Virtual Memory II
CSE 351 Winter 2018

Instructor: Mark Wyse
Teaching Assistants: Kevin Bi, Parker DeWilde, Emily Furst, Sarah House, Waylon Huang, Vinny Palaniappan

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

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation

Physical Address Space (DRAM)

PP

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

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 PPI!

Review: Terminology
❖ Context switch
❖ Switch between processes on the same CPU
❖ Page in
❖ Move pages of virtual memory from disk to physical memory
❖ Page out
❖ Move pages of virtual memory from physical memory to disk
❖ Thrashing
❖ Total working set size of processes is larger than physical memory and causes excessive paging in and out instead of doing useful computation

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

Administrative
❖ Lab 4 due Wednesday by 11:59 pm!
❖ Homework 5 released today, due 3/7
❖ Processes and Virtual Memory
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)

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)

Address Translation: Page Hit

1) Processor sends virtual address to MMU (memory management unit)
2) MMU fetches PTE from page table in cache/memory (uses PTBR to find beginning of page table for current process)
3) MMU sends physical address to cache/memory requesting data
4) Cache/memory sends data to processor

Address Translation: Page Fault

1) Processor sends virtual address to MMU
2) MMU fetches PTE from page table in cache/memory (uses PTBR to find beginning of page table for current process)
3) Valid bit is zero, so MMU triggers page fault exception
4) Handler identifies victim (and, if dirty, pages it out to disk)
5) Handler pages in new page and updates PTE in memory
6) Handler returns to original process, restarting faulting instruction

Hmm... Translation Sounds Slow

- The MMU accesses memory twice: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs may be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles
- What can we do to make this faster?
  - Solution: add another cache!

Speeding up Translation with a TLB

- Translation Lookaside Buffer (TLB):
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory
A TLB hit eliminates a memory access!

A TLB miss incurs an additional memory access (the PTE)
- Fortunately, TLB misses are rare

## Fetching Data on a Memory Read

1. Check TLB
   - Input: VPN, Output: PPN
   - TLB Hit: Fetch translation, return PPN
   - TLB Miss: Check page table (in memory)
     - Page Table Hit: Load page table entry into TLB
     - Page Fault: Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

2. Check cache
   - Input: physical address, Output: data
   - Cache Hit: Return data value to processor
   - Cache Miss: Fetch data value from memory, store it in cache, return it to processor

## Address Translation

- Virtual Address
  - TLB Miss
  - TLB Hit
  - Page Fault
  - Protection Check
  - Page Table "Walk"
  - Update TLB
  - Protection Fault
  - Physical Address

- Physical Address
  - Find in Disk
  - Find in Mem
  - SIGSEGV
  - Check cache

## Context Switching Revisited

- What needs to happen when the CPU switches processes?
  - Registers:
    - Save state of old process, load state of new process
    - Including the Page Table Base Register (PTBR)
  - Memory:
    - Nothing to do! Pages for processes already exist in memory/disk and protected from each other
  - TLB:
    - Invalidate all entries in TLB — mapping is for old process’ VAs
  - Cache:
    - Can leave alone because storing based on PAs — good for shared data

## 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)
  - VPO Virtual page offset
  - VPN Virtual page number
  - TLBI TLB index
  - TLBT TLB tag

- Components of the physical address (PA)
  - PPO Physical page offset (same as VPO)
  - PPN Physical page number
Simple Memory System Example (small)

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

Page table (partial):

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>02</td>
<td>0</td>
</tr>
<tr>
<td>0 2</td>
<td>03</td>
<td>0</td>
</tr>
<tr>
<td>0 3</td>
<td>04</td>
<td>0</td>
</tr>
</tbody>
</table>

Virtual Page Offset
Physical Page Offset

Simple Memory System: TLB

- 16 entries total
- 4-way set associative

Why does the TLB ignore the page offset?

Current State of Memory System

Page table (partial):

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>02</td>
<td>0</td>
</tr>
<tr>
<td>0 2</td>
<td>03</td>
<td>0</td>
</tr>
<tr>
<td>0 3</td>
<td>04</td>
<td>0</td>
</tr>
</tbody>
</table>

Virtual Address: 0x03D4

Physical Request Example #1

Virtual Address: 0x03D4

Physical Address:

Memory Request Example #1

VPN
PPO
Memory Request Example #2

Virtual Address: 0x038F

Physical Address: T

Note: It is just coincidence that the PPN is the same width as the cache Tag

Memory Request Example #3

Virtual Address: 0x0020

Physical Address: T

Note: It is just coincidence that the PPN is the same width as the cache Tag

Memory Request Example #4

Virtual Address: 0x036B

Physical Address: T

Note: It is just coincidence that the PPN is the same width as the cache Tag

Virtual Memory 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 sharing
- Simplifies protection by providing permissions checking

Memory System Summary

Memory Caches (L1/L2/L3)
- Purely a speed-up technique
- Behavior invisible to application programmer and (mostly) OS
- Implemented totally in hardware

Virtual Memory
- Supports many OS-related functions
- Process creation, task switching, protection
- Operating System (software)
  - Allocates/shares physical memory among processes
  - Maintains high-level tables tracking memory type, source, sharing
  - Handles exceptions, fills in hardware-defined mapping tables
- Hardware
  - Translates virtual addresses via mapping tables, enforcing permissions
  - Accelerates mapping via translation cache (TLB)