Virtual Memory

How is it possible for each process to have contiguous addresses and so many of them?


A System Using Physical Addressing

• Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing

• Used in all modern servers, desktops, and laptops
• One of the great ideas in computer science

Why Virtual Memory (VM)?

• Uses main memory efficiently
  • Use DRAM as a cache for the parts of a virtual address space

• Simplifies memory management
  • Each process gets the same uniform linear address space

• Isolates address spaces
  • One process can’t interfere with another’s memory
  • User program cannot access privileged kernel information

Address Spaces

• Linear address space: Ordered set of contiguous non-negative integer addresses:
  \[ \{0, 1, 2, 3, \ldots \} \]

• Virtual address space: Set of \( N = 2^n \) virtual addresses
  \[ \{0, 1, 2, 3, \ldots, N-1\} \]

• Physical address space: Set of \( M = 2^m \) physical addresses
  \[ \{0, 1, 2, 3, \ldots, M-1\} \]

• Clean distinction between data (bytes) and their attributes (addresses)
• Each object can now have multiple addresses
• Every byte in main memory: one physical address, one (or more) virtual addresses
VM as a Tool for Caching

- **Virtual memory** is an array of N contiguous bytes stored on disk.
- The contents of the array on disk are cached in **physical memory** (DRAM cache)
- These cache blocks are called pages (size is \( P = 2^p \) bytes)

**Physical memory**

<table>
<thead>
<tr>
<th>VP ( 0 )</th>
<th>VP ( 1 )</th>
<th>VP ( 2 )</th>
<th>VP ( 3 )</th>
<th>VP ( 4 )</th>
<th>VP ( 5 )</th>
<th>VP ( 6 )</th>
<th>VP ( 7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
</tr>
</tbody>
</table>

**Memory resident page table** (DRAM)

<table>
<thead>
<tr>
<th>PTE ( 0 )</th>
<th>PTE ( 1 )</th>
<th>PTE ( 2 )</th>
<th>PTE ( 3 )</th>
<th>PTE ( 4 )</th>
<th>PTE ( 5 )</th>
<th>PTE ( 6 )</th>
<th>PTE ( 7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
</tr>
</tbody>
</table>

**Virtual memory** (disk)

<table>
<thead>
<tr>
<th>VP ( 0 )</th>
<th>VP ( 1 )</th>
<th>VP ( 2 )</th>
<th>VP ( 3 )</th>
<th>VP ( 4 )</th>
<th>VP ( 5 )</th>
<th>VP ( 6 )</th>
<th>VP ( 7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
<td>Physical page number or disk address</td>
</tr>
</tbody>
</table>

**DRAM Cache Organization**

- DRAM cache organization driven by the enormous miss penalty
- DRAM is about \( 10x \) slower than SRAM
- Disk is about \( 10,000x \) slower than DRAM

**Consequences**

- Large page (block) size: typically 4-8 KB, sometimes 4 MB
- Fully associative
  - Any VP can be placed in any PP
- Requires a "large" mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms
  - Too complicated and open-ended to be implemented in hardware
- Write-back rather than write-through

**Page Tables**

- A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages.
- Per-process kernel data structure in DRAM

**Page Hit**

- **Page hit**: reference to VM word that is in physical memory (DRAM cache hit)

**Page Fault**

- **Page fault**: reference to VM word that is not in physical memory (DRAM cache miss)

**Handling Page Fault**

- Page miss causes page fault
Handling Page Fault
- Page miss causes page fault
- Page fault handler selects a victim to be evicted (here VP 4)

Locality to the Rescue Again!
- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the working set
  - Programs with better temporal locality will have smaller working sets
  - If (working set size < main memory size) Good performance for one process after compulsory misses
    - If ( SUM(working set sizes) > main memory size ) Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously

VM as a Tool for Memory Management
- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well chosen mappings simplify memory allocation and management
  - Memory allocation
    - Each virtual page can be mapped to any physical page
    - A virtual page can be stored in different physical pages at different times
    - Sharing code and data among processes
      - Map virtual pages to the same physical page (here: PP 6)
Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

- **Loading**
  - `execve()` allocates virtual pages for `.text` and `.data` sections
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

---

VM as a Tool for Memory Protection

- **Extend PTEs with permission bits**
  - Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

---

VM Address Translation

- **Virtual Address Space**
  - \( V = \{0, 1, ..., N-1\} \)

- **Physical Address Space**
  - \( P = \{0, 1, ..., M-1\} \)

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

---

Summary of Address Translation Symbols

- **Basic Parameters**
  - \( N = 2^n \): Number of addresses in virtual address space
  - \( M = 2^m \): Number of addresses in physical address space
  - \( P = 2^p \): Page size (bytes)

- **Components of the virtual address (VA)**
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number

- **Components of the physical address (PA)**
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag

---

Address Translation With a Page Table

- **Virtual address**
  - Page table: base register (PTBR)
  - Address Translation with a Page Table
Address Translation: Page Hit

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor

Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

Integrating VM and Cache

TLB Hit

A TLB hit eliminates a memory access

TLB Miss

A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare.

