Virtual Memory: Concepts

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
Used by microcontrollers like those in Arduino, cars etc.

Simple but fragile to program for:
- Buffer overrun in buflab corrupts Firefox memory!
All problems in CS can be solved by another level of indirection

Butler Lampson, co-inventor of PC
A System Using Virtual Addressing

- Used in all modern servers, desktops, and laptops
Why Virtual Memory (VM)?

- **Simplified memory management**
  - Each process gets the same uniform linear address space

- **Process Isolation**
  - Different processes have different virtual address spaces
    - One process can’t interfere with another’s memory

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

- Address spaces
- VM as a tool for
  - Caching
  - memory management
  - protection
- Address translation
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)

![Diagram showing virtual memory and physical memory with cache states](image)
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about $10x$ slower than slowest 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 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**: VM data accessed is found in physical memory
Page Fault

- **Page fault:** VM data accessed is not found in physical memory
Handling Page Fault

- Page fault handler selects a victim to be evicted
Handling Page Fault

- Page fault handler selects a victim to be evicted
- Offending instruction is restarted: page hit!
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
Memory Management

- Each process has its own virtual address (VA) space
  - It can view memory as a simple linear array
  - One process cannot change another processes memory

![Diagram of virtual and physical address spaces](image)
Memory Management

- Sharing among processes
  - Map different virtual pages to the same physical page

![Diagram of memory management and address translation]

VA space for Process 1

VA space for Process 2

Physical Address Space (DRAM)

(read-only library code)
# 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()` causes kernel to allocate virtual pages
- Kernel copies `.text` and `.data` sections, page by page, from disk to memory
Memory Protection

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

- Address spaces
- VM as a tool for
  - caching
  - memory management
  - memory protection
- Address translation
Address translation

- Direct 1 to 1 mapping from Virtual Address to Physical Addresses

<table>
<thead>
<tr>
<th>VA</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>0x98</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
</tbody>
</table>
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)**
  - **PDBR**: Page directory base registers
  - **TLB**: Translation lookaside buffer
  - **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

How kernel tells hardware where to find the page table.

Page table base register

Page table address for process

Page table

Valid    Physical page number (PPN)

PTE (page table entry)

Physical page number (PPN)

Physical page offset (PPO)

Physical address

Virtual address

Virtual page number (VPN)

Virtual page offset (VPO)
Simple Memory System Example

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

![Diagram showing virtual and physical addresses with PPN, PPO, VPN, and VPO labels.](Diagram.png)
## Memory System Page Table Example

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Address Translation Example

Virtual Address: 0x0354

What’s the corresponding PPN? Physical address?
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
Speeding up translation with a Translation Lookaside Buffer (TLB)

A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Multilevel Page Tables

- **Suppose:**
  - 4KB \(2^{12}\) page size, 48-bit address space, 8-byte \(2^3\) PTE

- **Problem:**
  - Would need a 512 GB page table!
    - \(2^{48-12} \times 2^3 = 2^{39}\) bytes = 512GB

- **Solution:**
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table
    - Level 2 table: each PTE points to a page
A Two-Level Page Table Hierarchy

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

Virtual memory

- VP 0
- ... 
- VP 1023
- VP 1024
- ... 
- VP 2047
- Gap

- 1023 unallocated pages
- VP 9215
- 1023 unallocated VM pages
- 6K unallocated VM pages
- 2K allocated VM pages for code and data

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