CSE 153
Design of Operating Systems
Fall 2016

Lecture 11: Paging/Virtual Memory

Some slides modified from originals by Dave O’hallaron
Today

● Address spaces
● VM as a tool for caching
● VM as a tool for memory management
● VM as a tool for memory protection
● Address translation
A System Using Physical Addressing

- Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames.
A System Using Virtual Addressing

- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science
Address Spaces

- **Linear address space**: Ordered set of contiguous non-negative integer addresses:
  \[ \{0, 1, 2, 3 \ldots \} \]

- **Virtual address space**: Set of \( N = 2^n \) virtual addresses
  \[ \{0, 1, 2, 3, \ldots, N-1\} \]

- **Physical address space**: Set of \( M = 2^m \) physical addresses
  \[ \{0, 1, 2, 3, \ldots, M-1\} \]

- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory: one physical address, one (or more) virtual addresses
Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space

- Simplifies memory management
  - Each process gets the same uniform linear address space

- Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
**Virtual memory** is an array of N contiguous bytes stored on disk.

- The contents of the array on disk are cached in *physical memory (DRAM cache)*
  - These cache blocks are called *pages* (size is $P = 2^p$ bytes)

![Diagram of virtual memory and physical memory with caching]

- Virtual pages (VPs) stored on disk
- Physical pages (PPs) cached in DRAM
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM

- Consequences
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    » Any VP can be placed in any PP
    » Requires a “large” mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    » Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through
A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages.

- Per-process kernel data structure in DRAM

### Page Tables Diagram

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Physical page number or disk address**

<table>
<thead>
<tr>
<th>Physical memory (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
</tr>
<tr>
<td>VP 3</td>
</tr>
<tr>
<td>VP 4</td>
</tr>
<tr>
<td>VP 5</td>
</tr>
<tr>
<td>VP 6</td>
</tr>
<tr>
<td>VP 7</td>
</tr>
</tbody>
</table>

**Memory resident page table (DRAM)**

<table>
<thead>
<tr>
<th>Memory resident page table (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
</tr>
<tr>
<td>PP 1</td>
</tr>
<tr>
<td>PP 2</td>
</tr>
</tbody>
</table>

**Virtual memory (disk)**
Page Hit

- **Page hit**: reference to VM word that is in physical memory (DRAM cache hit)
- **Page fault**: reference to VM word that is not in physical memory (DRAM cache miss)

### Diagram

**Virtual address**

<table>
<thead>
<tr>
<th>Valid</th>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**Physical page number or disk address**

**Memory resident page table (DRAM)**

**Physical memory (DRAM)**

- PP 0
- VP 1
- VP 2
- VP 7
- PP 3
- VP 4

**Virtual memory (disk)**

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
Handling Page Fault

- Page miss causes page fault (an exception)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!
Locality to the Rescue!

- Virtual memory works because of locality

- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets

- If (working set size < main memory size)
  - Good performance for one process after compulsory misses

- If (SUM(working set sizes) > main memory size )
  - *Thrashing*: Performance meltdown where pages are swapped (copied) in and out continuously
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
**VM as a Tool for Memory Management**

- **Key idea:** each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well chosen mappings simplify memory allocation and management

![Virtual Address Space Diagram]

---

**Diagram Explanation**

- **Virtual Address Space for Process 1:**
  - VP 1
  - VP 2
  - ... (N-1)

- **Virtual Address Space for Process 2:**
  - VP 1
  - VP 2
  - ... (N-1)

- **Address Translation:**
  - Mappings scatter addresses through physical memory (DRAM)
  - Well chosen mappings simplify memory allocation and management

- **Physical Address Space (DRAM):**
  - PP 2
  - PP 6
  - PP 8
  - ... (M-1)
VM as a Tool for Memory Management

- **Memory allocation**
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- **Sharing code and data among processes**
  - Map virtual pages to the same physical page (here: PP 6)
Sharing

• Can map shared memory at same or different virtual addresses in each process’ address space
  ◆ Different:
    » 10\textsuperscript{th} virtual page in P1 and 7\textsuperscript{th} virtual page in P2 correspond to the 2\textsuperscript{nd} physical page
    » Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  ◆ Same:
    » 2\textsuperscript{nd} physical page corresponds to the 10\textsuperscript{th} virtual page in both P1 and P2
    » Less flexible, but shared pointers are valid
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
- How does this help fork()?
Execution of fork()

Parent process’s page table

| Page 1 |
| Page 2 |

Physical Memory

Child process’s page table

| Page 1 |
| Page 2 |
fork() with Copy on Write

When either process modifies Page 1, page fault handler allocates new page and updates PTE in child process.

