Virtual Memory
October 26, 2000

Topics

• Motivations for VM
• Address translation
• Accelerating translation with TLBs
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: ~16,000,000,000,000,000,000 (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

- **CPU (regs)**: 32 B, 3 ns, $100/MB, 8 B
- **Cache**: 32 KB-4MB, 6 ns, $1.25/MB, 32 B
- **Memory**: 128 MB, 60 ns, 4 KB
- **Disk Memory**: 30 GB, 8 ms, $0.008/MB

Comparisons:
- **Register**
  - Size: 32 B
  - Speed: 3 ns
  - Cost: $100/MB
  - Line size: 8 B
- **Cache**
  - Size: 32 KB-4MB
  - Speed: 6 ns
  - Cost: $1.25/MB
  - Line size: 32 B
- **Memory**
  - Size: 128 MB
  - Speed: 60 ns
  - Cost: $1.25/MB
  - Line size: 4 KB
- **Disk Memory**
  - Size: 30 GB
  - Speed: 8 ms
  - Cost: $0.008/MB

Comparison States:
- Larger, Slower, Cheaper

Symbols:
- `-- cache` (horizontal double arrow)
- `-- virtual memory` (horizontal double arrow)
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

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>243</td>
</tr>
<tr>
<td>X</td>
<td>17</td>
</tr>
<tr>
<td>J</td>
<td>105</td>
</tr>
</tbody>
</table>

Object Name

= X?
Locating an Object in a “Cache” (cont.)

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

![Diagram of DRAM Cache and Page Table]

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>D: 0</td>
<td>243</td>
</tr>
<tr>
<td>J: On Disk</td>
<td>17</td>
</tr>
<tr>
<td>X: 1</td>
<td></td>
</tr>
<tr>
<td>N-1:</td>
<td>105</td>
</tr>
</tbody>
</table>
A System with Physical Memory Only

Examples:
• most Cray machines, early PCs, nearly all embedded systems, etc.

Addresses generated by the CPU point directly to bytes in physical memory.
A System with Virtual Memory

Examples:
- workstations, servers, modern PCs, etc.

Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)
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?

Linux/x86 process memory image

- kernel virtual memory
- stack
- Memory mapped region for shared libraries
- runtime heap (via malloc)
- uninitialized data (.bss)
- initialized data (.data)
- program text (.text)
- forbidden

memory invisible to user code

%esp

the “brk” ptr
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

Virtual Address Space for Process 1:
- 0
- VP 1
- VP 2
- ...
- N-1

Virtual Address Space for Process 2:
- 0
- VP 1
- VP 2
- ...
- N-1

Address Translation:
- 0
- PP 2
- PP 7
- PP 10
- M-1

Physical Address Space (DRAM):
- (e.g., read/only library code)
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

```
<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>
```

“Handles”
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)

Page Tables

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>Read?</th>
<th>Write?</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
<td></td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>No</td>
<td>XXXXXXX</td>
</tr>
</tbody>
</table>

Process i:

Process j:

Memory
VM Address Translation

\[ V = \{0, 1, \ldots, N-1\} \text{ virtual address space} \]
\[ P = \{0, 1, \ldots, M-1\} \text{ physical address space} \]

\[ N > M \]

MAP: \( V \rightarrow P \cup \{\emptyset\} \) address mapping function

\[ \text{MAP}(a) = a' \text{ if data at virtual address } a \text{ is present at physical address } a' \text{ in } P \]
\[ = \emptyset \text{ if data at virtual address } a \text{ is not present in } P \]

Diagram:
- Processor
- Hardware Addr Trans Mechanism
  - virtual address \( a \)
  - part of the on-chip memory mgmt unit (MMU)
- Main Memory
  - physical address \( a' \)
  - \( \emptyset \) for page fault
- Secondary memory
  - OS performs this transfer (only if miss)
  - fault handler

"class18.ppt"

CS 213 F'00
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.
Page Tables

Memory resident page table (physical page or disk address)

Virtual Page Number

Valid

0

1

1

1

1

1

1

0

0

1

1

1

0

1

1

1

Physical Memory

Disk Storage (swap file or regular file system file)
Address Translation via Page Table

- Virtual page number (VPN)
- Page offset
- Physical page number (PPN)
- Page offset

If valid = 0, then page not in memory.
Page Table Operation

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 Look-up
- 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

The diagram illustrates the process of translation with a TLB:

1. CPU requests a virtual address (VA) translation.
2. The VA is sent to the TLB lookup module.
3. If a hit is found in the TLB, the physical address (PA) is returned directly to the CPU.
4. If a miss occurs in the TLB, the CPU requests the VA translation again, sending it to the main memory.
5. The main memory sends the physical address and data back to the CPU.
6. If a hit occurs in the main memory, the data is returned to the CPU.
7. If a miss occurs in the main memory, the CPU requests the VA translation again, sending it to the TLB.

This process speeds up translation and reduces latency in memory access.
Address Translation with a TLB

Virtual address = virtual page number p page offset

Physical address = index tag

Cache hit

TLB hit

valid tag physical page number

Cache

valid tag data

TLB

TLB hit

n-1 valid page offset 0

virtual address

cache hit

data
Simple Memory System Example

Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bits

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

VPN  VPO
(Virtual Page Number)  (Virtual Page Offset)

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

PPN  PPO
(Physical Page Number)  (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>0</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>10</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
<td>0B</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>0</td>
<td>0</td>
<td>0C</td>
<td>0</td>
<td></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>0</td>
<td>0</td>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>0</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

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  PPN    CO     CI     CT
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

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

Virtual Address 0x03D4

Virtual Page Number (VPN) ___
Translation Lookaside Buffer (TLBI) ___
Translation Lookaside Buffer (TLBT) ___
TLB Hit? ___
Page Fault? ___
Page Frame Number (PPN): ___

Physical Address

Offset ___
Page Offset (PO) ___
Translation Offset (TO) ___
Translation Hit? ___
Byte: ___
Address Translation Example #2

Virtual Address 0x027C

Physical Address

Offset ___  CI___  CT ____  Hit? ___  Byte: ____
Address Translation Example #3

Virtual Address 0x0040

<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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CT ___ CI ___ CO ___

Offset ___ CI ___ CT ____ Hit? __ Byte: ____

class18.ppt
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!
  \(- \quad 2^{20} \times 4\) bytes

Common solution
- multi-level page tables
- e.g., 2-level table (P6)
  \(- \quad \text{Level 1 table: 1024 entries, each of which points to a Level 2 page table.}
  \- \quad \text{Level 2 table: 1024 entries, each of which points to a page}
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