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 #1: DRAM a “Cache” for Disk

Full address space is quite large:
- 32-bit addresses: \(~4,000,000,000\) (4 billion) bytes
- 64-bit addresses: \(~16,000,000,000,000,000\) \(\times 16\) quintillion) bytes

Disk storage is \(~156X\) cheaper than DRAM storage
- 8 GB of DRAM: \(~$10,000\)
- 8 GB of disk: \(~$64\)

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk

Levels in Memory Hierarchy

<table>
<thead>
<tr>
<th>Register Cache Memory Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>size: 32 B 32 KB-4MB 128 MB 30 GB</td>
</tr>
<tr>
<td>speed: 3 ns 6 ns 60 ns 8 ms</td>
</tr>
<tr>
<td>$/Mbyte: $100/MB $1.25/MB $0.008/MB</td>
</tr>
<tr>
<td>line size: 8 B 32 B 4 KB</td>
</tr>
</tbody>
</table>

larger, slower, cheaper
**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 These 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
- No tag for block not in cache
- Hardware retrieves information
  - can quickly match against multiple tags

**DRAM Cache**

- Each allocate 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
- OS retrieves information

---

**Locating an Object in a “Cache” (cont.)**
### Page Faults (Similar to “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?

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 as assigned to physical memory

Contrast: Macintosh Memory Model

MAC OS 1–9
- Does not use traditional virtual memory

All program objects accessed through “handles”
- Indirect reference through pointer table
- Objects stored in shared global address space

Macintosh Memory Management

Allocation / Deallocation
- Similar to free-list management of malloc/free

Compaction
- Can move any object and just update the (unique) pointer in pointer table
Mac vs. VM-Based Memory Mgmt

Allocating, deallocating, and moving memory:
- can be accomplished by both techniques

Block sizes:
- Mac: variable-sized
  - may be very small or very large
- VM: fixed-size
  - size is equal to one page (4KB on x86 Linux systems)

Allocating contiguous chunks of memory:
- Mac: contiguous allocation is required
- VM: can map contiguous range of virtual addresses to disjoint ranges of physical addresses

Protection
- Mac: “wild write” by one process can corrupt another’s data

MAC OS X

“Modern” Operating System
- Virtual memory with protection
- Preemptive multitasking
  - Other versions of MAC OS require processes to voluntarily relinquish control

Based on MACH OS
- Developed at CMU in late 1980’s

Motivation #3: Protection

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

<table>
<thead>
<tr>
<th>Process:</th>
<th>VP 0:</th>
<th>VP 1:</th>
<th>VP 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Read?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Write?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process:</th>
<th>VP 0:</th>
<th>VP 1:</th>
<th>VP 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>Read?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Write?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VM Address Translation

V = {0, 1, . . . , N–1} virtual address space
P = {0, 1, . . . , M–1} physical address space
N > M

MAP: V → P ∪ {∅} address mapping function

MAP(a) = a’ if data at virtual address a is present at physical address a’ in P
= ∅ if data at virtual address a is not present in P

Processor

Hardware Addr Trans Mechanism

Main Memory

Secondary memory

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

Notice that the page offset bits don't change as a result of translation.

Address Translation via Page Table

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

Computing Physical Address
- Page Table Entry (PTE) provides information 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
    » Must load page from disk into main memory before continuing

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

Most Caches “Physically Addressed”
- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at the same 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

Simple Memory System Example

Addressing
- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bits
Simple Memory System Page Table

- Only show first 16 entries

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>2B</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>09</td>
<td>17</td>
</tr>
<tr>
<td>02</td>
<td>0A</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>0B</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>0C</td>
<td>1</td>
</tr>
<tr>
<td>05</td>
<td>0D</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>0E</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>0F</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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>00</td>
<td>09</td>
</tr>
<tr>
<td>1</td>
<td>02</td>
<td>04</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>03</td>
<td>00</td>
<td>1A</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>03</td>
<td>00</td>
</tr>
</tbody>
</table>

Simple Memory System Cache

- 16 lines
- 4-byte line size
- 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>11</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>0B</td>
<td>02</td>
<td>04</td>
<td>08</td>
<td>9</td>
<td>2D</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>0</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>2C</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

Address Translation Example #1

Virtual Address 0x03D4

<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>11</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>0B</td>
<td>02</td>
<td>04</td>
<td>08</td>
<td>9</td>
<td>2D</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>0</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>2C</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

Physical Address
Address Translation Example #2

Virtual Address 0x027C

Physical Address

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
- 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

Address Translation Example #3

Virtual Address 0x0040

Physical Address

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
- User 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