Protection bits set to prevent either process from writing to any page.
Simplifying Linking and Loading

**Linking**
- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

**Loading**
- `execve()` allocates virtual pages for `.text` and `.data` sections = creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

---

`Kernel virtual memory`

- `User stack (created at runtime)`
- `Memory-mapped region for shared libraries`
- `Run-time heap (created by malloc)`
- `Read/write segment (.data, .bss)`
- `Read-only segment (.init, .text, .rodata)`
- `Unused`

```
Memory invisible to user code
%esp (stack pointer)
brk

Loaded from the executable file
```

```
Kernel	virtual	memory
Memory-mapped	region	for	shared	libraries
Run-Sme	heap	(created	by	malloc)
User	stack	(created	at	runtime)
Unused
```
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM Address Translation

- **Virtual Address Space**
  - $V = \{0, 1, \ldots, N-1\}$

- **Physical Address Space**
  - $P = \{0, 1, \ldots, M-1\}$

- **Address Translation**
  - $MAP: V \rightarrow P \cup \{\emptyset\}$
  - For virtual address $a$:
    - $MAP(a) = a'$ if data at virtual address $a$ is at physical address $a'$ in $P$
    - $MAP(a) = \emptyset$ if data at virtual address $a$ is not in physical memory
      - Either invalid or stored on disk
Summary of Address Translation Symbols

- **Basic Parameters**
  - \( N = 2^n \): Number of addresses in virtual address space
  - \( M = 2^m \): Number of addresses in physical address space
  - \( P = 2^p \): Page size (bytes)

- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number

- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN**: Physical page number
  - **CO**: Byte offset within cache line
  - **CI**: Cache index
  - **CT**: Cache tag
Address Translation With a Page Table

- Virtual address
  - Virtual page number (VPN)
  - Virtual page offset (VPO)

- Page table
  - Page table base register (PTBR)
  - Page table address for process
  - Valid
    - Valid bit = 0: page not in memory (page fault)
  - Physical page number (PPN)

- Physical address
  - Physical page number (PPN)
  - Physical page offset (PPO)
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Integrating VM and Cache

CPU Chip

VA

CPU

MMU

PTE

PTEA

PA

L1 cache

Memory

Data

PTEA hit

PTEA miss

PA miss

PA hit

Data

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

- Solution: *Translation Lookaside Buffer* (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Reloading the TLB

- If the TLB does not have mapping, two possibilities:
  1. MMU loads PTE from page table in memory
     - Hardware managed TLB, OS not involved in this step
     - OS has already set up the page tables so that the hardware can access it directly
  2. Trap to the OS
     - Software managed TLB, OS intervenes at this point
     - OS does lookup in page table, loads PTE into TLB
     - OS returns from exception, TLB continues

- A machine will only support one method or the other
- At this point, there is a PTE for the address in the TLB
Page Faults

- PTE can indicate a protection fault
  - Read/write/execute – operation not permitted on page
  - Invalid – virtual page not allocated, or page not in physical memory

- TLB traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
  - Invalid
    - Virtual page not allocated in address space
      - OS sends fault to process (e.g., segmentation fault)
    - Page not in physical memory
      - OS allocates frame, reads from disk, maps PTE to physical frame
Multi-Level Page Tables

- Suppose:
  - 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE

- Problem:
  - Would need a 512 GB page table!
    - $2^{48} \times 2^{-12} \times 2^3 = 2^{39}$ bytes

- Common solution:
  - Multi-level page tables
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table (always memory resident)
    - Level 2 table: each PTE points to a page (paged in and out like any other data)
A Two-Level Page Table Hierarchy

Level 1
- page table
  - PTE 0
  - PTE 1
  - PTE 2 (null)
  - PTE 3 (null)
  - PTE 4 (null)
  - PTE 5 (null)
  - PTE 6 (null)
  - PTE 7 (null)
  - PTE 8
  - (1K - 9) null PTEs

Level 2
- page tables
  - PTE 0
  - ... PTE 1023

Virtual memory
  - VP 0
  - ... VP 1023
  - GAP
  - ... VP 2047
  - 2K allocated VM pages for code and data
  - 6K unallocated VM pages
  - 1023 unallocated pages

32 bit addresses, 4KB pages, 4-byte PTEs

1 allocated VM page for the stack
TLB Misses (2)

Note that:

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
  - Assuming page tables are in OS virtual address space
  - Not a problem if tables are in physical memory
  - Yes, this is a complicated situation!

