15-213
“The course that gives CMU its Zip!”

Virtual Memory
October 26, 2004

Topics
■ Motivations for VM
■ Address translation
■ Accelerating translation with TLBs
Classic 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 has 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.
Modern Motivations for VM

- Memory sharing and control
  - Copy on write: share physical memory among multiple processes until a process tries to write to it. At that point make a copy. For example, this eliminates the need for `vfork()`
  - Shared libraries
  - Protection (debugging) via Segment-Drivers (Solaris)

- Sparse address space support (64bit systems)

- Memory as a fast communication device
  - Part of memory is shared by multiple processes

- Multiprocessing (beyond the scope of 15-213)
Why does VM Work?

It is not used!
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: \(~16,000,000,000,000,000\) (16 quintillion) bytes

Disk storage is \(~500\times\) cheaper than DRAM storage
- 80 GB of DRAM: \(~$25,000\)
- 80 GB of disk: \(~$50\)

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

Register | Cache | Memory | Disk Memory
---|---|---|---
Size: 32 B | 32 KB-4MB | 1024 MB | 100 GB
Latency: < 1 ns | ~2 ns | > 50 ns | > 8 ms
$/Mbyte: $125/MB | $0.20/MB | $0.001/MB
Line size: 8(16) B | 32(64) B | 4(64+) KB | 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 ~160,000X slower than DRAM!

- Importance of exploiting spatial locality:
  - First byte is ~160,000X 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. ~10ms
- 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
```

```
<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” (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
A System with Physical Memory Only

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

- Addresses generated by the CPU correspond 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 OS-managed lookup table (page table)
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
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 mapped region for shared libraries</th>
</tr>
</thead>
<tbody>
<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>

Kernel virtual memory

%esp

memory invisible to user code

the “brk” ptr
**Solution: Separate Virt. 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

```
P1 Pointer Table
```

```
Shared Address Space
```

```
A
```
```
B
```
```
C
```
```
D
```
```
E
```

“Handles”

```
P2 Pointer Table
```

```
Process P1
```
```
Process P2
```

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

Diagram:
- P1 Pointer Table
- Process P1
- Shared Address Space
- Process P2
- P2 Pointer 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>Process i:</th>
<th>Read?</th>
<th>Write?</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</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>XXXXXXXX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>Read?</th>
<th>Write?</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>No</td>
<td>XXXXXXXX</td>
</tr>
</tbody>
</table>

Memory

0:

1:

N-1:
VM Address Translation

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

Physical Address Space
- $P = \{0, 1, \ldots, M-1\}$
- $M < N$ (usually, but $\geq 4$ Gbyte on an IA32 possible)

Address Translation
- $\text{MAP}: V \rightarrow P \cup \{\emptyset\}$
- For virtual address $a$:
  - $\text{MAP}(a) = a'$ if data at virtual address $a$ at physical address $a'$ in $P$
  - $\text{MAP}(a) = \emptyset$ if data at virtual address $a$ not in physical memory
    » Either invalid or stored on disk
VM Address Translation: Hit

- Processor
- Virtual address
- Hardware Addr Trans Mechanism
- Part of the on-chip Memory Management Unit (MMU)
- Physical address
- Main Memory
- a
- a'

Diagram showing the flow from processor to main memory through the hardware address translation mechanism, with virtual and physical addresses indicated.
VM Address Translation: Miss

Processor → Hardware Addr Trans Mechanism → Main Memory → Secondary memory

virtual address → a → part of the on-chip Memory Management Unit (MMU) → physical address → a' → fault handler

page fault

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
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)
Address Translation via Page Table

page table base register

VPN acts as table index

virtual address

valid access physical page number (PPN)

if valid=0 then page not in memory

physical page number (PPN)
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 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
Page Table Operation

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
Most Caches were “Physically Addressed”

- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at 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

![Diagram showing the process of translation using a TLB]

- CPU
- TLB Lookup
- Cache
- Main Memory

VA → TLB Lookup

hit PA → Cache

miss → Translation

miss → Main Memory

hit → data

miss
Address Translation with a TLB

- TLB hit
- Cache hit

Virtual page number

Page offset

Virtual address

Physical page number

Physical address

Tag

Index

Byte offset

Valid

Data
Simple Memory System Example

Addressing

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

![Diagram of memory addressing]

- VPN (Virtual Page Number) vs VPO (Virtual Page Offset)
- PPN (Physical Page Number) vs PPO (Physical Page Offset)
## 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>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</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>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</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>
</tr>
</tbody>
</table>

Tag PPN Valid
00 – 0
04 – 0
06 – 0
0A 34 1
02 – 0
**Simple Memory System Cache**

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

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

<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>

(PPO and PPN)
Address Translation Example #1

Virtual Address 0x03D4

```
Virtual Address: 0x03D4
TLBT  TLBI
000111101010100
```

Physical Address

```
Physical Address: 0x03D4
CT  CI  CO
00110101010100
```

Offset 0  CI 0x5  CT 0x0D  Hit? Y  Byte: 0x36
Address Translation Example #2

Virtual Address 0x0B8F

\[
\begin{array}{cccccccccccc}
\text{TLBT} & & & & & & & & & & \text{TLBI} \\
13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

VPN 0x2E TLBI 2 TLBT 0xB TLB Hit? NO Page Fault? YES PPN: TBD

Physical Address

\[
\begin{array}{cccccccccccc}
\text{CT} & & & & & & & & & & \text{Cl} & & & & \text{Co} \\
11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\end{array}
\]

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

Virtual Address 0x0020

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

VPN 0x00  TLBI 0  TLBT 0x00  TLB Hit? NO  Page Fault? NO  PPN: 0x28

Physical Address

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

Offset 0  CI 0x8  CT 0x28  Hit? NO  Byte: MEM
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
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