Recall: General Address translation

- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between the two views
- **Translation ⇒ much easier to implement protection!**
  - If task A cannot even gain access to task B’s data, no way for A to adversely affect B
  - Extra benefit: every program can be linked/loaded into same region of user address space
Recall: How to Implement Simple Paging?

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

- This physical page appears in address space of both processes
  - They can share information by read and writing to this page
- However, this mapping not great:
  - Process A, R/W at address: 0x00002xxx
  - Process B, RO at address: 0x00004xxx
- Better to map at same virtual address so that you can share linked objects!
Recall: Where is page sharing used?

• The “kernel region” of every process has the same page table entries
  – The process cannot access it at user level
  – But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?

• Different processes running same binary!
  – Execute-only, but do not need to duplicate code segments

• User-level system libraries (execute only)

• Shared-memory segments between different processes
  – Can actually share objects directly between processes
    » Must map page into same place in address space!
  – This is a limited form of the sharing that threads have within a single process
### Summary: Paging

<table>
<thead>
<tr>
<th>Virtual memory view</th>
<th>Page Table</th>
<th>Physical memory view</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td>stack</td>
</tr>
<tr>
<td>1110 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td>heap</td>
</tr>
<tr>
<td>1000 0000</td>
<td></td>
<td>data</td>
</tr>
<tr>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 0000</td>
<td></td>
<td>code</td>
</tr>
<tr>
<td>code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>page # offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What happens if stack grows to 1110 0000?
Summary: Paging

Virtual memory view

- Stack: 1111 1111
- Heap: 1110 0000
- Data: 1000 0000
- Code: 0100 0000

Page Table

- 1111: 11101, 11110, 11101, 11110, null, null, null, null
- 1110: 10111, null, null, null, null
- 1101: null, null, null, null
- 1100: null, null, null, null
- 1011: null, null, null, null
- 1010: null, null, null, null
- 1001: null, null, null, null
- 1000: null, null, null, null
- 0111: null, null, null, null
- 0110: null, null, null, null
- 0101: null, null, null, null
- 0100: null, null, null, null
- 0011: null, null, null, null
- 0010: null, null, null, null
- 0001: null, null, null, null
- 0000: null, null, null, null

Physical memory view

- Stack: 1110 0000
- Heap: 0111 0000
- Data: 0101 0000
- Code: 0001 0000

Allocate new pages where room!
Recall: How big do things get?

- 32-bit address space => \(2^{32}\) bytes (4 GB)
  - Note: “b” = bit, and “B” = byte
  - And for memory:
    » “K” (kilo) = \(2^{10}\) = 1024 \(\approx 10^3\) (But not quite!): Sometimes called “Ki” (Kibi)  
    » “M” (mega) = \(2^{20}\) = (1024)^2 = 1,048,576 \(\approx 10^6\) (But not quite!): Sometimes called “Mi” (Mibi)  
    » “G” (giga) = \(2^{30}\) = (1024)^3 = 1,073,741,824 \(\approx 10^9\) (But not quite!): Sometimes called “Gi” (Gibi)

- Typical page size: 4 KB
  - how many bits of the address is that? (remember \(2^{10} = 1024\))
  - Ans – 4KB = \(4 \times 2^{10} = 2^{12}\) \(\Rightarrow\) 12 bits of the address

- So how big is the simple page table for each process?
  - \(2^{32}/2^{12} = 2^{20}\) (that’s about a million entries) \(\times\) 4 bytes each \(\Rightarrow\) 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory

- How big is a simple page table on a 64-bit processor (x86_64)?
  - \(2^{64}/2^{12} = 2^{52}\)(that’s \(4.5 \times 10^{15}\) or 4.5 exa-entries) \(\times\) 8 bytes each = \(36 \times 10^{15}\) bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
  - This is really a lot of space – for only the page table!!!

- The address space is sparse, i.e. has holes that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing
Recall: Page Table Discussion

• What needs to be switched on a context switch?
  – Page table pointer and limit
• What provides protection here?
  – Translation (per process) and dual-mode!
  – Can’t let process alter its own page table!
• 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 is really big?
    » Not all pages used all the time \(\Rightarrow\) would be nice to have working set of page table in memory
• Simple Page table is way too big!
  – Does it all need to be in memory?
  – How about multi-level paging?
  – or combining paging and segmentation
How to Structure a Page Table

• Page Table is a map (function) from VPN to PPN

  Virtual Address  Page Table  Physical Address

• Simple page table corresponds to a very large lookup table
  – VPN is index into table, each entry contains PPN

• What other map structures can you think of?
  – Trees?
  – Hash Tables?
Fix for sparse address space: The two-level page table

