Virtual Memory I
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Midterms Graded
  - If you did not receive an email from Gradescope let us know
- Homework 4 – Due this Friday 5/19
  - Cache questions
- Lab 4 – Due Tuesday 5/23
  - Cache runtimes and parameter puzzles
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:

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

Machine code:

```
0111010000011000 100011010000010000000010
1000100111000010 110000011111101000011111
```

Computer system:

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

OS:

- Windows 8
- Mac
- Linux
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:** Virtual memory 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

Physical main memory

Process $i$

Process $j$

Problem 4: How To Share?

Physical main memory

Process $i$

Process $j$
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 (now have to look up 2 things instead of 1)
  - But don’t have to track all uses of name/address (single source!)

- **Examples**:
  - **Phone system**: cell phone number portability
  - **Domain Name Service (DNS)**: translation from name to IP address
  - **Call centers**: route calls to available operators, etc.
  - **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, ..., N-1}

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

```
Process 1’s Virtual Address Space
      ▶️
      ▶️
      ▶️
      ▶️

Physical Memory

Process 2’s Virtual Address Space
      ▶️
      ▶️
      ▶️
      ▶️

Disk

“Swap Space”
```
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
    VP 0  Unallocated
    VP 1  Unallocated
        VP 2^{n-p-1}

  Physical memory
      0     Empty
      PP 0  Empty
      PP 1  Empty
          PP 2^{m-p-1}

  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)
Memory Hierarchy: Core 2 Duo

Not drawn to scale

SRAM
Static Random Access Memory

L1 I-cache
32 KB

L1 D-cache

L2 unified cache
~4 MB

Main Memory

DRAM
Dynamic Random Access Memory

~8 GB

Disk
~500 GB

Throughput:
16 B/cycle
Latency:
3 cycles

Miss Penalty (latency)
33x

8 B/cycle
14 cycles

2 B/cycle
100 cycles

1 B/30 cycles
millions

Miss Penalty (latency)
10,000x

CPU Reg

Miss Penalty (latency)

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?

CPU Chip

Virtual address (VA) 0x4100

MMU

Physical address (PA) 0x4

Memory Management Unit

Main memory

0:  
1:  
2:  
3:  
4:  
5:  
6:  
7:  
8:  
...

M-1:

Data (int/float)
Address Translation: Page Tables

- CPU-generated address can be split into:

  $n$-bit address: \begin{tabular}{|c|c|}
  \hline
  Virtual Page Number & Page Offset \\
  \hline
\end{tabular}

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

- Use lookup table that we call the \textit{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 \textit{every} virtual page – why?
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.
Page Hit

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

**Example**: Page size = 4 KiB

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

**Virtual memory (DRAM/disk)**:
- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6

**Physical memory (DRAM)**:
- VP 0
- VP 1
- VP 2
- VP 3

**Page Table (DRAM)**

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

**Page Table**:

- Valid: 1 indicates the page is present in physical memory.
- PPN/Disk Addr: Physical page number.
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: beforehand
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;
}
```

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

User code

OS Kernel code

- `movl` exception: page fault
- `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)

![Diagram of page table and physical memory]

- **Virtual address**
- **Page Table (DRAM)**
  - Valid
  - PPN/Disk Addr
  - PTE 0:
    - 0: null
    - 1
  - PTE 7:
    - 0
    - 0: null
    - 1
    - ...: ...

- **Physical memory (DRAM)**
  - PP 0
  - PP 3
  - VP 1
  - VP 2
  - VP 3
  - VP 7

- **Virtual memory (DRAM/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)
- Offending instruction is restarted: page hit!
Question

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

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