Virtual Memory

Computer Systems Organization (Spring 2015)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska
Teaching Assistants: Paige Connelly & Carlos Guzman

Slides adapted from
Andrew Case, Jinyang Li, Mohamed Zahran, Stewart Weiss, Randy Bryant and Dave O'Hallaron
How is it possible for each process to have contiguous addresses and so many of them?

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 ... \}

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

• **Physical address space:** Set of \(M = 2^m\) physical addresses
  \{0, 1, 2, 3, ..., 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)?

• 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
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 states]

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

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

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

![Diagram of page tables]

<table>
<thead>
<tr>
<th>Physical page number or disk address</th>
<th>Valid</th>
<th>Memory resident page table (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
<td>0</td>
<td>VP 1</td>
</tr>
<tr>
<td>PP 0</td>
<td>1</td>
<td>VP 2</td>
</tr>
<tr>
<td>PP 0</td>
<td>1</td>
<td>VP 7</td>
</tr>
<tr>
<td>PP 0</td>
<td>0</td>
<td>VP 4</td>
</tr>
<tr>
<td>PP 3</td>
<td>0</td>
<td>VP 1</td>
</tr>
<tr>
<td>PP 3</td>
<td>1</td>
<td>VP 2</td>
</tr>
<tr>
<td>PP 3</td>
<td>1</td>
<td>VP 3</td>
</tr>
<tr>
<td>PP 3</td>
<td>0</td>
<td>VP 4</td>
</tr>
<tr>
<td>PP 3</td>
<td>0</td>
<td>VP 6</td>
</tr>
<tr>
<td>PP 3</td>
<td>0</td>
<td>VP 7</td>
</tr>
</tbody>
</table>
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
Handling Page Fault

• Page miss causes page fault
• 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!

<table>
<thead>
<tr>
<th>Physical page number or disk address</th>
<th>Valid</th>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
<td></td>
<td>VP 0</td>
<td>VP 1</td>
</tr>
<tr>
<td>PP 3</td>
<td></td>
<td>VP 2</td>
<td>VP 7</td>
</tr>
<tr>
<td>VP 3</td>
<td></td>
<td>VP 3</td>
<td>VP 3</td>
</tr>
<tr>
<td>VP 4</td>
<td></td>
<td>VP 4</td>
<td>VP 4</td>
</tr>
<tr>
<td>VP 6</td>
<td></td>
<td>VP 6</td>
<td>VP 6</td>
</tr>
<tr>
<td>VP 7</td>
<td></td>
<td>VP 7</td>
<td>VP 7</td>
</tr>
</tbody>
</table>

Virtual memory (disk)

Physical memory (DRAM)

Virtual address

Memory resident page table (DRAM)
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
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:](image1)

![Virtual Address Space for Process 2:](image2)

![Address translation](image3)

![Physical Address Space (DRAM):](image4)
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)
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
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)
VM Address Translation
VM Address Translation

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

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

• 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)

Physical address

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

Page table

Valid  Physical page number (PPN)

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

Page table address for process

Page table base register (PTBR)
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

PTEA: page table entry address
PTE: page table entry
PA: physical address
VA: virtual address
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

PTEA: page table entry address
PTE: page table entry
PA: physical address
VA: virtual 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.
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

- 2K allocated VM pages for code and data
- 6K unallocated VM pages
- 1023 unallocated pages
- 1 allocated VM page for the stack

32 bit addresses, 4KB pages, 4-byte PTEs
Why Two-level Page Table Reduces Memory Requirement?

• If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist.
• Only the level 1 table needs to be in main memory at all times.
• The level 2 page tables can be created and paged in and out by the VM system as they are needed.
Review of 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
Simple Memory System Example

• Addressing
  • 14-bit virtual addresses
  • 12-bit physical address
  • Page size = 64 bytes
Simple Memory System Page Table

Only show first 16 entries (out of 256)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple Memory System TLB

- 16 entries
- 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System Cache

• 16 lines, 4-byte block size
• Physically addressed
• Direct mapped

```
Idx  Tag  Valid  B0  B1  B2  B3
0    19   1      99  11  23  11
1    15   0      -   -   -   -
2    1B   1      00  02  04  08
3    36   0      -   -   -   -
4    32   1      43  6D  8F  09
5    0D   1      36  72  F0  1D
6    31   0      -   -   -   -
7    16   1      11  C2  DF  03
```

```
Idx  Tag  Valid  B0  B1  B2  B3
8    24   1      3A  00  51  89
9    2D   0      -   -   -   -
A    2D   1      93  15  DA  3B
B    0B   0      -   -   -   -
C    12   0      -   -   -   -
D    16   1      04  96  34  15
E    13   1      83  77  1B  D3
F    14   0      -   -   -   -
```
Address Translation Example #1

