Introduction to Computer Systems
15-213, fall 2009
16th Lecture, Oct. 21st

Instructors:
Majd Sakr and Khaled Harras
Today

- Virtual memory (VM)
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
  - Allocation, multi-level page tables
Virtual Memory (Previous Lectures)

- Programs refer to virtual memory addresses
  - `movl (%ecx), %eax`
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Actually implemented with hierarchy of different memory types
  - 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

- **But why virtual memory?**
- **Why not physical memory?**
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

Physical main memory

Process 1
Process 2
Process 3
...
Process n

X

stack
heap
.text
.data
...

What goes where?
Problem 3: How To Protect

Physical main memory

Process i

Process j

Problem 4: How To Share?

Physical main memory

Process i

Process j
Solution: Level Of Indirection

- Each process gets its own private memory space
- Solves the previous problems
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
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, workstations
- One of the great ideas in computer science
- **MMU checks the cache**
Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed

- **Simplifies memory management for programmers**
  - Each process gets the same full, private linear address space

- **Isolates address spaces**
  - One process can’t interfere with another’s memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions
Today

- Virtual memory (VM)
  - Overview and motivation
  - **VM as tool for caching**
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
  - Allocation, multi-level page tables
VM as a Tool for Caching

- **Virtual memory**: array of $N = 2^n$ contiguous bytes
  - think of the array (allocated part) as being stored on disk
- Physical main memory (DRAM) = cache for allocated virtual memory
- Blocks are called pages; size = $2^p$

![Diagram showing virtual and physical memory mapping]

Virtual pages (VP's) stored on disk

Physical pages (PP's) cached in DRAM
Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

Throughput: 16 B/cycle
Latency: 3 cycles

Miss penalty (latency): 30x

L1 I-cache
32 KB

L1 D-cache

~4 MB

L2 unified cache

~4 GB

Main Memory

1 B/30 cycles
millions

Miss penalty (latency): 10,000x

~500 GB

Disk
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
A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs

- Per-process kernel data structure in DRAM
Address Translation With a Page Table

Virtual address

Page table

Valid bit = 0: page not in memory (page fault)

Physical address

Valid Physical page number (PPN)
**Page Hit**

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

- *Page miss:* reference to VM word that is not in physical memory

![Diagram of memory resident page table and physical memory](image)

- **Virtual address**
  - **Physical page number or disk address**
  - **Memory resident page table (DRAM)**
  - **Physical memory (DRAM)**
    - VP 1
    - VP 2
    - VP 7
    - 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!
Why does it work? Locality

- 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

- Virtual memory (VM)
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
  - Allocation, multi-level page tables
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 for Process 1:
0
<table>
<thead>
<tr>
<th>VP 1</th>
<th>VP 2</th>
</tr>
</thead>
</table>
N-1

Virtual Address Space for Process 2:
0
<table>
<thead>
<tr>
<th>VP 1</th>
<th>VP 2</th>
</tr>
</thead>
</table>
N-1

Address translation

Physical Address Space (DRAM)
0
<table>
<thead>
<tr>
<th>PP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 6</td>
</tr>
<tr>
<td>PP 8</td>
</tr>
</tbody>
</table>
M-1

(e.g., read-only library code)
```
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)
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
Today

- **Virtual memory (VM)**
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - **VM as tool for memory protection**
  - Address translation
  - Allocation, multi-level page tables
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)

### Process i:

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

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

<table>
<thead>
<tr>
<th>VP 2:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
<td></td>
</tr>
</tbody>
</table>

### Process j:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VP 1:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VP 2:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
<td></td>
</tr>
</tbody>
</table>

### Physical Address Space

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
Today

- **Virtual memory (VM)**
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - *Address translation*
  - Allocation, multi-level page tables
Address Translation: Page Hit

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
Address Translation: Page Fault

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
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 1-cycle 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 add’l memory access (the PTE)
Fortunately, TLB misses are rare
Simple Memory System Example

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

![Diagram of memory addressing](image)
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

```
Idx | Tag | Valid | B0 | B1 | B2 | B3
---|-----|-------|----|----|----|----
 0  | 19  | 1     | 99 | 11 | 23 | 11
 1  | 15  | 0     | -- | -- | -- | --
 2  | 1B  | 1     | 00 | 02 | 04 | 08
 3  | 36  | 0     | -- | -- | -- | --
 4  | 32  | 1     | 43 | 6D | 8F | 09
 5  | 0D  | 1     | 36 | 72 | F0 | 1D
 6  | 31  | 0     | -- | -- | -- | --
 7  | 16  | 1     | 11 | C2 | DF | 03

Idx | Tag | Valid | B0 | B1 | B2 | B2 | B3
---|-----|-------|----|----|----|----|----
 8  | 24  | 1     | 3A | 00 | 51 | 89
 9  | 2D  | 0     | -- | -- | -- | -- | --
 A  | 2D  | 1     | 93 | 15 | DA | 3B
 B  | 0B  | 0     | -- | -- | -- | -- | --
 C  | 12  | 0     | -- | -- | -- | -- | --
 D  | 16  | 1     | 04 | 96 | 34 | 15
 E  | 13  | 1     | 83 | 77 | 1B | D3
 F  | 14  | 0     | -- | -- | -- | -- | --
```
Address Translation Example #1

Virtual Address: 0x03D4

Physical Address:

Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

Physical Address

VPN: 0x2E TLBI: 2 TLBT: 0x0B TLB Hit? N Page Fault? Y PPN: TBD

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

Virtual Address: 0x0020

Physical Address

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

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

Physical Address

CO 0  CI 0x8  CT 0x28  Hit? N  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 programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions
Today

- Virtual memory (VM)
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
  - Allocation, multi-level page tables
Allocating Virtual Pages

- Example: Allocating VP5
Allocating Virtual Pages

- Example: Allocating VP 5
- Kernel allocates VP 5 on disk and points PTE 5 to it

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

- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Multi-Level Page Tables

- **Given:**
  - 4KB \( (2^{12}) \) page size
  - 48-bit address space
  - 4-byte PTE

- **Problem:**
  - Would need a 256 GB page table!
    - \( 2^{48} \times 2^{-12} \times 2^2 = 2^{38} \) bytes

- **Common solution**
  - Multi-level page tables
  - Example: 2-level page table
  - Level 1 table: each PTE points to a page table
  - Level 2 table: each PTE points to a page
    - (paged in and out like other data)
  - Level 1 table stays in memory
  - Level 2 tables paged in and out
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
  - ... (null)
  - PTE 1023

Virtual memory
  - VP 0
  - PTE 0
  - ... (null)
  - PTE 1023
  - VP 1023
  - PTE 1023
  - VP 1024
  - PTE 1023
  - VP 2047
  - Gap
  - 1023 unallocated pages
  - VP 9215
  - 1023 unallocated pages
  - 1 allocated VM page for the stack
  - 6K unallocated VM pages
  - 2K allocated VM pages for code and data
Translating with a k-level Page Table

Virtual Address

Physical Address