Recall: Simple Segmentation (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>0x4000</td>
<td>0x8000</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Recall: Simple 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
  - Con: 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 \(\Rightarrow\) would be nice to have working set of page table in memory

- How about combining paging and segmentation?
  - Segments with pages inside them?
  - Need some sort of multi-level translation
Memory Layout for Linux 32-bit

- Kernel space:
  - User code cannot read from or write to these addresses.
  - User code runs in a segmentation fault
- Stack (grows down)
- Memory Mapping Segment
  - File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/ld/"ld.a"
- Data Segment
  - Static variables: initialized by the programmer. Example: static char *user_name
  - Text segment (ILF)
    - Stores the binary image of the process (e.g., /bin/"/bin/"

Physical Address: 10 bits 10 bits 12 bits
Offset

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

Virtual Address:
- P1 index
- P2 index
- Offset
- PageTablePtr
- 10 bits 10 bits 12 bits
- Physical Address:
- Physical Page #
- Offset
- 4KB

- Tree of Page Tables
- 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

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</th>
<th>O</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>S</th>
<th>C</th>
<th>A</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>P: Present</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>W: Writeable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>U: User accessible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>PWT: Page write transparent: external cache write-through</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>PCD: Page cache disabled (page cannot be cached)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>A: Accessed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>D: Dirty (PTE only): page has been modified recently</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>L: L=1⇒4MB page (directory only). Bottom 22 bits of virtual address serve as offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

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
Recall: Segments + Pages

- 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:
  Virtual Sea # | Virtual Page # | Offset
page #0 V,R
page #1 V,R
page #2 V,R,W
page #3 V,R,W
page #4 N
page #5 V,R,W
```

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

Making it real:
X86 Memory model with segmentation (16/32-bit)

X86 Segment Descriptors (32-bit Protected Mode)

- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
- There are 6 registers: SS, CS, DS, ES, FS, GS
- What is in a segment register?
  - A pointer to the actual segment description:

    | Segment selector [13 bits] | G/L | RPL |
    |-----------------------------|-----|-----|
    | G/L selects between GDT and LDT tables (global vs local descriptor tables) |
    | Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory |
    | Includes length of table (for < 2^13) entries |
    | Descriptor format (64 bits): |

    | Base address (32-bit) | G | LG | Offset (25-16) | P | CPL | S | Type |
    |-----------------------|---|----|----------------|---|-----|---|------|
    | Base address (64-bit) |

- G: Granularity of segment (0: 16bit, 1: 4KiB unit)
- DB: Default operand size (0: 16bit, 1: 32bit)
- A: Freely available for use by software
- P: Segment present
- DPL: Descriptor Privilege Level
- S: System Segment (0: System, 1: code or data)
- Type: Code, Data, Segment

Recall: How are segments used?

- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process
- In legacy applications (16-bit mode):
  - Segments provide protection for different components of user programs
  - Separate segments for chunks of code, data, stacks
  - Limited to 64K segments
- Modern use in 32-bit Mode:
  - Segments “flattened”, i.e. every segment is 4GB in size
  - One exception: Use of GS (or FS) as a pointer to “Thread Local Storage” (TLS)
    » A thread can make accesses to TLS like this:
      mov eax, gs(0x0)
- Modern use in 64-bit (“long”) mode
  - Most segments (SS, CS, DS, ES) have zero base and no length limits
  - Only FS and GS retain their functionality (for use in TLS)
Administrivia

- Midterm I coming up on Wednesday!
  - October 14th: 6:30-9:30PM
  - Rooms: 155/145 Dwinelle
  - All topics up to and including Today
  - Closed book
  - 1 page hand-written notes both sides
- Division by login:
  - Logins aa-jh: 155 Dwinelle
  - Logins ji-pz: 145 Dwinelle
- Project 2 technically released on Wednesday as well

X86_64: Four-level page table!

48-bit Virtual Address:
- 9 bits P1 index
- 9 bits P2 index
- 9 bits P3 index
- 9 bits P4 index
- 12 bits Offset

PageTablePtr

4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

IA64: 64bit addresses: Six-level page table?!?

64bit Virtual Address:
- 7 bits Virtual P1 index
- 9 bits Virtual P2 index
- 9 bits Virtual P3 index
- 9 bits Virtual P4 index
- 9 bits Virtual P5 index
- 9 bits Virtual P6 index
- 12 bits Offset

No!
Too slow
Too many almost-empty tables

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!
**IA64: Inverse Page Table (IPT)**

Idea: index page table by physical pages instead of VM

```
+-------------------+---+-------------------+---+
| VMpage0           | 0 | VMpage1           | 1 |
| VMpage2           | 2 | VMpage3           | 3 |
```

Virtual memory

<table>
<thead>
<tr>
<th>Process id</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VMpage0</td>
</tr>
<tr>
<td>1</td>
<td>VMpage1</td>
</tr>
<tr>
<td>0</td>
<td>VMpage2</td>
</tr>
<tr>
<td>xx</td>
<td>free</td>
</tr>
<tr>
<td>1</td>
<td>VMpage3</td>
</tr>
</tbody>
</table>

Physical memory

