Virtual Memory II
CSE 351 Winter 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Jonathan Chen
Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

https://xkcd.com/1495/
Administrivia

- Lab 4 – Due Monday (3/02)
  - Cache parameter puzzles and code optimizations

- hw19 due Wednesday (3/04)

- hw20 due Friday (3/06)
Mapping

- A virtual address (VA) can be mapped to either physical memory or disk
  - Unused VAs may not have a mapping
  - VAs from different processes may map to the same location in memory/disk
A System Using Physical Addressing

- Used in “simple” systems with (usually) just one process:
  - Embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

- Physical addresses are completely invisible to programs
  - Used in all modern desktops, laptops, servers, smartphones...
  - One of the great ideas in computer science
Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - Some non-cached parts stored on disk
    - Some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - Transfer data back and forth as needed

- **Simplifies memory management for programmers**
  - Each process “gets” the same full, private linear address space

- **Isolates address spaces (protection)**
  - One process can’t interfere with another’s memory
    - They operate in *different address spaces*
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions
VM and the Memory Hierarchy

- Think of virtual memory as array of \( N = 2^n \) contiguous bytes
- *Pages* of virtual memory are usually stored in physical memory, but sometimes spill to disk
  - Pages are another unit of aligned memory (size is \( P = 2^p \) bytes)
  - Each virtual page can be stored in *any* physical page (no fragmentation!)

![Diagram of virtual memory and physical memory](image)
Memory Hierarchy: Core 2 Duo

Not drawn to scale

Memory Hierarchy:

- **CPU**
  - Throughput: 16 B/cycle
  - Latency: 3 cycles

- **SRAM**
  - L1 I-cache: 32 KB
  - L1 D-cache
  - Throughput: 8 B/cycle
  - Latency: 14 cycles
  - L2 unified cache
  - Throughput: 2 B/cycle
  - Latency: 100 cycles
  - Miss Penalty (latency): 33x

- **DRAM**
  - Main Memory
  - Throughput: 1 B/30 cycles
  - Latency: millions
  - Miss Penalty (latency): 10,000x

- **Disk**
  - Throughput: ~500 GB
Virtual Memory Design Consequences

- Large page size: typically 4-8 KiB or 2-4 MiB
  - Can be up to 1 GiB (for “Big Data” apps on big computers)
  - Compared with 64-byte cache blocks

- Fully associative
  - Any virtual page can be placed in any physical page
  - Requires a “large” mapping function – different from CPU caches

- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware

- Write-back rather than write-through
  - Really don’t want to write to disk every time we modify something in memory
  - Some things may never end up on disk (e.g. stack for short-lived process)
Why does VM work on RAM/disk?

- Avoids disk accesses because of *locality*
  - Same reason that L1 / L2 / L3 caches work

- The set of virtual pages that a program is “actively” accessing at any point in time is called its *working set*
  - If *(working set of one process $\leq$ physical memory)*:
    - Good performance for one process (after compulsory misses)
  - If *(working sets of all processes $>$ physical memory)*:
    - *Thrashing*: Performance meltdown where pages are swapped between memory and disk continuously (CPU always waiting or paging)
    - This is why your computer can feel faster when you add RAM
Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- **Address translation**
- VM as a tool for memory management
- VM as a tool for memory protection
How do we perform the virtual → physical address translation?
Address Translation: Page Tables

CPU-generated address can be split into:

\[ \text{n-bit address: } \begin{array}{c|c}
\text{Virtual Page Number} & \text{Page Offset} \\
\end{array} \]

- Request is Virtual Address (VA), want Physical Address (PA)
- Note that Physical Offset = Virtual Offset (page-aligned)

Use lookup table that we call the page table (PT)

- Replace Virtual Page Number (VPN) for Physical Page Number (PPN) to generate Physical Address
- Index PT using VPN: page table entry (PTE) stores the PPN plus management bits (e.g. Valid, Dirty, access rights)
- Has an entry for every virtual page
Page Table Diagram

- Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- How many page tables in the system?
  - One per process
Page Table Address Translation

In most cases, the MMU can perform this translation without software assistance.
Polling Question

- How many bits wide are the following fields?
  - 16 KiB pages
  - 48-bit virtual addresses
  - 16 GiB physical memory
  - Vote at: [http://pollev.com/rea](http://pollev.com/rea)

<table>
<thead>
<tr>
<th></th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>34</td>
<td>24</td>
</tr>
<tr>
<td>(B)</td>
<td>32</td>
<td>18</td>
</tr>
<tr>
<td>(C)</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>(D)</td>
<td>34</td>
<td>20</td>
</tr>
</tbody>
</table>
Page Hit

- **Page hit**: VM reference is in physical memory

**Example**: Page size = 4 KiB

- **Virtual Addr**: `0x00740b`
- **VPN**:
- **PPN**:

**Physical Addr**:

- **Valid**
- **PPN/Disk Addr**

```
<table>
<thead>
<tr>
<th>PTE 0</th>
<th>0</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>PTE 7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

```
VP 1
VP 2
VP 7
VP 4
```

```
PP 0
PP 3
```

---

**Virtual memory (DRAM/disk)**

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

- **Page fault:** VM reference is NOT in physical memory

**Example:** Page size = 4 KiB
Provide a virtual address request (in hex) that results in this particular page fault:

**Virtual Addr:**
Reminder: Page Fault Exception

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

Page fault handler must load page into physical memory

Returns to faulting instruction: `mov` is executed again!
  - Successful on second try

```c
int a[1000];
int main () {
    a[500] = 13;
}
```

```asm
80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10
```

Diagram:
- User code: `movl` → _exception: page fault_ → OS Kernel code: handle_page_fault: _Create page and load into memory_
  - Returns
Handling a Page Fault

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

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

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

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
- Offending instruction is restarted: page hit!
Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection
VM for Managing Multiple Processes

- Key abstraction: each process has its own virtual address space
  - It can view memory as *a simple linear array*
- With virtual memory, this simple linear virtual address space need not be contiguous in physical memory
  - Process needs to store data in another VP? Just map it to *any* PP!

![Address translation diagram]

Virtual Address Space for Process 1:
- VP 1
- VP 2
- ...
- \( \text{N-1} \) (e.g., read-only library code)

Virtual Address Space for Process 2:
- VP 1
- VP 2
- ...
- \( \text{N-1} \) (same as Process 1)

Physical Address Space (DRAM):
- PP 2
- PP 6
- PP 8
- ...
- \( \text{M-1} \) (e.g., read-only library code)
Simplifying Linking and Loading

**Linking**
- Each program has similar virtual address space
- Code, Data, and Heap always start at the same addresses

**Loading**
- `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

![Diagram of virtual memory segments](image)
VM for Protection and Sharing

- The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes
  - **Sharing**: map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection**: process can’t access physical pages to which none of its virtual pages are mapped (here: Process 2 can’t access PP 2)
Memory Protection Within Process

- VM implements read/write/execute permissions
  - Extend page table entries with permission bits
  - MMU checks these permission bits on every memory access
    - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)

<table>
<thead>
<tr>
<th>Process (i):</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</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>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
**Review Question**

- What should the permission bits be for pages from the following sections of virtual memory?

<table>
<thead>
<tr>
<th>Section</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Literals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>