Virtual Address: \(0x03D4\)

Physical Address

![Diagram showing address translation process]
Address Translation Example #2

Virtual Address: \texttt{0x0B8F}

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\texttt{0x0B8F} \quad \texttt{0x2E} \quad \texttt{0x0B} \quad \texttt{N} \quad \texttt{Y} \quad \texttt{TBD}

Physical Address

<table>
<thead>
<tr>
<th>CT</th>
<th>Cl</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\texttt{CO} \quad \texttt{Cl} \quad \texttt{CT} \quad \texttt{Hit?} \quad \texttt{Byte:}
Address Translation Example #3

Virtual Address: \(0x0020\)

Virtual Address: \(0x0020\)

Physical Address

Physical Address
Intel Core i7 Memory System

Processor package
Core x4

- Registers
- Instruction fetch
- L1 d-cache 32 KB, 8-way
- L1 i-cache 32 KB, 8-way
- L2 unified cache 256 KB, 8-way
- L1 d-TLB 64 entries, 4-way
- L1 i-TLB 128 entries, 4-way
- L2 unified TLB 512 entries, 4-way
- QuickPath interconnect 4 links @ 25.6 GB/s each
- DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s
  32 GB/s total (shared by all cores)
- Main memory

To other cores
To I/O bridge
Review of 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)
  • $\text{TLBI}$: TLB index
  • $\text{TLBT}$: TLB tag
  • $\text{VPO}$: Virtual page offset
  • $\text{VPN}$: Virtual page number

• Components of the physical address (PA)
  • $\text{PPO}$: Physical page offset (same as $\text{VPO}$)
  • $\text{PPN}$: Physical page number
  • $\text{CO}$: Byte offset within cache line
  • $\text{CI}$: Cache index
  • $\text{CT}$: Cache tag
End-to-end Core i7 Address Translation

Virtual address (VA) → TLB

TLB hit → L1 TLB (16 sets, 4 entries/set)

L1 TLB hit → PTE

Page tables → Physical address (PA)

CPU

VPN VPO

TLBT TLBI

VPN1 VPN2 VPN3 VPN4

CR PTE PTE PTE PTE

TLB

miss

L1

hit

L1 miss

L2, L3, and main memory

32/64

Result

L1 d-cache (64 sets, 8 lines/set)

Physical address (PA)
Virtual Memory of a Linux Process

Different for each process:
- Process-specific data structs (ptables, task and mm structs, kernel stack)
- Kernel code and data

Identical for each process:
- Physical memory
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Kernel virtual memory

Process virtual memory

Process virtual memory:
- Different for each process
- Kernel virtual memory

Virtual memory:
- Kernel virtual memory
- Process virtual memory

%esp
brk
0x08048000 (32)
0x00400000 (64)
Linux Organizes VM as Collection of “Areas”

- **pgd:**
  - Page global directory address
  - Points to L1 page table

- **vm_prot:**
  - Read/write permissions for this area

- **vm_flags**
  - Pages *shared* with other processes or *private* to this process
Linux Page Fault Handling

Segmentation fault: accessing a non-existing page

Normal page fault

Protection exception: e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)
Memory Mapping

• VM areas initialized by associating them with disk objects.
  • Process is known as memory mapping.

• Area can be backed by (i.e., get its initial values from):
  • Regular file on disk (e.g., an executable object file)
    • Initial page bytes come from a section of a file
  • Anonymous file (e.g., nothing)
    • First fault will allocate a physical page full of 0's (demand-zero page)
    • Once the page is written to (dirtied), it is like any other page

• Dirty pages are copied back and forth between memory and a special swap file.
Demand paging

• **Key point**: no virtual pages are copied into physical memory until they are referenced!
  • Known as *demand paging*

• Crucial for time and space efficiency
Sharing Revisited: Shared Objects

• Process 1 maps the shared object.
Sharing Revisited: Shared Objects

- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.
Sharing Revisited: Private Copy-on-write (COW) Objects

- Two processes mapping a private copy-on-write (COW) object.
- Area flagged as private copy-on-write.
- PTEs in private areas are flagged as read-only.
Sharing Revisited: Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

---

**Diagram Description**

- **Process 1**: Virtual memory
- **Physical Memory**: Copy-on-write object
- **Process 2**: Virtual memory
- **Copy-on-write page**: Write to private copy-on-write page