Computer Systems
CSE 410 Autumn 2013
12 – Virtual Memory
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer system:

Virtual Memory Overview

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Virtual Memory (VM)

- Overview and motivation
- Indirection
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example
Processes

Definition: A process is an instance of a running program
- One of the most important ideas in computer science
- Not the same as “program” or “processor”

Process provides each program with two key abstractions:
- Logical control flow
  - Each process seems to have exclusive use of the CPU
- Private virtual address space
  - Each process seems to have exclusive use of main memory

How are these illusions maintained?
- Process executions interleaved (multi-tasking) – last section
- Address spaces managed by virtual memory system – this section!
Virtual Memory (Previous Lectures)

- Programs refer to virtual memory addresses
  - `movl (%ecx),%eax`
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - System provides address space private to particular “process”

- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space

- *What problems does virtual memory solve?*
Problem 1: How Does Everything Fit?

64-bit addresses: 16 Exabyte

Physical main memory: Few Gigabytes

And there are many processes ....
Problem 2: Memory Management

What goes where?

Process 1
Process 2
Process 3
...
Process n

stack
heap
.text
.data
...

Physical main memory

Virtual Memory Overview
Problem 3: How To Protect

Process i

Process j

Problem 4: How To Share?

Process i

Process j
Virtual Memory (VM)

- Overview and motivation
- **Indirection**
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example
How would you solve those problems?

- Fitting a huge memory into a tiny physical memory
- Managing the memory spaces of multiple processes
- Protecting processing from stepping on each other’s memory
- Allowing processes to share common parts of memory
Indirection

- “Any problem in computer science can be solved by adding another level of indirection”

- Without Indirection

- With Indirection
Indirection

- Indirection: the ability to reference something using a name, reference, or container instead the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.

  - Without Indirection
    - Name
    - Thing

  - With Indirection
    - Name
    - Thing

- Examples:
  Domain Name Service (DNS) name->IP address, phone system (e.g., cell phone number portability), snail mail (e.g., mail forwarding), 911 (routed to local office), DHCP, call centers that route calls to available operators, etc.
Solution: Level Of Indirection

- Each process gets its own private virtual address space
- Solves the previous problems
Address Spaces

- **Virtual address space:** Set of $N = 2^n$ virtual addresses
  
  \{0, 1, 2, 3, ..., N-1\}

- **Physical address space:** Set of $M = 2^m$ physical addresses ($n > m$)
  
  \{0, 1, 2, 3, ..., M-1\}

- Every byte in main memory:
  one physical address; zero, one, or more virtual addresses
A virtual address can be mapped to either physical memory or disk.
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 desktops, laptops, servers
- One of the great ideas in computer science
Virtual Memory (VM)

- Overview and motivation
- Indirection
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example
VM and the Memory Hierarchy

- Think of virtual memory as an array of $N = 2^n$ contiguous bytes stored *on a disk*

- Then physical main memory (DRAM) is used as a *cache* for the virtual memory array
  - The cache blocks are called *pages* (size is $P = 2^p$ bytes)

```
Virtual memory
VP 0
  Unallocated
  Cached
  Uncached
VP 1
  Unallocated
  Cached
  Uncached
VP 2^{n-p-1}
  Unallocated
  Cached
  Uncached

Physical memory
PP 0
  Empty
PP 1
  Empty
PP 2^{m-p-1}
  Empty

Virtual pages (VPs) stored on disk
Physical pages (PPs) cached in DRAM
```
Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

- L1 I-cache
- L1 D-cache
- ~4 MB L2 unified cache
- ~4 GB Main Memory
- ~500 GB Disk

Throughput:
- L1 I-cache: 16 B/cycle
- L1 D-cache: 8 B/cycle
- ~4 MB L2 cache: 2 B/cycle
- ~4 GB Main Memory: 1 B/30 cycles

Latency:
- L1 I-cache: 3 cycles
- L1 D-cache: 14 cycles
- ~4 MB L2 cache: 100 cycles
- ~4 GB Main Memory: millions

Miss penalty (latency):
- L1 I-cache: 33x
- L1 D-cache: 10,000x
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
    - (for first byte; faster for next byte)

- Consequences?
  - Block size?
  - Associativity?
  - Write-through or write-back?
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
    - (for first byte; faster for next byte)

- 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
Indexing into the “DRAM Cache”

How do we perform the VA -> PA translation?
A *page table* (PT) is an array of *page table entries* (PTEs) that maps virtual pages to physical pages.

