Virtual Memory

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

Next time
- Linking
Main memory - An array of M contiguous byte-sized cells, each with a unique physical address

Physical addressing
- Most natural way to access it – Addresses generated by the CPU correspond to bytes in it
- Used in simple systems like early PCs and embedded microcontrollers (e.g. cars and elevators)
A system with virtual addressing

- Modern processors use virtual addresses
- CPU generates virtual address and address translation is done by dedicated hardware (*memory management unit*) via OS-managed lookup table
Motivations for virtual memory

- Use physical DRAM as a cache for the disk
  - Address space of a process can exceed physical memory
  - 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

**Register**
- size: 32 B
- speed: 1 ns
- $/Mbyte: $125/MB
- line size: 8 B

**Cache**
- 32KB-4MB
- 2 ns
- $125/MB
- 32 B

**Memory**
- 1024 MB
- 30 ns
- $0.20/MB
- 4 KB

**Disk Memory**
- 100 GB
- 8 ms
- $0.001/MB

- 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 ~100X cheaper than DRAM storage
  - 2TB of DRAM: ~ $10,000
  - 2TB GB of disk: ~$100

- 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
Virtual and physical pages

- Data on disk is partitioned into fixed sized blocks – virtual pages
- Physical memory into blocks of equal size – physical pages or page frames
- At any point, VM pages are partitioned into three sets
  - Unallocated
  - Allocated
  - Cached
If DRAM were to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size? Large, since disks are better at xferring large blocks.
- Associativity? High, to minimize miss rate.
- Write through/write back? Write back, 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

```
Object Name

X

= X?
```

```

<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></td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td>J</td>
</tr>
</tbody>
</table>
```

“Cache”
Locating an object in “Cache”

- DRAM Cache
  - Each allocated page of virtual memory has entry in *page table*
  - Mapping from virtual pages to physical pages

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 0</td>
<td>243</td>
</tr>
<tr>
<td>1: On Disk</td>
<td>17</td>
</tr>
<tr>
<td>7: 1</td>
<td>105</td>
</tr>
</tbody>
</table>

Virtual memory (disk)

Physical memory (DRAM “Cache”)

Page Table
Locating an object in "Cache"

- Mapping from virtual pages to physical pages
- Page table entry even if page not in memory
  - Specifies disk address
  - Only way to indicate where to find page
  - OS retrieves information

```
Virtual address
Object Name
X

Page Table
Location
0: 0
1: On Disk
7: 1

“Cache”
Data
0: 243
1: 17
3: 105
```
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.

Before fault

After fault
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

How does anything get done?! Locality to the rescue!
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.
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

<table>
<thead>
<tr>
<th>Memory image</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="diagram.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Kernel virtual memory</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Memory mapped region for shared libraries</td>
</tr>
<tr>
<td>Runtime heap (via malloc)</td>
</tr>
<tr>
<td>Uninitialized data (.bss)</td>
</tr>
<tr>
<td>Initialized data (.data)</td>
</tr>
<tr>
<td>Program text (.text)</td>
</tr>
<tr>
<td>Forbidden</td>
</tr>
</tbody>
</table>

Memory invisible to user code
Solution: Separate virtual addr. spaces

- Each process has its own virtual address space
  - OS controls how virtual pages are assigned to physical mem.

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address Translation

Physical Address Space (DRAM)

(e.g., read/only library code)
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
Motivation #3: Protection

- Page table entry contains access rights information
  - HW enforces this protection (trap into OS if violation occurs)
VM address translation

- First some terminology ...