- When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
    » These faults are handled quickly, just read PTE from the page table in memory and load into TLB
  - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
    » Becomes a page fault…
Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes

```
13 12 11 10  9  8  7  6  5  4  3  2  1  0

VPN  VPO
```

Virtual Page Number Virtual Page Offset

```
11 10  9  8  7  6  5  4  3  2  1  0

PPN  PPO
```

Physical Page Number Physical Page Offset
Simple Memory System
Page Table

Only show first 16 entries (out of 256)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple Memory System TLB

- 16 entries
- 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

<table>
<thead>
<tr>
<th>PPO</th>
<th>PPN</th>
<th>CO</th>
<th>CI</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Address Translation

Example #1

Virtual Address: \(0x03D4\)

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __        PPN: ____

Physical Address

CO ___ CI___ CT ____ Hit? __              Byte: ____

VPN \(0x0F\)     TLBI \(0x3\)   TLBT \(0x03\)   TLB Hit? Y   Page Fault? N   PPN: \(0x0D\)

CO __0__ CI.0x5   CT 0x0D   Hit? Y   Byte: 0x36
Address Translation
Example #2

Virtual Address: 0xB8F

VPN ___
祝贺 ___
TLBI ___
TLBT __
TLB Hit? ___
Page Fault? __
PPN: ___

Physical Address

CO ___
CI___
CT ___
Hit? ___
Byte: ___
Address Translation
Example #3

Virtual Address: 0x0020

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address

CT ___ CI ___ CO ___
11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 0 0 1 0 0 0 0 0 0

CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem
**Intel Core i7 Memory System**

- **L1 d-cache**: 32 KB, 8-way
- **L1 i-cache**: 32 KB, 8-way
- **L2 unified cache**: 256 KB, 8-way
- **L1 d-TLB**: 64 entries, 4-way
- **L1 i-TLB**: 128 entries, 4-way
- **L2 unified TLB**: 512 entries, 4-way
- **L3 unified cache**: 8 MB, 16-way (shared by all cores)
- **DDR3 Memory controller**: 3 x 64 bit @ 10.66 GB/s, 32 GB/s total (shared by all cores)
- **Main memory**

**Processor package**

**Core x4**

- ** Registers**
- **Instruction fetch**
- **MMU (addr translation)**
- **QuickPath interconnect**: 4 links @ 25.6 GB/s each

To other cores

To I/O bridge
End-to-end Core i7 Address Translation

Virtual address (VA)

36 12

VPN VPO

32 4

TLBT TLBI

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

9 9 9 9

TLB hit

PTE PTE PTE PTE

Page tables

CR3

32/64

Result

L1 hit

L1 d-cache (64 sets, 8 lines/set)

L1 miss

Physical address (PA)

PPN PPO

L2, L3, and main memory

CT CI CO

40 6 6

## Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page table physical base address</td>
<td>Unused</td>
<td>G</td>
<td>PS</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available for OS (page table location on disk) | P=0

### Each entry references a 4K child page table

- **P**: Child page table present in physical memory (1) or not (0).
- **R/W**: Read-only or read-write access access permission for all reachable pages.
- **U/S**: user or supervisor (kernel) mode access permission for all reachable pages.
- **WT**: Write-through or write-back cache policy for the child page table.
- **CD**: Caching disabled or enabled for the child page table.
- **A**: Reference bit (set by MMU on reads and writes, cleared by software).
- **PS**: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).
- **G**: Global page (don’t evict from TLB on task switch)

**Page table physical base address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
Summary

- Programmer’s view of virtual memory
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    » Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions
Summary (2)

Paging mechanisms:

- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)
  - Demand paged virtual memory (space)

- Recap address translation

- Advanced Functionality
  - Sharing memory
  - Copy on Write
  - Mapped files

Next time: Paging policies
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open, read into buffer, operate on buffer, …”
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    - Another use of the dirty bit in PTE