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

<table>
<thead>
<tr>
<th>Register</th>
<th>Cache</th>
<th>Memory</th>
<th>Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>32 B</td>
<td>32 KB-4 MB</td>
<td>1024 MB</td>
</tr>
<tr>
<td>speed</td>
<td>1 ns</td>
<td>2 ns</td>
<td>30 ns</td>
</tr>
<tr>
<td>$/Mbyte</td>
<td>$125/MB</td>
<td>$0.20/MB</td>
<td>$0.001/MB</td>
</tr>
<tr>
<td>line size</td>
<td>8 B</td>
<td>32 B</td>
<td>4 KB</td>
</tr>
</tbody>
</table>


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

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
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

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Location</th>
<th>“Cache”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>243</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>105</td>
</tr>
</tbody>
</table>

A System with Physical Memory Only

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

![Diagram showing CPU, Physical Addresses, Memory, Page Table, and a list of physical addresses](image)

- Addresses generated by the CPU correspond directly to bytes in physical memory

A System with Virtual Memory

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

![Diagram showing CPU, Virtual Addresses, Page Table, Physical Addresses, Memory, and a list of virtual and physical addresses](image)

- 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.

![Diagram showing before and after a page fault](image)

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

![Diagram showingservicing a page fault](image)

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
- kernel virtual memory
- memory invisible to user code
- memory mapped region
- forked shared libraries
- initialized data
- stack initialized data
- text (text)
- data
- program text (data)
- forked

- the “brk” ptr

Page 2
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

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Motivation #3: Protection

Page table entry contains access rights information

- hardware enforces this protection (trap into OS if violation occurs)

VM Address Translation

Virtual Address Space
- V = {0, 1, ..., N-1}

Physical Address Space
- P = {0, 1, ..., M-1}
- M < N

Address Translation
- MAP: V → P U {∅}
- For virtual address a:
  - MAP(a) = a' if data at virtual address a at physical address a' in P
  - MAP(a) = ∅ if data at virtual address a not in physical memory
    - Either invalid or stored on disk

VM Address Translation: Hit

Processor

Main Memory

Hardware Addr Trans Mechanism

fault handler

Virtual Address Translation: Miss

Processor

Main Memory

Hardware Addr Trans Mechanism

Page offset bits don’t change as a result of translation
Page Tables

Address Translation via Page Table

Page Table Operation

Integrating VM and Cache

Most Caches “Physically Addressed”

Perform Address Translation Before Cache Lookup
**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 a small number of pages

**Address 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 a small number of pages

---

**Simple Memory System Example**

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

**Simple Memory System Page Table**
- Only show first 16 entries

**Simple Memory System TLB**
- 16 entries
- 4-way associative

**Simple Memory System Cache**
- 16 lines
- 4-byte line size
- Direct mapped
**Address Translation Example #1**

Virtual Address 0x03D4

```
|   |   |   |   |   |   |   |   | 1 | 0 |
```

VPN: 
TLBI: 
Page Fault?: 
PPN:

Physical Address

```
|   |   |   |   |   |   |   |   | 1 | 0 |
```

Offset: 
CI: 
CT: 
Hit?: 
Byte:

---

**Address Translation Example #2**

Virtual Address 0x0B8F

```
|   |   |   |   |   |   |   |   | 1 | 0 |
```

VPN: 
TLBI: 
Page Fault?: 
PPN:

Physical Address

```
|   |   |   |   |   |   |   |   | 1 | 0 |
```

Offset: 
CI: 
CT: 
Hit?: 
Byte:

---

**Address Translation Example #3**

Virtual Address 0x0040

```
|   |   |   |   |   |   |   |   | 1 | 0 |
```

VPN: 
TLBI: 
Page Fault?: 
PPN:

Physical Address

```
|   |   |   |   |   |   |   |   | 1 | 0 |
```

Offset: 
CI: 
CT: 
Hit?: 
Byte:

---

**Multi-Level Page Tables**

**Given:**
- 4KB (212) page size
- 32-bit address space
- 4-byte PTE

**Problem:**
Would need a 4 MB page table!

220 * 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