- Tree of Page Tables
  - “Magic” 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e., CR3)
- 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
Example: x86 classic 32-bit address translation

- Intel terminology: Top-level page-table called a “Page Directory”
  - With “Page Directory Entries”
- CR3 provides physical address of the page directory
  - This is what we have called the “PageTablePtr” in previous slides
  - Change in CR3 changes the whole translation table!
What is in a Page Table Entry (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 as prior slides (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>P</th>
<th>S</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- P: Present (same as “valid” bit in other architectures)
- W: Writeable
- U: User accessible
- PWT: Page write transparent: external cache write-through
- PCD: Page cache disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty (PTE only): page has been modified recently
- PS: Page Size: \( \text{PS}=1 \Rightarrow 4\text{MB} \) 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
Sharing with multilevel page tables

- Entire regions of the address space can be efficiently shared

```
Virtual Address:
  Virtual P1 index  Virtual P2 index  Offset

PageTablePtr

PageTablePtr'

10 bits  10 bits  12 bits
Physical Page #  Offset
```

4KB
Summary: Two-Level Paging

Virtual memory view

- stack
- heap
- data
- code

Page Table (level 1)

- stack
- heap
- data
- code

Page Tables (level 2)

- stack
- heap
- data

Physical memory view

- stack
- heap
- data
- code
Summary: Two-Level Paging

Virtual memory view

- stack
- heap
- data
- code

Page Table (level 1)

- 111 110 101 100 011 010 001 000
- null

Page Tables (level 2)

- 11 1101 1100 1011 1010
- 10 0111 0110
- 01 0111 0110
- 00 0110 0110

Physical memory view

- stack 1110 0000
- heap 1000 0000 (0x80)
- data 0010 0000
- code 0001 0000

Page Tables (level 2)

- 11 1101 1100 1011 1010
- 10 0111 0110
- 01 0111 0110
- 00 0110 0110

Page Tables (level 1)

- 111 110 101 100 011 010 001 000
- null

Page Tables (level 0)

- 11 1101 1100 1011 1010
- 10 0111 0110
- 01 0111 0110
- 00 0110 0110
Multi-level Translation: 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):

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

  | 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         |

Virtual Address: Virtual Page # Offset

Physical Address:

- 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 about Sharing (Complete Segment)?

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
<th>Page #</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
<td>page #0</td>
<td>VR</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
<td>page #1</td>
<td>VR</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
<td>page #2</td>
<td>VR,W</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
<td>page #3</td>
<td>VR,W</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
<td>page #5</td>
<td>VR,W</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process A:

Process B:
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, the 10b-10b-12b configuration keeps tables to exactly one page in size  
  – Two (or more, if >2 levels) lookups per reference  
    » Seems very expensive!
Recall: Dual-Mode Operation

- Can a process modify its own translation tables? NO!
  - If it could, could get access to all of physical memory (no protection!)
- 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 bit(s) in control register only accessible in Kernel mode
  - Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode (more in moment)
- Note that x86 model actually has more modes:
  - Traditionally, four “rings” representing priority; most OSes use only two:
    » Ring 0 ⇒ Kernel mode, Ring 3 ⇒ User mode
    » Called “Current Privilege Level” or CPL
  - Newer processors have additional mode for hypervisor (“Ring -1”)
- Certain operations restricted to Kernel mode:
  - Modifying page table base (CR3 in x86), and segment descriptor tables
    » Have to transition into Kernel mode before you can change them!
  - Also, all page-table pages must be mapped only in kernel mode
Making it real: x86 Memory model with segmentation (16/32-bit)

Segment Selector from instruction: \texttt{mov eax, gs(0x0)}

2-level page table in 10-10-12 bit address

Combined address is 32-bit “linear” Virtual address

First level called “directory”

Second level called “table”
X86 Segment Descriptors (32-bit Protected Mode)

• Segments are implicit in the instruction (e.g., code segments) or 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:
  – G/L selects between GDT and LDT tables (global vs local descriptor tables)
  – RPL: Requestor’s Privilege Level (RPL of CS ⇒ Current Privilege Level)

• Two registers: GDTR/LDTR hold pointers to global/local descriptor tables in memory
  – Descriptor format (64 bits):

    G: Granularity of segment [ Limit Size ] (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: Access requires Max(CPL,RPL)≤DPL
    S: System Segment (0: System, 1: code or data)
    Type: Code, Data, Segment

Segment Register

Segment selector [13 bits]  GL  RPL
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
    » RPL of Code Segment ⇒ CPL (Current Privilege Level)
  – Limited to 64K segments

• Modern use in 32-bit Mode:
  – Even though there is full segment functionality, segments are set up as “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:
      \[ \text{mov eax, gs(0x0)} \]

• 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 TLS
X86_64: Four-level page table!

