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

Problem 3: Protection

Problem 4: Sharing
**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

**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
  - 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 = $2^p$

![Virtual memory diagram]

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

![DRAM cache organization diagram]

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

![Page table diagram]

Page Hit

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

![Page hit diagram]
**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)

**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:

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

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)
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 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

TLB Hit
A TLB hit eliminates a memory access

TLB Miss
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
  - 32-bit physical address
  - Page size = 64 bytes
### 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>12</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>08</td>
<td>03</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>04</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>05</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>06</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>07</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>08</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0A</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0B</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0C</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0D</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>0E</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0F</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>03</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>04</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>05</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>06</td>
<td>1</td>
</tr>
</tbody>
</table>

### Simple Memory System TLB

- 16 entries
- 4-way associative

![TLB Diagram](image)

### Simple Memory System Cache

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

![Cache Diagram](image)

### Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

![Address Translation Example](image)

### Address Translation Example #2

Virtual Address: 0x0B8F

Physical Address

![Address Translation Example](image)

### Allocating Virtual Pages

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

![Page Allocation Diagram](image)
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…