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Physical page number or disk address</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Physical memory (DRAM)**
- PP 0: VP 1, VP 2, VP 7, VP 4
- PP 3: VP 1, VP 2, VP 3, VP 4, VP 6, VP 7

**Virtual memory (disk)**
- VP 1, VP 2, VP 3, VP 4, VP 6, VP 7

**Memory resident page table (DRAM)**
- VP 1, VP 2

**Virtual Memory as Cache**

*How many page tables are in the system?*

One per process
Address Translation With a Page Table

In most cases, the hardware (the MMU) can perform this translation on its own, without software assistance.
**Page Hit**

- **Page hit:** reference to VM byte that is in physical memory

![Diagram](Virtual MEMORY AS CACHE.png)

- **Virtual address**
  - **PTE 0**
  - **PTE 7**

- **Physical page number or disk address**
  - **Valid**
  - **null**
  - **1**
  - **0**
  - **1**
  - **0**

- **Memory resident page table (DRAM)**
  - **Physical memory (DRAM)**
    - **VP 1**
    - **VP 2**
    - **VP 7**
    - **VP 4**
    - **VP 3**
    - **VP 6**
    - **VP 7**

- **Virtual memory (disk)**

**Virtual Memory as Cache**
Page Fault

- **Page fault:** reference to VM byte that is **NOT** in physical memory

What happens when a page fault occurs?

Virtual Memory as Cache
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}

80483b7:    c7 05 10 9d 04 08 0d  movl $0xd,0x8049d10
```

- Page handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
- Successful on second try
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!
Why does it work? Locality

- Virtual memory works well because of locality
  - Same reason that L1 / L2 / L3 caches work

- The set of virtual pages that a program is “actively” accessing at any point in time is called its **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
Virtual Memory (VM)

- Overview and motivation
- Indirection
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example
VM for Managing Multiple Processes

- **Key abstraction:** each process has its own virtual address space
  - It can view memory as a *simple linear array*
- **With virtual memory,** this simple linear virtual address space need not be contiguous in physical memory
  - Process needs to store data in another VP? Just map it to *any* PP!

---

**Virtual Address Space for Process 1:**
- 0
- VP 1
- VP 2
- ...
- N-1

**Virtual Address Space for Process 2:**
- 0
- VP 1
- VP 2
- ...
- N-1

**Address translation**
- 0
- PP 2
- PP 6
- PP 8
- ...
- M-1

**Physical Address Space (DRAM):**
- (e.g., read-only library code)
VM for Protection and Sharing

The mapping of VPs to PPs provides a simple mechanism for protecting memory and for sharing memory btw. processes

- Sharing: just map virtual pages in separate address spaces to the same physical page (here: PP 6)
- Protection: process simply can’t access physical pages it doesn’t have a mapping for (here: Process 2 can’t access PP 2)
Memory Protection Within a Single Process

- Extend PTEs with permission bits
- MMU checks these permission bits on every memory access
  - If violated, raises exception and OS sends SIGSEGV signal to process

### Process i:

<table>
<thead>
<tr>
<th>Valid</th>
<th>SUP</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

### Process j:

<table>
<thead>
<tr>
<th>Valid</th>
<th>SUP</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Address Translation
Address Translation: Page Hit

1) Processor sends virtual address to MMU (*memory management unit*)
2-3) MMU fetches PTE from page table in cache/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 cache/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
Hmm... Translation Sounds Slow!

- The MMU accesses memory *twice*: once to first get the PTE for translation, and then again for the actual memory request from the CPU
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- *What can we do to make this faster?*
Speeding up Translation with a TLB

Solution: add another cache!

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
  - Modern Intel processors: 128 or 256 entries in TLB
TLB Hit

A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare
Virtual Memory (VM)

- Overview and motivation
- Indirection
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example
Simple Memory System Example

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

Virtual Page Number (VPN) and Virtual Page Offset (VPO)

Physical Page Number (PPN) and Physical Page Offset (PPO)
Simple Memory System Page Table

- Only showing 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

**Virtual Memory Example**

<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

Virtual Memory Example
Current state of caches/tables

TLB

<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>

Page table

<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>

Cache

<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>

Virtual Memory Example
Address Translation Example #1

Virtual Address: 0x03D4

Virtual Memory Example

Physical Address
Address Translation Example #2

Virtual Address: 0x0B8F

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

VPN: 0x2E TLBI: 2 TLB Hit: N Page Fault: ? PPN: TBD

Physical Address

<table>
<thead>
<tr>
<th>CT</th>
<th>CI</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CO:  ____ CI:  ____ CT:  ____ Hit?:  ____ Byte:  ____
Address Translation Example #3

Virtual Address: 0x0020

Virtual Memory Example

Byte: Mem
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 sharing
  - Simplifies protection by providing a convenient interpositioning point to check permissions
Memory System Summary

- **L1/L2 Memory Cache**
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware

- **Virtual Memory**
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Software
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)