Speeding up Translation with a TLB

• Page table entries (PTEs) are cached in L1 like any other memory word
  • PTEs may be evicted by other data references
  • PTE hit still requires a small L1 delay
• Solution: 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
Multi-Level Page Tables

- Suppose:
  - 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE

- Problem:
  - Would need a 512 GB page table!
    - $2^n \times 2^{12} \times 2^{39}$ bytes

- Common solution:
  - Multi-level page tables
    - Example: 2-level page table
      - Level 1 table: each PTE points to a page table (always memory resident)
      - Level 2 table: each PTE points to a page (paged in and out like any other data)

Why Two-level Page Table Reduces Memory Requirement?

- If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist.
- Only the level 1 table needs to be in main memory at all times.
- The level 2 page tables can be created and paged in and out by the VM system as they are needed.

Simple Memory System Example

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

Review of Symbols

- Basic Parameters
  - $N = 2^n$: Number of addresses in virtual address space
  - $M = 2^n$: Number of addresses in physical address space
  - $P = 2^p$: Page size (bytes)

- Components of the virtual address (VA)
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number

- Components of the physical address (PA)
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag

Simple Memory System Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System TLB

- 16 entries
- 4-way associative

Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

Address Translation Example #1
Virtual Address: 0x03D4

Address Translation Example #2
Virtual Address: 0x0B8F

Address Translation Example #3
Virtual Address: 0x0020

Intel Core i7 Memory System
Processor package

Core 1
- 512 KB L1 data cache, 8-way
- 128 KB L1 instruction cache
- 256 KB L2 unified cache, 8-way
- 3 MB L3 unified cache

Core 2
- 512 KB L1 data cache, 8-way
- 128 KB L1 instruction cache
- 256 KB L2 unified cache, 8-way
- 3 MB L3 unified cache

Core 3
- 512 KB L1 data cache, 8-way
- 128 KB L1 instruction cache
- 256 KB L2 unified cache, 8-way
- 3 MB L3 unified cache

Core 4
- 512 KB L1 data cache, 8-way
- 128 KB L1 instruction cache
- 256 KB L2 unified cache, 8-way
- 3 MB L3 unified cache

QuickPath interconnect
- 4 links (PCI Express 32 Gbps)

Main memory
- 32 GB total
- Shared by all cores
Review of Symbols

• Basic Parameters
  • \( N = 2^n \): Number of addresses in virtual address space
  • \( M = 2^m \): Number of addresses in physical address space
  • \( P = 2^p \): Page size (bytes)

• Components of the virtual address (VA)
  • TLBI: TLB index
  • TLBT: TLB tag
  • VPO: Virtual page offset
  • VPN: Virtual page number

• Components of the physical address (PA)
  • PPO: Physical page offset (same as VPO)
  • PPN: Physical page number
  • CO: Byte offset within cache line
  • CI: Cache index
  • CT: Cache tag

End-to-end Core i7 Address Translation

Virtual Memory of a Linux Process

Linux Organizes VM as Collection of “Areas”

Memory Mapping

• VM areas initialized by associating them with disk objects.
  • Process is known as memory mapping.

• Area can be backed by (i.e., get its initial values from):
  • Regular file on disk (e.g., an executable object file)
    • Initial page bytes come from a section of a file
  • Anonymous file (e.g., nothing)
    • First fault will allocate a physical page full of 0’s (demand zero page)
    • Once the page is written to (allocated), it is like any other page

• Dirty pages are copied back and forth between memory and a special swap file.
Demand paging

• **Key point:** no virtual pages are copied into physical memory until they are referenced!
  • Known as *demand paging*

• Crucial for time and space efficiency

---

Sharing Revisited: Shared Objects

- Process 1 maps the shared object.
- Notice how the virtual addresses can be different.

---

Sharing Revisited: Private Copy-on-write (COW) Objects

- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

---

Sharing Revisited: Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!