```
0x0000 - 0x7000: VMpage0, pid 0
0x1000 - 0x1300: VMpage1, pid 0
0x2000 - 0x2300: VMpage2, pid 0
0x3000 - 0x3700: VMpage3, pid 0
```

Inverse Page Table

```
0x0000 - 0x7000: VMpage0
0x1000 - 0x1300: VMpage2
0x2000 - 0x2300: VMpage1
0x3000 - 0x3700: VMpage3
```

**IPT address translation**

- Use a hash map

```
VMPage2 (52b) Offset (12b)
```

Process 0 virtual address

<table>
<thead>
<tr>
<th>Offset (12b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
</tr>
<tr>
<td>0x1</td>
</tr>
<tr>
<td>0x2</td>
</tr>
<tr>
<td>0x3</td>
</tr>
<tr>
<td>0x4</td>
</tr>
<tr>
<td>0x5</td>
</tr>
<tr>
<td>0x6</td>
</tr>
<tr>
<td>0x7</td>
</tr>
</tbody>
</table>

Physical address

<table>
<thead>
<tr>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
</tr>
<tr>
<td>0x1</td>
</tr>
<tr>
<td>0x2</td>
</tr>
<tr>
<td>0x3</td>
</tr>
<tr>
<td>0x4</td>
</tr>
<tr>
<td>0x5</td>
</tr>
<tr>
<td>0x6</td>
</tr>
<tr>
<td>0x7</td>
</tr>
</tbody>
</table>

**Summary: Inverted Table**

- Virtual memory view
  - Stack
  - Heap
  - Data
  - Code

- Physical memory view
  - Stack
  - Heap
  - Data
  - Code

**Address Translation Comparison**

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Segmentation</td>
<td>Fast context switching: Segment mapping maintained by CPU</td>
</tr>
<tr>
<td>Paging (single-level page)</td>
<td>No external fragmentation, fast easy allocation</td>
</tr>
<tr>
<td>Paged segmentation</td>
<td>Table size ~ # of pages in virtual memory, fast easy allocation</td>
</tr>
<tr>
<td>Two-level pages</td>
<td>Table size ~ # of pages in physical memory</td>
</tr>
<tr>
<td>Inverted Table</td>
<td>Hash function more complex</td>
</tr>
</tbody>
</table>

**External fragmentation**

**Internal fragmentation**

**Large table size ~ virtual memory**

**Multiples memory references per page access**
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!

Recall: Dual-Mode Operation

- 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
- Recall: 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
- Certain operations restricted to Kernel mode:
  - Including modifying the page table (CR3 in x86), and GDT/LDT
  - Have to transition into Kernel mode before you can change them

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
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

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)

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

"Less’ Law?"

Time

Performance

1000

100

10

1

1980

1981

1982

1983

1984

1985

1986

1987

1988

1989

1990

1991

1992

1993

1994

1995

1996

1997

1998

1999

2000

µProc 60%/yr.
(2X/1.5yr)

DRAM 9%/yr.
(2X/10 yrs)
Another Major Reason to Deal 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

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

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

Set Select

Data Select

- **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

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x50</td>
<td>0x01</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
<td></td>
</tr>
</tbody>
</table>

---

- **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

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid</th>
<th>Cache Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
<td></td>
</tr>
</tbody>
</table>

---

- **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

<table>
<thead>
<tr>
<th>Cache Tag (27 bits long)</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

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

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

Set Set Set
- 0 1 2 3

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</td>
<td>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 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 ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W 34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R 0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R 0</td>
</tr>
</tbody>
</table>
Example: R3000 pipeline includes TLB “stages”

MIPS R3000 Pipeline

```
<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>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:

```
Virtual Address

V page no. offset

TLB Lookup

V Access Rights PA

P page no. offset
```

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

Overlapping TLB & Cache Access (1/2)

- Main idea:
  - Offset in virtual address exactly covers the “cache index” and “byte select”
  - Thus can select the cached byte(s) in parallel to perform address translation

Virtual address: Virtual Page # Offset

Physical address: tag / page # index byte

Overlapping TLB & Cache Access

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

```
TLB

4K Cache

assoc lookup

index

page # disp 00

Hit/ Miss

FN

Hit/ Miss

Data
```

- 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
Putting Everything Together: Address Translation

- Physical Address:
  - Offset
  - Physical Page #
- Virtual Address:
  - Offset
  - Virtual P2 index
  - VirtualP1 index
- Page Table (1st level)
- Page Table (2nd level)

Putting Everything Together: TLB

- Physical Address:
  - Offset
- Virtual Address:
  - Offset
- Page Table (1st level)
- Page Table (2nd level)
- TLB:

Putting Everything Together: Cache

- Physical Address:
  - Offset
- Virtual Address:
  - Offset
- Page Table (1st level)
- Page Table (2nd level)
- TLB:

Next Up: What happens when ...

- Process
- virtual address
- MMU
- physical address
- instruction
- MMU
- MMU
- page fault
- physical address
- retry
- exception
- handler
- load page from disk
- scheduler
- OS
- update PT entry
- load page from disk
- scheduler
Summary (1/3)
• Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
• Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
• Inverted page table
  - Size of page table related to physical memory size
• PTE: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)

Summary (2/3)
• 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 (3/3): Translation Caching (TLB)
• 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