48-bit Virtual Address:

- Virtual P1 index
- Virtual P2 index
- Virtual P3 index
- Virtual P4 index
- Offset

PageTablePtr

8 bytes

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

Physical Address:
(40-50 bits)

Physical Page #
12bit Offset
From x86_64 architecture specification

- All current x86 processor support a 64-bit operation
- 64-bit words (so ints are 8 bytes) but 48-bit addresses
• Larger page sizes (2MB, 1GB) make sense since memory is now cheap
  – Great for kernel, large libraries, etc.
  – Use limited primarily by internal fragmentation…
IA64: 64bit addresses: Six-level page table?!!

<table>
<thead>
<tr>
<th>64bit Virtual Address:</th>
<th>7 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual P1 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P2 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P3 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P4 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P5 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P6 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Offset</td>
</tr>
</tbody>
</table>

No!

Too slow
Too many almost-empty tables
Alternative: 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
    » PowerPC, UltraSPARC, IA64

- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table
# Address Translation Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Segmentation</td>
<td>Fast context switching (segment map maintained by CPU)</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (Single-Level)</td>
<td>No external fragmentation</td>
<td>Large table size (~ virtual memory)</td>
</tr>
<tr>
<td></td>
<td>Fast and easy allocation</td>
<td>Internal fragmentation</td>
</tr>
<tr>
<td>Paged Segmentation</td>
<td>Table size ~ # of pages in virtual memory</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Multi-Level Paging</td>
<td>Table size ~ # of pages in virtual memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fast and easy allocation</td>
<td></td>
</tr>
<tr>
<td>Inverted Page Table</td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No cache locality of page table</td>
</tr>
</tbody>
</table>
How is the Translation Accomplished?

- The MMU must translate virtual address to physical address on:
  - Every instruction fetch
  - Every load
  - Every store
- What does the MMU need to do to translate an address?
  - 1-level Page Table
    » Read PTE from memory, check valid, merge address
    » Set “accessed” bit in PTE, Set “dirty bit” on write
  - 2-level Page Table
    » Read and check first level
    » Read, check, and update PTE
  - N-level Page Table …
- MMU does Page Table Tree Traversal to translate each address
Where and What is the MMU?

- The processor requests READ Virtual-Address to memory system
  - Through the MMU to the cache (to the memory)
- Some time later, the memory system responds with the data stored at the physical address (resulting from virtual → physical) translation
  - Fast on a cache hit, slow on a miss
- So what is the MMU doing?
- On every reference (I-fetch, Load, Store) read (multiple levels of) page table entries to get physical frame or FAULT
  - Through the caches to the memory
  - Then read/write the physical location
Recall: CS61c Caching Concept

- **Cache**: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time = 
  \[(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})\]
Recall: In Machine Structures (eg. 61C) …

- Caching is the key to memory system performance

Average Memory Access Time (AMAT)

\[ \text{AMAT} = (\text{Hit Rate} \times \text{HitTime}) + (\text{Miss Rate} \times \text{MissTime}) \]

Where HitRate + MissRate = 1

HitRate = 90% \(\Rightarrow\) AMAT = \((0.9 \times 1) + (0.1 \times 101)\) = 11.1 ns

HitRate = 99% \(\Rightarrow\) AMAT = \((0.99 \times 1) + (0.01 \times 101)\) = 2.01 ns

**MissTime**\(_{L1}\) includes HitTime\(_{L1}\) + MissPenalty\(_{L1}\) \(\equiv\) HitTime\(_{L1}\) + AMAT\(_{L2}\)
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
Recall: Memory Hierarchy

- Caching: Take advantage of the principle of locality to:
  - Present the illusion of having as much memory as in the cheapest technology
  - Provide average speed similar to that offered by the fastest technology

![Memory Hierarchy Diagram]

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>100Bs</th>
<th>10kBs</th>
<th>100kBs</th>
<th>MBs</th>
<th>GBs</th>
<th>100GBs</th>
<th>TBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (ns)</td>
<td>0.3</td>
<td>1</td>
<td>3</td>
<td>10-30</td>
<td>100</td>
<td>100,000 (0.1 ms)</td>
<td>10,000,000 (10 ms)</td>
</tr>
</tbody>
</table>
How do we make Address Translation Fast?

