Virtual Memory I
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://rebrn.com/re/bad-chrome-1162082/
Administrivia

- Homework 3 due tonight @ 11:45pm
- Lab 4 due Monday, Nov. 28 (after break)

Thanksgiving Break

- Will try to record next Wednesday’s lecture
  - Also see recorded 351 videos on Virtual Memory
- Next week will be a “virtual section” – worksheet and solutions posted on website
Roadmap

C:
```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```
get_mpg:
    pushq  %rbp
    movq  %rsp, %rbp
    ...
    popq  %rbp
    ret
```

Machine code:
```
0111010000011000
1000110100000100
1000110000001100
1001001110000010
110000011111101000011111
```

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8
Mac
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

**Warning:** VM is pretty complex, but crucial for understanding how processes work and for debugging performance.

*Not to be confused with “Virtual Machine” which is a whole other thing.*
Memory as we know it so far... is virtual!

- Programs refer to virtual memory addresses
  - `movq (%rdi),%rax`
  - Conceptually memory is just a very large array of bytes
  - System provides private address space to each process

- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space

- But...
  - We probably don’t have $2^w$ bytes of physical memory
  - We certainly don’t have $2^w$ bytes of physical memory for every process
  - Processes should not interfere with one another
    - Except in certain cases where they want to share code or data
Problem 1: How Does Everything Fit?

64-bit virtual addresses can address several exabytes
(18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes
(e.g. 8,589,934,592 bytes)

(Not to scale; physical memory would be smaller than the period at the end of this sentence compared to the virtual address space.)

1 virtual address space per process, with many processes...
Problem 2: Memory Management

We have multiple processes:

Process 1
Process 2
Process 3
...
Process n

Each process has...

- stack
- heap
- .text
- .data
- ...

What goes where?

Physical main memory
Problem 3: How To Protect

Problem 4: How To Share?
How can we solve these problems?

1) Fitting a huge address space into a tiny physical memory
2) Managing the address spaces of multiple processes
3) Protecting processes from stepping on each other’s memory
4) Allowing processes to share common parts of memory
Indirection

- “Any problem in computer science can be solved by adding another level of indirection.” –David Wheeler, inventor of the subroutine

- Without Indirection

- With Indirection

What if I want to move Thing?
Indirection

- **Indirection**: The ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.
  - Adds some work ("overhead"; now have to look up 2 things instead of 1)
  - But don’t have to track all uses of name/address (single source!)

- **Examples**:
  - **911**: routed to local office
  - **Call centers**: route calls to available operators, etc.
  - **Phone system**: cell phone number portability
  - **Domain Name Service (DNS)**: translation from name to IP address
  - **Dynamic Host Configuration Protocol (DHCP)**: local network address assignment
Indirection in Virtual Memory

- Each process gets its own private virtual address space
- Solves the previous problems!
Address Spaces

- **Virtual address space**: Set of $N = 2^n$ virtual addr
  - $\{0, 1, 2, 3, \ldots, N-1\}$

- **Physical address space**: Set of $M = 2^m$ physical addr
  - $\{0, 1, 2, 3, \ldots, M-1\}$

- Every byte in main memory has:
  - one physical address (PA)
  - zero, one, *or more* virtual addresses (VAs)
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 an 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)

- "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 showing virtual memory and physical memory with various states: Unallocated, Cached, Uncached, Empty, and Physical pages (PPs) cached in DRAM.](attachment:virtual_memory_diagram.png)
Memory Hierarchy: Core 2 Duo

Not drawn to scale

CPU

Reg

SRAM
Static Random Access Memory

L1 I-cache

L1 D-cache

32 KB

L2 unified cache

~4 MB

DRAM
Dynamic Random Access Memory

Main Memory

~8 GB

Disk
~500 GB

Throughput:
16 B/cycle

Latency:
3 cycles

8 B/cycle
14 cycles

2 B/cycle
100 cycles

1 B/30 cycles
millions

Miss Penalty (latency)
33x

Miss Penalty (latency)
10,000x
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* ≤ *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
Address Translation

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

- CPU-generated address can be split into:
  
  \[ n\text{-bit address: } \begin{array}{c|c}
  \text{Virtual Page Number} & \text{Page Offset} \\
  \end{array} \]

  - 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 – why?
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)

**Virtual address (VA)**
- Virtual page number (VPN)
- Virtual page offset (VPO)

**Page table**
- Valid
- PPN

**Physical address (PA)**
- Physical page number (PPN)
- Physical page offset (PPO)

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

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

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

Example: Page size = 4 KiB

Virtual Addr: \(0x00740b\)  
Physical Addr:

VPN:  
PPN:
**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:**
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;
}
```

![Image with assembly code]

- 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)

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>PPN/Disk Addr</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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
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!

### 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>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Physical memory (DRAM)

- VP 0
- VP 3

### Virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
Peer Instruction Question

- How many bits wide are the following fields?
  - 16 KiB pages
  - 48-bit virtual addresses
  - 16 GiB physical memory

<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>
Summary

- Virtual memory provides:
  - Ability to use limited memory (RAM) across multiple processes
  - Illusion of contiguous virtual address space for each process
  - Protection and sharing amongst processes

- Indirection via address mapping by page tables
  - Part of memory management unit and stored in memory
  - Use virtual page number as index into lookup table that holds physical page number, disk address, or NULL (unallocated page)
  - On page fault, throw exception and move page from swap space (disk) to main memory