Virtual Memory: Concepts

15-213: Introduction to Computer Systems
16th Lecture, Oct. 22, 2013

Instructors:
Randy Bryant, Dave O’Hallaron, and Greg Kesden
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 \}\n
- **Virtual address space:** Set of \(N = 2^n\) virtual addresses
  
  \{0, 1, 2, 3, \ldots, N-1\}\n
- **Physical address space:** Set of \(M = 2^m\) physical addresses
  
  \{0, 1, 2, 3, \ldots, M-1\}\n
- 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)?

- **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

- **Isolates address spaces**
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information
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 memory and physical memory mapping]

- 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
Page Tables

- 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>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Physical page number or disk address

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

Memory resident page table (DRAM)
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 Again!

- 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:

Address translation

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

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation:

Physical Address Space (DRAM):
(e.g., read-only library code)
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
  - `execve()` = creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

---

**Memory Mapping**
- Kernel virtual memory
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by `malloc`)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused

---

- Memory invisible to user code
- `%esp` (stack pointer)
- `brk`
- Loaded from the executable file
- Unused

---

- Memory map shown at `0x00000000`
- User stack space at `0x00000000`
- Memory-mapped region for shared libraries at `0x04000000`
- Run-time heap at `0x08048000`
- Read/write segment at `0x08048000`
- Read-only segment at `0x08048000`
- Unused at `0x08048000`

---

- Memory map shown at `0x00000000`
- User stack space at `0x00000000`
- Memory-mapped region for shared libraries at `0x04000000`
- Run-time heap at `0x08048000`
- Read/write segment at `0x08048000`
- Read-only segment at `0x08048000`
- Unused at `0x08048000`
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)
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**
  - $MAP: V \rightarrow P \cup \{\emptyset\}$
  - For virtual address $a$:
    - $MAP(a) = a'$ if data at virtual address $a$ is at physical address $a'$ in $P$
    - $MAP(a) = \emptyset$ 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 bit = 0: page not in memory (page fault)

Page table address for process

Page table base register (PTBR)

Physical address

Physical page number (PPN)  Physical page offset (PPO)
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
Address Translation: Page Fault

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

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

- Solution: *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
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Multi-Level Page Tables

- **Suppose:**
  - 4KB \(2^{12}\) page size, 48-bit address space, 8-byte PTE

- **Problem:**
  - Would need a 512 GB page table!
    - \(2^{48} \times 2^{-12} \times 2^3 = 2^{39}\) bytes

- **Common solution:**
  - Multi-level page tables
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table (always memory resident)
    - Level 2 table: each PTE points to a page (paged in and out like any other data)
A Two-Level Page Table Hierarchy

Level 1
page table

Level 2
page tables

Virtual
memory

PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8
(1K - 9)
null PTEs

PTE 0
... PTE 1023

PTE 0
... PTE 1023

PTE 0
... PTE 1023

0

VP 0
... VP 1023
VP 1024
... VP 2047
Gap

1023 unallocated
pages
VP 9215

1023 null
PTEs
PTE 1023

6K unallocated VM pages

2K allocated VM pages for code and data

1 allocated VM page for the stack

1023 unallocated pages

32 bit addresses, 4KB pages, 4-byte PTEs
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 programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions