Virtual Memory

Today
- Motivations for VM
- Address translation
- Accelerating translation with TLBs

Next time
- Dynamic memory allocation and memory bugs
The most natural way to access memory – Addresses generated by the CPU correspond directly to bytes in physical memory

Used in simple systems like early PCs and embedded microcontrollers (e.g. in cars and elevators)
A system with virtual addressing

- Modern processors use virtual addresses
- Dedicated hardware (*memory management unit*) converts virtual to physical addresses via OS-managed lookup table (page table)
Motivations for virtual memory

- Use physical DRAM as a cache for the disk
  - Address space of a process can exceed physical memory size
  - Sum of address spaces of multiple processes can exceed physical memory

- Simplify memory management
  - Multiple processes resident in main memory.
    - Each process with its own address space
  - Only “active” code and data is actually in memory
    - Allocate more memory to process as needed.

- Provide protection
  - One process can’t interfere with another.
    - because they operate in different address spaces.
  - User process cannot access privileged information
    - different sections of address spaces have different permissions.
Levels in memory hierarchy

<table>
<thead>
<tr>
<th>Register</th>
<th>Cache</th>
<th>Memory</th>
<th>Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:</td>
<td>32 B</td>
<td>1024 MB</td>
<td>100 GB</td>
</tr>
<tr>
<td>speed:</td>
<td>1 ns</td>
<td>30 ns</td>
<td>8 ms</td>
</tr>
<tr>
<td>$/Mbyte:</td>
<td>$125/MB</td>
<td>$0.20/MB</td>
<td>$0.001/MB</td>
</tr>
<tr>
<td>line size:</td>
<td>8 B</td>
<td>4 KB</td>
<td></td>
</tr>
</tbody>
</table>

10x slower 100,000x slower

larger, slower, cheaper
Motivation: DRAM a “cache” for disk

- Full address space is quite large:
  - 32-bit addresses: ~4 Gigabytes (4 billion) bytes
  - 64-bit addresses: ~16 Exabytes (16 quintillion) bytes

- Disk storage is ~300X cheaper than DRAM storage
  - 80 GB of DRAM: ~ $33,000
  - 80 GB of disk: ~ $110

- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk
DRAM vs. SRAM as a “cache”

- DRAM vs. disk is more extreme than SRAM vs. DRAM
  - Access latencies:
    - DRAM ~10X slower than SRAM
    - Disk ~100,000X slower than DRAM
  - Importance of exploiting spatial locality:
    - First byte is ~100,000X slower than successive bytes on disk
      - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
  - Bottom line:
    - Design decisions made for DRAM caches driven by enormous cost of misses
Impact of properties on design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?
- Line size? Large, since disk better at transferring large blocks
- Associativity? High, to minimize miss rate
- Write through or write back?
  - Write back, since can’t afford to perform small writes to disk

What would the impact of these choices be on:
- Miss rate: Extremely low. << 1%
- Hit time: Must match cache/DRAM performance
- Miss latency: Very high. ~20ms
- Tag storage overhead: Low, relative to block size
Locating an object in a “Cache”

- **SRAM Cache**
  - Tag stored with cache line
  - Maps from cache block to memory blocks
    - From cached to uncached form
    - Save a few bits by only storing tag
  - No tag for block not in cache
  - Hardware retrieves information
    - Can quickly match against multiple tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>D</td>
</tr>
<tr>
<td>1:</td>
<td>X</td>
</tr>
<tr>
<td>N-1:</td>
<td>J</td>
</tr>
</tbody>
</table>

Object Name

\[ X \]
Locating an object in “Cache” (cont.)

- **DRAM Cache**
  - Each allocated page of virtual memory has entry in *page table*
  - Mapping from virtual pages to physical pages
    - From uncached form to cached form
  - Page table entry even if page not in memory
    - Specifies disk address
    - Only way to indicate where to find page
  - OS retrieves information

![Diagram of Page Table and "Cache"]

- **Location**
  - D: 0
  - J: On Disk
  - X: 1

- **Data**
  - N-1: 105
  - 1: 17
  - 0: 243

- **Object Name**
  - X
Page faults (like “cache misses”)

- What if an object is on disk rather than in memory?
  - Page table entry indicates virtual address not in memory
  - OS exception handler invoked to move data from disk into memory
    - current process suspends, others can resume
    - OS has full control over placement, etc.
Servicing a page fault

- Processor signals controller
  - Read block of length P starting at disk address X and store starting at memory address Y

- Read occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller

- I/O controller signals completion
  - Interrupt processor
  - OS resumes suspended process
Motivation #2: Memory management

- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
  - what if two processes access something at the same address?

