CSE 153
Design of Operating Systems
Winter 2018

Lecture 16: Paging/Virtual Memory (1)

Some slides modified from originals by Dave O’hallaron
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
A System Using Physical Addressing

- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science
Address Spaces

- **Linear address space**: Ordered set of contiguous non-negative integer addresses:
  \( \{0, 1, 2, 3 \ldots \} \)

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

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

- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses
Why Virtual Memory (VM)?

- Virtual memory is page with a new ingredient
  - Allow pages to be on disk
    » In a special partition (or file) called swap

- Motivation?
  - Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space

- Simplifies memory management
  - Each process gets the same uniform linear address space
  - With VM, this can be big!
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
**VM as a Tool for Caching**

- *Virtual memory* is an array of N contiguous bytes stored on disk.
- The contents of the array on disk are cached in *physical memory* (DRAM cache)
  - These cache blocks are called *pages* (size is $P = 2^p$ bytes)

![Diagram showing virtual and physical memory]

- Virtual pages (VPs) stored on disk
- Physical pages (PPs) cached in DRAM
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about $10x$ slower than SRAM
  - Disk is about $10,000x$ slower than DRAM

- Consequences
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through
A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages.
- Per-process kernel data structure in DRAM

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>Valid</th>
<th>Physical page number or disk address</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>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTE 7</th>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Physical memory (DRAM)**
  - PP 0: VP 1, VP 2, VP 7, VP 4
  - PP 3: VP 1, VP 2, VP 3, VP 4, VP 6, VP 7

- **Virtual memory (disk)**
  - VP 1, VP 2, VP 3, VP 4, VP 6, VP 7
**Page Hit**

- **Page hit**: reference to VM word that is in physical memory (DRAM cache hit)
Page Fault

- **Page fault**: reference to VM word that is not in physical memory (DRAM cache miss)
Handling Page Fault

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

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

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling 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!
Locality to the Rescue!

- Virtual memory works because of locality

- At any point in time, programs tend to access a set of active virtual pages called the working set
  - Programs with better temporal locality will have smaller working sets

- If (working set size < main memory size)
  - Good performance for one process after compulsory misses

- If (SUM(working set sizes) > main memory size)
  - Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM as a Tool for Memory Management

Key idea: each process has its own virtual address space
- It can view memory as a simple linear array
- Mapping function scatters addresses through physical memory
  » Well chosen mappings simplify memory allocation and management

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:
VM as a Tool for Memory Management

- Memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)
Sharing

- Can map shared memory at same or different virtual addresses in each process’ address space
  - Different:
    » 10th virtual page in P1 and 7th virtual page in P2 correspond to the 2nd physical page
    » Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same:
    » 2nd physical page corresponds to the 10th virtual page in both P1 and P2
    » Less flexible, but shared pointers are valid
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()

- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction

- How does this help fork()?
Execution of fork()

Parent process’s page table

Page 1
Page 2

Child process’s page table

Page 1
Page 2

Physical Memory
fork() with Copy on Write

When either process modifies Page 1, page fault handler allocates a new page and updates PTE in child process.

Parent process’s page table

<table>
<thead>
<tr>
<th>Page 1</th>
<th>Page 2</th>
</tr>
</thead>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>Page 1</th>
<th>Page 2</th>
</tr>
</thead>
</table>

Child process’s page table

Protection bits set to prevent either process from writing to any page.
Simplifying Linking and Loading

**Linking**
- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

**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 memory regions and virtual addresses](image)
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

Physical Address Space
- PP 2
- PP 4
- PP 6

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM Address Translation

- Virtual Address Space
  - \( V = \{0, 1, \ldots, N-1\} \)

- Physical Address Space
  - \( P = \{0, 1, \ldots, M-1\} \)

- Address Translation
  - \( MAP: V \rightarrow P \cup \{\varnothing\} \)
  - For virtual address \( a \):
    - \( MAP(a) = a' \) if data at virtual address \( a \) is at physical address \( a' \) in \( P \)
    - \( MAP(a) = \varnothing \) if data at virtual address \( a \) is not in physical memory
      - Either invalid or stored on disk
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)**
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number

- **Components of the physical address (PA)**
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag
Address Translation With a Page Table

Virtual address

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

Page table

Valid  Physical page number (PPN)

Physical address

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

Page table base register (PTBR)

Page table address for process

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

Valid bit = 0: page not in memory (page fault)
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Integrating VM and Cache

CPU Chip

CPU

MMU

PTEA

PTE

Memory

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Elephant(s) in the room

- Problem 1: Translation is slow!
  - Many memory accesses for each memory access
  - Caches are useless!

- Problem 2: Page table can be gigantic!
- We need one for each process
- All your memory are belong to us!

“Unfortunately, there’s another elephant in the room.”