- **Virtual Address Space**
  - \( V = \{0, 1, \ldots, N-1\} \ (N = 2^n) \)

- **Physical Address Space**
  - \( P = \{0, 1, \ldots, M-1\} \ (M = 2^m) \)
  - \( M < N \)

- **Address Translation**
  - \( \text{MAP}: \ V \rightarrow \ P \cup \{\emptyset\} \)
  - For virtual address \( a \):
    - \( \text{MAP}(a) = a' \) if data at virtual address \( a \) is at physical address \( a' \) in \( P \)
    - \( \text{MAP}(a) = \emptyset \) 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 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

| Physical Memory |

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

- Processor
  - virtual address
- Hardware Addr Trans Mechanism
  - part of the on-chip memory mgmt unit (MMU)
- Main Memory
  - physical address
VM address translation: Miss

Processor

Hardware Addr Trans Mechanism

Main Memory

Secondary memory

page fault

fault handler

virtual address

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

physical address

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

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

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

- **Virtual page number (VPN)** acts as a table index.
- If valid=0, then page not in memory.
- Page table base register.
- Physical page number (PPN) and page offset.
Page table operation

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

![Diagram of page table operation]

- Physical page number (PPN) to physical address
- Virtual page number (VPN) to virtual address

**Page table address for process**

- Virtual address
  - Virtual page number (VPN)
  - Page offset

- Physical address
  - Physical page number (PPN)
  - Page offset

- Page table base register

- VPN acts as table index

- If valid=0, then page not in memory
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

![Diagram of page table operation]

- page table base register
- VPN acts as table index
- if valid=0 then page not in memory
- virtual address
- virtual page number (VPN)
  - p–1
  - p
  - m–1
  - n–1
- physical page number (PPN)
  - p–1
  - p
  - m–1
  - n–1
- valid
- access
- physical address
- Page table
- physical page number (PPN)
  - p–1
  - p
  - m–1
  - n–1
- 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

![Diagram of TLB and memory system](image)
Address translation with a TLB

- Virtual address = virtual page number + page offset
- Physical address = physical page number + byte offset

TLB hit

Cache hit
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 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 behaviour 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 a 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 ($2^6$)

Virtual address

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

VPN (Virtual Page Number)

VPO (Virtual Page Offset)

Physical address

<table>
<thead>
<tr>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

PPN (Physical Page Number)

PPO (Physical Page Offset)
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>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Set</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>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>2</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>2</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>2</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>3</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>3</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>3</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

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| CT | Cl | CO |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

PPN PPO

Idx | Tag | Valid | B0 | B1 | B2 | B3 | Idx | Tag | Valid | B0 | B1 | B2 | B3
---+-----+-------+----+----+----+----+---+-----+-------+----+----+----+----
 0  |  19  |   1   |  99|  11|  23|  11|  8  |  24 |   1   |  3A|  00|  51|  89
 1  |  15  |   0   |   |   |   |   |  9  |  2D |   0   |   |   |   |   
 2  | 1B   |   1   |  00|  02|  04|  08| A   | 2D  |   1   |  93|  15| DA |  3B
 3  | 36   |   0   |   |   |   |   | B   | 0B  |   0   |   |   |   |   
 4  | 32   |   1   |  43|  6D|  8F|  09| C   | 12  |   0   |   |   |   |   
 5  | 0D   |   1   |  36|  72| F0 | 1D| D   | 16  |   1   |  04|  96|  34|  15
 6  | 31   |   0   |   |   |   |   | E   | 13  |   1   |  83|  77|  1B| D3
 7  | 16   |   1   |  11| C2 | DF | 03| F   | 14  |   0   |   |   |   |   
```
Address translation problem 9.4

- **Virtual address** 0x03d7

![Diagram of TLB and VPN with virtual address 0x03d7]

<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>
Address translation problem 9.4

- Virtual address 0x03d7

```
13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 1 1 0 1 0 1 1 1
```

- Physical address

```
11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 0 1 0 1 0 1 1 1
```
Address translation problem 9.4

- **Virtual address** \(0x03d7\)
- **Physical address**

![Diagram of address translation process]

- Offset: \(0x3\)
- CI: \(0x5\)
- CT: \(0xd\)
- Hit?: Y
- Byte returned: \(0x1d\)

<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>

<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>
Summary

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

- Next time - Linking