Linux/x86 process
Memory image
Solution: Separate virtual addr. spaces

- Virtual and physical address spaces divided into equal-sized blocks
  - blocks are called “pages” (both virtual and physical)
- Each process has its own virtual address space
  - operating system controls how virtual pages are assigned to physical memory
Motivation #3: Protection

- Page table entry contains access rights information
  - hardware enforces this protection (trap into OS if violation occurs)

<table>
<thead>
<tr>
<th></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>

Process i:

<table>
<thead>
<tr>
<th></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>

Physical memory:

- PP 0
- PP 2
- PP 4
- PP 6
- PP 9
- PP 11
VM address translation

- Virtual Address Space
  - \( V = \{0, 1, \ldots, N-1\} \)

- Physical Address Space
  - \( P = \{0, 1, \ldots, M-1\} \)
  - \( M < N \)

- 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
Page tables

Virtual Page Number

Memory resident page table
(physical page or disk address)

Valid

Physical Memory

Disk Storage
(swap file or regular file system file)
VM address translation: Miss

Processor → Hardware Addr Trans Mechanism

Virtual address: a

Part of the on-chip memory mgmt unit (MMU)

Main Memory

Physical address: a'

Physical address

Virtual address
VM address translation: Miss

Processor

Hardware Addr Trans Mechanism

fault handler

Main Memory

Secondary memory

virtual address

part of the on-chip memory mgmt unit (MMU)

page fault

physical address

OS performs this transfer (only if miss)
VM address translation

- **Parameters**
  - $P = 2^p =$ page size (bytes).
  - $N = 2^n =$ Virtual address limit
  - $M = 2^m =$ Physical address limit

Page offset bits don’t change as a result of translation
Address translation via page table

- Virtual page number (VPN) acts as table index.
- Valid access: if valid=0, then page not in memory.
- Physical page number (PPN) is determined by the page table base register and the VPN.
Page table operation

- **Translation**
  - Separate (set of) page table(s) per process
  - VPN forms index into page table (points to a page table entry)
Page table operation

- Computing physical address
  - Page Table Entry (PTE) provides info about page
    - if (valid bit = 1) then the page is in memory.
      - Use physical page number (PPN) to construct address
    - if (valid bit = 0) then the page is on disk - page fault
Page table operation

- Checking protection
  - Access rights field indicate allowable access
    - e.g., read-only, read-write, execute-only
    - typically support multiple protection modes
  - Protection violation fault if user doesn’t have necessary permission

virtual physical page number (VPN)

page table base register

valid access physical page number (PPN)

virtual page number (VPN)

page offset
Integrating VM and cache

- Most caches “Physically Addressed”
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at a time
  - Allows multiple processes to share pages
  - Cache doesn’t need to be concerned with protection issues
    - Access rights checked as part of address translation

- Perform address translation before cache lookup
  - But this could involve a memory access itself (of the PTE)
  - Of course, page table entries can also become cached
Speeding up translation with a TLB

• “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
Address translation with a TLB

The diagram illustrates the process of address translation with a translation lookaside buffer (TLB). The virtual address consists of a virtual page number and a page offset. The virtual page number is used to look up the corresponding physical page number in the TLB. If a match is found (TLB hit), the physical address is determined. Otherwise, a cache lookup is performed. If the data is found in the cache (cache hit), the requested data is returned. If not, the data is not returned. The diagram shows the flow of data and the components involved in this process.
Multi-level page tables

Given:
- 4KB ($2^{12}$) page size
- 32-bit address space
- 4-byte PTE

Problem:
- Would need a 4 MB page table!
  - $2^{20} \times 4$ bytes

Common solution
- Paged the page table – multi-level page tables
- e.g., 2-level table (P6)
  - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
  - Level 2 table: 1024 entries, each of which points to a page
Multi-level page table with $k$ levels

- Virtual address split into $k$ VPNs and a VPO, each VPN $i$ is an index into a page table at level $i$
- Of course, now you really need a TLB!
Taking stock – main themes

- **Programmer’s view**
  - Large “flat” address space
    - Can allocate large blocks of contiguous addresses
  - Processor “owns” machine
    - Has private address space
    - Unaffected by behavior of other processes

- **System view**
  - Virtual address space created by mapping to set of pages
    - Need not be contiguous
    - Allocated dynamically
    - Enforce protection during address translation
  - OS manages many processes simultaneously
    - Continually switching among processes
    - Especially when one must wait for resource
      - E.g., disk I/O to handle page fault
Simple memory system

- Memory is byte addressable
- Access are to 1-byte words
- 14-bit virtual addresses, 12-bit physical address
- Page size = 64 bytes \((2^6)\)

![Diagram of virtual and physical addresses]

- **Virtual address**
  - 14-bit virtual address: \(VPO\)
  - 12-bit physical address: \(PPN\)
  - **VPN**: (Virtual Page Number)
  - **VPO**: (Virtual Page Offset)

- **Physical address**
  - Physical Page Number: \(PPN\)
  - Physical Page Offset: \(PPO\)
Simple memory system page table

- Only show first 16 entries

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple memory system TLB

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

- **Cache**
  - 16 lines
  - 4-byte line size
  - Direct mapped

![Cache Diagram]

<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>
Address translation problem 9.12

- **Virtual address** 0x03a9

- **Physical address**

```plaintext
Address translation problem 9.12

- Virtual address 0x03a9

- Physical address
```
Address translation problem 9.13

- **Virtual address** \(0x0040\)

  - **VPN** — **VPO**

- **Physical address**

  - **PPN** — **PPO**

VPN ___ TLBI ___ TLBTag ___ TLB Hit? ___ Page Fault? ___ PPN: ___

Offset ___ CI___ CT _____ Hit? ___ Byte returned: ____
Summary

- **Today**
  - VM motivation
  - VM mechanisms
  - Optimizing VM performance

- **Next time:** Dynamic memory allocation