterm Topics
  - Topics: Everything up to (and including) today
    - Lectures 1-13, chapters 1-8 (7th ed) or 1-9 (6th ed)
- Extra office hours
  - TBA
- Make sure to attend 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?

Processor–DRAM Memory Gap (latency)

"Moore's Law" (really Joy's Law)

µProc 60%/yr. (2X/1.5yr)

Processor–Memory Performance Gap: (grows 50% / year)

DRAM 9%/yr. (2X/10 yrs)

Why Does Caching Help? Locality!

Prob of reference

Terminology:

Address Space 2n - 1

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
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

**32-Block Address Space:**

```
<table>
<thead>
<tr>
<th>Block no.</th>
<th>0 1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>2 2 2 2 2 2 2 2</td>
</tr>
<tr>
<td>2</td>
<td>3 3 3 3 3 3 3 3</td>
</tr>
<tr>
<td>3</td>
<td>4 4 4 4 4 4 4 4</td>
</tr>
<tr>
<td>4</td>
<td>5 5 5 5 5 5 5 5</td>
</tr>
<tr>
<td>5</td>
<td>6 6 6 6 6 6 6 6</td>
</tr>
<tr>
<td>6</td>
<td>7 7 7 7 7 7 7 7</td>
</tr>
<tr>
<td>7</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
```

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

**Set associative:**
- Block 12 can go anywhere in set 0
  - Block 12 (mod 4)

**Fully associative:**
- Block 12 can go anywhere

---

A Summary on Sources of Cache Misses

- **Compulsory** (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "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 same cache location
  - Solutions: increase cache size, or increase associativity

- **Two others:**
  - **Coherence** (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

---

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

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

Reducing translation time further

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

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

Example: R3000 pipeline includes TLB “stages”

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd/ 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>E.A.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS R3000 Pipeline

- 64 entry, on-chip, fully associative, software TLB fault handler
- Allows context switching among 64 user processes without TLB flush

Virtual Address Space

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

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, 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