Virtual Memory II
CSE 351 Spring 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou

https://xkcd.com/1495/
Administrivia

- Lab 4 – Due Friday 5/22
  - Cache parameter puzzles and code optimizations

- You must log on with your @uw google account to access!!
  - Google doc for 11:30 Lecture: https://tinyurl.com/351-05-18A
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 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!)

Virtual memory

<table>
<thead>
<tr>
<th>Virtual pages (VP's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0</td>
</tr>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP ( 2^{n-p} ) - 1</td>
</tr>
</tbody>
</table>

Physical memory

<table>
<thead>
<tr>
<th>Physical pages (PP's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
</tr>
<tr>
<td>PP 1</td>
</tr>
<tr>
<td>PP ( 2^{m-p} ) - 1</td>
</tr>
</tbody>
</table>

Disk

"Swap Space"
**or: Virtual Memory as DRAM Cache for Disk**

- Think of virtual memory as an array of $N = 2^n$ contiguous bytes stored *on a disk*
- Then physical main memory is used as a *cache* for the virtual memory array
  - These “cache blocks” are called *pages* (size is $P = 2^p$ bytes)

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

Not drawn to scale

CPU Reg

Throughput: 16 B/cycle
Latency: 3 cycles

SRAM
Static Random Access Memory

L1 I-cache
32 KB

L1 D-cache

DRAM
Dynamic Random Access Memory

~4 MB
L2 unified cache

~8 GB
Main Memory

Disk
~500 GB

Miss Penalty (latency)
33x

Miss Penalty (latency)
10,000x

1 B/30 cycles
millions

2 B/cycle
100 cycles

8 B/cycle
14 cycles

Not drawn to scale
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 (physical memory is single set)
  - 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 (track dirty pages)
  - *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 $> \text{physical memory}$):
    - **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
Address Translation

How do we perform the virtual → physical address translation?
Address Translation: Page Tables

CPU-generated address can be split into:

- 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 Table (DRAM)**

<table>
<thead>
<tr>
<th>Virtual page # (VPN)</th>
<th>Valid</th>
<th>PPN/Disk Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0: 0</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>PTE 1: 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PTE 2: 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PTE 3: 3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PTE 4: 4</td>
<td>0</td>
<td>disk addr</td>
</tr>
<tr>
<td>PTE 5: 5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>PTE 6: 6</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>PTE 7: 7</td>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

**Physical memory (DRAM)**

- VP 1
- VP 2
- VP 7

**Physical page #**

- PP 0
- PP 1
- PP 2
- PP 3

**Virtual memory (DRAM/disk)**

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

#### Page Table has $2^n$ entries!

- 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

CPU

Page table base register (PTBR)

Page table address for process

Virtual address (VA)

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

Page table

Valid  PPN

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

Page table address at VPN entry

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

Physical address (PA)

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

changed on a context switch

n bits

m bits
Polling Question [VM II]

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

---

### Page Table (DRAM)

<table>
<thead>
<tr>
<th>Valid</th>
<th>PPN/Disk Addr</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>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PPN: 2</td>
</tr>
</tbody>
</table>

---

**Example**: Page size = 4 KiB

- **Virtual Addr**: 0x00740b
- **VPN**: 7
- **Physical Addr**: 0x602406
- **PPN**: 2

---

**Physical Memory (DRAM)**

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

**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: `0x003`
Reminder: Page Fault Exception

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

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

![Diagram](image)

- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try
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!

![Virtual Address Space Diagram]

---

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation

Virtual Address Space (DRAM)
(e.g., read-only library code)

Physical Address Space

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

N-1

M-1
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
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)
**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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Heap</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Static Data</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Literals</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Instructions</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*Note: Static data is considered in size, and literals are constant.*