Lecture 18: Virtual memory - theory

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
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

- Today: Virtual Memory in the abstract

- Next time: How it really works
Virtual Memory
(Up until now)

- 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: Protection

Physical main memory

Process i

Process j

Problem 4: Sharing

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
System Using Physical Addressing

- [Still] used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
• 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
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 = $2^p$
Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

- L1 I-cache: 32 KB throughputs: 16 B/cycle, latency: 3 cycles
- L1 D-cache: 32 KB throughputs: 8 B/cycle, latency: 14 cycles
- L2 unified cache: ~4 MB throughputs: 2 B/cycle, latency: 100 cycles

Throughput:
- L1 I-cache: 16 B/cycle
- L1 D-cache: 8 B/cycle
- L2 unified cache: 2 B/cycle
- Main Memory: 1 B/30 cycles

Latency:
- L1 I-cache: 3 cycles
- L1 D-cache: 14 cycles
- L2 unified cache: 100 cycles
- Main Memory: millions
- Disk: ~500 GB

Miss penalty (latency): 30x
Miss penalty (latency): 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
  – 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
Address Translation: Page Tables

- 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

```
<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<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

Virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7

Memory resident page table (DRAM)
Address Translation With a Page Table

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual page number (VPN)</td>
<td>Page table address for process</td>
</tr>
<tr>
<td>Virtual page offset (VPO)</td>
<td>Valid</td>
</tr>
<tr>
<td>Physical page number (PPN)</td>
<td>Page table address for process</td>
</tr>
<tr>
<td>Valid bit = 0: page not in memory (page fault)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical address</th>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number (PPN)</td>
<td>Virtual address</td>
</tr>
<tr>
<td>Physical page offset (PPO)</td>
<td>Physical address</td>
</tr>
</tbody>
</table>
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
Handling a page fault

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

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

- 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
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:

Virtual Address Space for Process 2:

Address translation

Physical Address Space (DRAM)

(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)

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:
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

```
Memory invisible to user code
%esp (stack pointer)
brk
Loaded from the executable file
```

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

```
Kernel virtual memory
User stack (created at runtime)
```

```
Memory
0xc0000000
0x40000000
0x08048000
0
```
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
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></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>

### 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></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

• 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
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 in L1 still requires an extra 1 (or 2)-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 showing virtual and physical page 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

<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>18</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: \textbf{0x03D4}

\begin{center}
\begin{tabular}{cccccccccccc}
13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{tabular}
\end{center}

VPN \textbf{0x0F} \quad TLBI \textbf{3} \quad TLBT \textbf{0x03} \quad TLB Hit? \textbf{Y} \quad Page Fault? \textbf{N} \quad PPN: \textbf{0x0D}

Physical Address

\begin{center}
\begin{tabular}{cccccccccccc}
11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{tabular}
\end{center}

CO \textbf{0} \quad CI \textbf{0x5} \quad CT \textbf{0x0D} \quad Hit? \textbf{Y} \quad Byte: \textbf{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

CO  PPN  Hit?  Byte:

PPN  PPO
Allocating Virtual Pages

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

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Valid</strong></td>
<td><strong>Physical page number or disk address</strong></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>VP 6</td>
</tr>
<tr>
<td>1</td>
<td>VP 2</td>
</tr>
<tr>
<td>1</td>
<td>VP 7</td>
</tr>
<tr>
<td>0</td>
<td>VP 1</td>
</tr>
<tr>
<td>0</td>
<td>VP 3</td>
</tr>
<tr>
<td>0</td>
<td>VP 4</td>
</tr>
<tr>
<td>0</td>
<td>VP 5</td>
</tr>
<tr>
<td>1</td>
<td>VP 3</td>
</tr>
</tbody>
</table>

- Physical memory (DRAM)
  - PP 0
  - VP 1
  - VP 2
  - VP 7
  - VP 3

- Virtual memory (disk)
  - VP 1
  - VP 2
  - VP 3
  - VP 4
  - VP 5
  - VP 6
  - VP 7
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)
Further Reading

• Intel TLBs:
  – Application Note:
    “TLBs, Paging-Structure Caches, and Their Invalidation”,
    April 2007

• More next time...