Virtual Memory

Fall, 2015

Yusung Kim
yskim525@gmail.com
A System Using Physical Addressing

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

- Uses main memory efficiently
  - Address space of a process can exceed physical memory size
  - Sum of address spaces of multiple processes can exceed physical memory size

- Need to provide a simple memory management
  - Multiple process resident in main memory
  - 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 and code
A System Using Virtual Addressing

- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science
Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  \{0, 1, 2, 3 ... \}

- **Physical address space:** Set of \( M = 2^m \) physical addresses
  \{0, 1, 2, 3, ..., M-1\}

- **Virtual address space:** Set of \( N = 2^n \) virtual addresses
  \{0, 1, 2, 3, ..., N-1\}

  - 32-bit addresses: 4 billion (~4 \( \times 10^9 \)) bytes == 4 GBytes
  - 64-bit addresses: 16 quintillion (~16 \( \times 10^{18} \)) bytes
Virtual Memory for Caching

- Conceptually, **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)

![Diagram of virtual memory and physical memory](image)

**Virtual memory**

- VP 0
- VP 1
- VP $2^{n-p-1}$

**Physical memory**

- PP 0
- PP 1
- PP $2^{m-p-1}$

Virtual pages (**VPs**) stored on disk

Physical pages (**PPs**) cached in DRAM
DRAM Cache Organization

- **DRAM cache has a big impact in the memory hierarchy**
  - 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
  - Highly sophisticated, expensive replacement algorithms
  - Write-back rather than write-through
Page Table

- 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)

Virtual address

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Memory resident page table (DRAM)

Physical memory (DRAM)

- VP 1
- VP 2
- VP 7
- VP 4

Virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
Page Fault

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

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Memory resident page table (DRAM)

Virtual address

Physical memory (DRAM)

Virtual memory (disk)
Handling Page Fault

- Page miss causes page fault (an exception)
Handling Page Fault

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

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

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

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- The read instruction is restarted: page hit!

Key point: Waiting until the miss to copy the page to DRAM is known as demand paging
Locality to the Rescue Again!

- Virtual memory seems to be inefficient, but it 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
    - working set size < main memory size
      - Good performance with good temporal locality
    - working set sizes > main memory size
      - **Thrashing:** Performance meltdown where pages are swapped (copied) in and out continuously
Virtual Memory for Memory Management

- Key idea: each process has its own virtual address space
  - Each process can view memory as a simple linear array
  - Multiple virtual pages can be mapped to the same shared physical page

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation

Physical Address Space (DRAM)

(e.g., read-only library code)
Simplifying Linking and Loading

- Each program has similar virtual address space.
- Code, stack, and shared libraries always start at the same address.
- Every process has the same operating system kernel code, and routines in the standard C library such as printf.
- Contiguous virtual memory pages are mapped to arbitrary physical pages located anywhere.
Virtual Memory for Memory Protection

- A modern computer system must provide the means for the operation system to control access to the memory system.
  - Should not be allowed to modify read-only data
  - Should not be allowed to read or modify data in the kernel
  - Should not be allowed to read or write the private memory of other processes.

- Providing separate virtual address spaces makes it easy to isolate the private memories.

- Address translation mechanism can be extended to provide finer access control
Virtual Memory for Memory Protection

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

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

**Physical Address Space**

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
VM Address Translation

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

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

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

Virtual address

Virtual page number (VPN)  Virtual page offset (VPO)

Physical address

Page table

Physical page number (PPN)  Physical page offset (PPO)

Valid bit = 0: page not in memory (page fault)

Page table base register (PTBR)

Page table address for process

Valid

Physical page number (PPN)
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

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
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 Look a side Buffer (TLB)*
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

  TLBT matches tag of line within set

  TLB tag (TLBT) | TLB index (TLBI) | VPO

  \[ T = 2^t \text{ sets} \]

  TLBI selects the set
A TLB hit eliminates a memory access.
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
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!
    - $2^{48} \times 2^{-12} \times 2^3 = 2^{39}$ bytes

- **Common solution: Multi-level page table**

- **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)
A Two-Level Page Table Hierarchy

**32 bit addresses, 4KB pages, 4-byte PTEs**

- **Level 1 page table**
  - PTE 0
  - PTE 1
  - PTE 2 (null)
  - PTE 3 (null)
  - PTE 4 (null)
  - PTE 5 (null)
  - PTE 6 (null)
  - PTE 7 (null)
  - PTE 8
  - (1K - 9) null PTEs

- **Level 2 page tables**
  - PTE 0
  - ... PTE 1023
  - 1023 null PTEs
  - PTE 1023

- **Virtual memory**
  - VP 0
  - ... VP 1023
  - VP 1024
  - ... VP 2047
  - Gap
  - 1023 unallocated pages
  - VP 9215

- **Notes**:
  - 2K allocated VM pages for code and data
  - 6K unallocated VM pages
  - 1023 unallocated pages
  - 1 allocated VM page for the stack
Translating with a k-level Page Table

Page table base register (PTBR)

VIRTUAL ADDRESS

n-1

VPN 1  VPN 2  ...  VPN k

Level 1 page table  Level 2 page table  ...  Level k page table

m-1

...  ...  ...  ...  ...

PPN

PHYSICAL ADDRESS

p-1

0

VPN

PPN

0

VPO

p-1

0

VPN

PPN

...
Summary

- **Programmer’s view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions
Putting it Together: Address Translation

- Assumptions
  - Memory accesses are to **1-byte words** (not 4-byte words)
  - Virtual addresses are **14 bits** wide ($n = 14$)
  - Physical addresses are **12 bits** wide ($m = 12$)
  - The page size is 64 bytes ($P = 64$)
  - The TLB is **four-way** set associative with **16 total entries**
  - The L1 d-cache is physically addressed and direct mapped, with a **4-byte line size** and **16 total sets**.
### TLB: four sets, 16 entries, four-way set 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>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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

### Example

**Virtual address**: 0x03d4

**TLBT**: 0x03

**TLBI**: 0x03

<table>
<thead>
<tr>
<th>bit position</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>VPO</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tag**: 0x03d4

**Tag Bytes**: 0x03 0x0f

**Tag Bytes**: 0x03 0x14
The low-order 2 bits serve as the block offset (CO)
The next 4 bits serve as the set index (CI)
The remaining 6 bits serve as the tag (CT)