- Cache results of recent translations!
  - Different from a traditional cache
  - Cache Page Table Entries using Virtual Page # as the key
Translation Look-Aside Buffer

• Record recent Virtual Page # to Physical Frame # translation
• If present, have the physical address without reading any of the page tables !!!
  – Even if the translation involved multiple levels
  – Caches the end-to-end result
• Was invented by Sir Maurice Wilkes – prior to caches
  – When you come up with a new concept, you get to name it!
  – People realized “if it’s good for page tables, why not the rest of the data in memory?”
• On a TLB miss, the page tables may be cached, so only go to memory when both miss
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 kind of Cache for TLB?

- Amount of Data = N * L * K
- Tag is portion of address that identifies line (w/o line offset)
- Write Policy (write-thru, write-back), Eviction Policy (LRU, …)

• Remember all those cache design parameters and trade-offs?
How might organization of TLB differ from that of a conventional instruction or data cache?

• Let’s do some review …
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?

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

![Diagram of Direct Mapped Cache]
Review: Set Associative Cache

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

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

- Example: Block 12 placed in 8 block cache

32-Block Address Space:

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

Set associative:
block 12 can go anywhere in set 0
(12 mod 4)

Fully associative:
block 12 can go anywhere
Which block should be replaced on a miss?

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

Miss rates for a workload:

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

• **Write through**: The information is written to both the block in the cache and to the block in the lower-level memory

• **Write back**: The information is written only to the block in the cache
  – Modified cache block is written to main memory only when it is replaced
  – Question is block clean or dirty?

• Pros and Cons of each?
  – WT:
    » **PRO**: read misses cannot result in writes
    » **CON**: Processor held up on writes unless writes buffered
  – WB:
    » **PRO**: repeated writes not sent to DRAM
      processor not held up on writes
    » **CON**: More complex
      Read miss may require writeback of dirty data
Questions about caches?

- How does operating system behavior affect cache performance?
- Switching threads?
- Switching contexts?
- Cache design? What addresses are used?
- What does our understanding of caches tell us about TLB organization?
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! (PT traversal)
  - Cost of Conflict (Miss Time) is high
  - Hit Time – dictated by clock cycle

- 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 (larger now)
  - Not very big, can support higher associativity

- Small TLBs 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

<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>EB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>E.A.</td>
<td>TLB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D-Cache</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

- Four different TLBs
  - Instruction TLB for 4K pages
    » 128 entries, 4-way set associative
  - Instruction TLB for large pages
    » 2 entries, fully associative
  - Data TLB for 4K pages
    » 128 entries, 4-way set associative
  - Data TLB for large pages
    » 8 entries, 4-way set associative
- All TLBs use LRU replacement policy
- Why different TLBs for instruction, data, and page sizes?
Intel Nahelem (2008)

- **L1 DTLB**
  - 64 entries for 4 K pages and
  - 32 entries for 2/4 M pages,

- **L1 ITLB**
  - 128 entries for 4 K pages using 4-way associativity and
  - 14 fully associative entries for 2/4 MiB pages

- unified 512-entry L2 TLB for 4 KiB pages, 4-way associative.
Current Intel x86 (Skylake, Cascade Lake)
Current Example: Memory Hierarchy

• Caches (all 64 B line size)
  – L1 I-Cache: 32 KiB/core, 8-way set assoc.
  – L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  – L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  – L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency

• TLB
  – L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    » 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  – L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    » 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations:
    » 4 entries; 4-way associative, 1G page translations:
  – L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    » 16 entries; 4-way set associative, 1 GiB page translations:
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!
  – Called “TLB Consistency”
Putting Everything Together: Address Translation

Virtual Address:
- Offset
- Virtual P1 index
- Virtual P2 index
- PageTablePtr

Page Table (1st level)
- Page Table (2nd level)

Physical Address:
- Physical Page #
- Offset

Physical Memory:

3/4/21

Crooks & Joseph CS162 © UCB Spring 2021
Lec 13.71
Putting Everything Together: TLB

Virtual Address:

- Offset
- Virtual P1 index
- Virtual P2 index

Physical Address:

- Physical Page #
- Offset

TLB:

- ...
Putting Everything Together: Cache
Page Fault

- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  - Causes a Fault / Trap
    » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the instruction
- Other Page Faults engage operating system to fix the situation and retry the instruction
  - Allocate an additional stack page, or
  - Make the page accessible - Copy on Write,
  - Bring page in from secondary storage to memory – demand paging
- Fundamental inversion of the hardware / software boundary
Next Up: What happens when ...

virtual address

physical address

page fault

update PT entry

load page from disk

retry

exception

scheduler

Process

instruction

Operating System

Page Fault Handler

MMU

PT

frame#

offset

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**
  - Use of hash-table to hold translation entries
  - Size of page table ~ size of physical memory rather than size of virtual memory
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 Lookaside Buffer” (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
  - TLB is logically in front of cache (need to overlap with cache access)

- Next Time: What to do on a page fault?