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 ~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

Levels in Memory Hierarchy

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

Object Name

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>D</td>
<td>243</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>J</td>
<td>155</td>
</tr>
</tbody>
</table>
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
- Specify disk address
- Only way to indicate where to find page
- OS retrieves information

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>D: 0</td>
<td>0: 243</td>
</tr>
<tr>
<td></td>
<td>J: On Disk</td>
<td>1: 17</td>
</tr>
<tr>
<td></td>
<td>X: 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N-1: 105</td>
<td></td>
</tr>
</tbody>
</table>

A System with Physical Memory Only

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

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.

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

- Stack
- Memory mapped region (shared libraries)
- Runtime heap (via malloc)
- Program text (text)
- kernel virtual memory
- The "fork" ptr
- Memory invisible to user code
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 are assigned to physical memory.

<table>
<thead>
<tr>
<th>Process 1: VP 1</th>
<th>VP 2</th>
<th>PP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td>VP 1</td>
<td>VP 2</td>
</tr>
<tr>
<td>N-1</td>
<td>VP 2</td>
<td>PP 2</td>
</tr>
<tr>
<td>N-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Virtual Address Space for Process 1: VP 1

Physical Address Space (DRAM)

Virtual Address Space for Process 2: VP 1

Physical Address Space (DRAM)

Contrast: Macintosh Memory Model

MAC OS 1–9

- Does not use traditional virtual memory.
- Process P1
  - Pointer Table P1
  - Shared Address Space
  - "Handles" P2
  - Pointer Table 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.

Shared Address Space

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

Memory

Process i:

<table>
<thead>
<tr>
<th>Physical Addr</th>
<th>Yes</th>
<th>No</th>
<th>PP 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read? Write?</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>Read? Write?</td>
<td>Yes</td>
<td>No</td>
<td>XXXXX</td>
</tr>
<tr>
<td>Read? Write?</td>
<td>No</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>Read? Write?</td>
<td>No</td>
<td>No</td>
<td>XXXXX</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process j:

<table>
<thead>
<tr>
<th>Physical Addr</th>
<th>Yes</th>
<th>No</th>
<th>PP 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read? Write?</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>Read? Write?</td>
<td>Yes</td>
<td>No</td>
<td>XXXXX</td>
</tr>
<tr>
<td>Read? Write?</td>
<td>No</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>Read? Write?</td>
<td>No</td>
<td>No</td>
<td>XXXXX</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $a$:
  - $\text{MAP}(a) = a'$ 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
\[ a \]
Hardware
Addr Trans
Mechanism
\[ a' \]
Main
Memory
physical address
part of the
on-chip
memory mgmt unit (MMU)

VM Address Translation: Miss

Processor
\[ a \]
Hardware
Addr Trans
Mechanism
\[ \emptyset \]
Main
Memory
\[ a' \]
Secondary
Memory
physical address
part of the
on-chip
memory mgmt unit (MMU)

OS performs this transfer (only if miss)

Page offset bits don't change as a result of translation

Address Translation via Page Table

Virtual Memory
\[ m-1 \]
page table base register
\[ p \]
physical address
\[ \text{VPN acts as table index} \]
valid access
physical page number (PPN)

If valid=0 then page not in memory

physical page number (PPN)
page offset
physical address

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

Page Tables

Virtual Page Number
Memory resident
page table
(physical page in disk address)

Physical Memory

Disk Storage
(swap file or regular file system file)
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.
  - If (valid bit = 0) then the page is on disk
- Page fault

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 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
- Virtual page number (VPN) acts as table index
- Valid physical page number (PPN) access
- VPN acts as index
- Valid physical page number (PPN) access
- Valid tag physical page number index
- Tag physical page number data
- Physical page number (PPN) access
- Page not in memory
- Page in memory
Simple Memory System Example

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

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

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>01</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>0A</td>
<td>03</td>
<td>02</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>0</td>
<td>0C</td>
<td>05</td>
<td>18</td>
<td>0D</td>
</tr>
<tr>
<td>06</td>
<td>0</td>
<td>0E</td>
<td>07</td>
<td>0</td>
<td>0F</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>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>0</td>
<td>0</td>
<td>02</td>
<td>0</td>
<td>0</td>
<td>02</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>0</td>
<td>0</td>
<td>03</td>
<td>0</td>
<td>0</td>
<td>03</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Simple Memory System Cache
- 16 lines
- 4-byte line size
- Direct mapped

<table>
<thead>
<tr>
<th>Line</th>
<th>Tag</th>
<th>Valid</th>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>0</td>
<td>0A</td>
<td>0C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3A</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>0</td>
<td>0E</td>
<td>0F</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>0</td>
<td>04</td>
<td>05</td>
</tr>
</tbody>
</table>

Address Translation Example #1

Virtual Address 0x3D4

Address Translation Example #2

Virtual Address 0xB8F
Address Translation Example #3

Virtual Address: 0x0040

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} * 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