Virtual Memory: Concepts

CS351: Systems Programming
Day 19: Oct. 27, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and 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, laptops, and smart phones
- 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
  \[ \{0, 1, 2, 3, \ldots, M-1\} \]
Why Virtual Memory (VM)?

- **Uses main memory efficiently**
  - Use DRAM as a cache for 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 and code
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

- Conceptually, *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 memory

- VP 0
  - Unallocated
  - Cached
  - Uncached
- VP 1
  - Unallocated
  - Cached
  - Uncached
- VP $2^{n-p-1}$
  - Cached
  - Uncached

Virtual pages (VPs) stored on disk

Physical memory

- PP 0
  - Empty
- PP 1
  - Empty
- PP $2^{m-p-1}$
  - Empty

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 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - 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>Memory resident page table (DRAM)</th>
<th>Physical memory (DRAM)</th>
<th>Virtual memory (disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0: 0 null</td>
<td>PP 0: VP 1</td>
<td>VP 1</td>
</tr>
<tr>
<td>1</td>
<td>PP 3: VP 2</td>
<td>VP 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>VP 3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>VP 4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>VP 5</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>VP 6</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>VP 7</td>
</tr>
</tbody>
</table>
Page Hit

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

![Diagram of virtual memory and page tables]
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!

Key point: Waiting until the miss to copy the page to DRAM is known as **demand paging**
Allocating Pages

Allocating a new page (VP 5) of virtual memory.
Locality to the Rescue Again!

- Virtual memory seems terribly inefficient, but it 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 can improve locality

![Diagram showing virtual address space for Process 1 and Process 2, with address translation between virtual and physical address spaces.](image-url)
VM as a Tool for Memory Management

- **Simplifying 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)

<table>
<thead>
<tr>
<th>Virtual Address Space for Process 1:</th>
<th>Virtual Address Space for Process 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VP 1</td>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
<td>VP 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Address Space (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>PP 2</td>
</tr>
<tr>
<td>PP 6</td>
</tr>
<tr>
<td>PP 8</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Address translation

(e.g., read-only library code)
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
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
- MMU checks these bits on each access

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

<table>
<thead>
<tr>
<th>Physical Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 2</td>
</tr>
<tr>
<td>PP 4</td>
</tr>
<tr>
<td>PP 6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Physical Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 8</td>
</tr>
<tr>
<td>PP 9</td>
</tr>
<tr>
<td>PP 11</td>
</tr>
</tbody>
</table>

Illinois Tech CS351 Fall 2022
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
Address Translation With a Page Table

**Virtual address**

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

**Physical address**

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

- **Valid bit = 0:** Page not in memory (page fault)
- **Valid bit = 1:**

**Page table**

- **Page table base register (PTBR)**
- **Physical page table address for the current process**
Address Translation: Page Hit

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

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 set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

  TLBT matches tag of line within set

  TLBI selects the set

  T = 2^t sets
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 table**

- **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

32 bit addresses, 4KB pages, 4-byte PTEs
Translating with a k-level Page Table

Page table base register (PTBR)

VIRTUAL ADDRESS

PHYSICAL ADDRESS

Level 1
page table

Level 2
page table

Level k
page table
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
Next week: recorded lectures

<table>
<thead>
<tr>
<th>Oct 31</th>
<th>Nov 01</th>
<th>Nov 02</th>
<th>Nov 03</th>
</tr>
</thead>
</table>
| LAB    | LEC 20: Virtual Memory: Systems  
Preparation: Read CS:APP 9.7-9.8 | Nov 02 | LEC 21: Storage Allocation: Basic  
Preparation: Read CS:APP 9.9 |

<table>
<thead>
<tr>
<th>Nov 07</th>
<th>Nov 08</th>
<th>Nov 09</th>
<th>Nov 10</th>
</tr>
</thead>
</table>
| LAB    | LEC 22: Storage Allocation: Advanced  
Preparation: Read CS:APP 9.9-9.11  
Assigned: Lab 5: Malloc lab | Nov 09 | LEC 23: Network Programming: Part 1  
Preparation: Read CS:APP 11.1-11.4 |

- **LEC 20** and **LEC 21** will be pre-recorded and circulated on Blackboard.
  - Do not come to SB104 those days – there will not be an in-person lecture.
  - My away-at-a-conference days are marked on the course calendar.
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- The form is anonymous. (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- https://forms.gle/qoeEbBuTYXo5FiU1A
- I’ll remind about this at each lecture.