Virtual Memory

Fall 2012

Instructors: Aykut and Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O’Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.
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
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
Recall: Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- Note: system provides private address spaces to each “process”
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others
Recall: Simple Addressing Modes

- **Normal** (R) \(\text{Mem}[\text{Reg}[R]]\)
  - Register R specifies memory address

  \[
  \text{movl} \ (\%\text{ecx}),\%\text{eax}
  \]

- **Displacement** D(R) \(\text{Mem}[\text{Reg}[R]+D]\)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \[
  \text{movl} \ 8(\%\text{ebp}),\%\text{edx}
  \]
Lets think on this: physical memory?

- How does everything fit?
  - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
  - 64-bit addresses: ~16,000,000,000,000,000,000,000 (16 quintillion) bytes

- How to decide which memory to use in your program?
  - What about after a fork()?

- What if another process stores data into your memory?
  - How could you debug your program?
So, we add a level of indirection

- **One simple trick solves all three problems**
  - Each process gets its own private image of memory
    - appears to be a full-sized private memory range
  - This fixes “how to choose” and “others shouldn’t mess w/yours”
    - surprisingly, it also fixes “making everything fit”
  - Implementation: translate addresses transparently
    - add a mapping function
      - to map private addresses to physical addresses
    - do the mapping on every load or store

- **This mapping trick is the heart of virtual memory**
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 datum can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses
A System Using Physical Addressing

- Used in some “simple” systems, like embedded microcontrollers in 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
Why Virtual Memory?

(1) VM allows efficient use of limited main memory (RAM)
   ▪ Use RAM as a cache for the parts of a virtual address space
     ▪ some non-cached parts stored on disk
     ▪ some (unallocated) non-cached parts stored nowhere
   ▪ Keep only active areas of virtual address space in memory
     ▪ transfer data back and forth as needed

(2) VM simplifies memory management for programmers
   ▪ Each process gets a full, private linear address space

(3) VM isolates address spaces
   ▪ One process can’t interfere with another’s memory
     ▪ because they operate in different address spaces
   ▪ User process cannot access privileged information
     ▪ different sections of address spaces have different permissions
Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
(1) 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](image)
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
Enabling data structure: Page Table

- 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
Page Hit

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

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

- **Virtual address**
- **Physical page number or disk address**
- **Memory resident page table (DRAM)**
- **Virtual memory (disk)**
- **Physical memory (DRAM)**
Page Fault

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

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

- **Physical memory (DRAM)**
  - VP 0
  - VP 1
  - VP 2
  - VP 3
  - VP 4
  - VP 7

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

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Memory resident page table (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0</td>
<td>Valid: 0 null 1 1 0 1 0 1</td>
</tr>
<tr>
<td>PTE 7</td>
<td>Valid: 0 null 0 1</td>
</tr>
</tbody>
</table>

Physical page number or disk address

Physical memory (DRAM)

Virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
- VP 4
- VP 7
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!

![Diagram of memory mapping and page fault handling](Image)
Allocating Pages

- Operating system allocates a new page of virtual memory, for example, as a result of calling `malloc`.
- In the example, VP 5 is allocated by creating room on disk and updating PTE 5 to point to the newly created page on disk.

```
        Virtual address
        +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
        |     |     |     |     |     |     |     |     |     |     |     |     |
        | PTE 0 | PTE 1 | PTE 2 | PTE 3 | PTE 4 | PTE 5 | PTE 6 | PTE 7 | PTE 8 | PTE 9 | PTE 10 | PTE 11 |
        +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
        |     |     |     |     |     |     |     |     |     |     |     |     |
        | Valid | Valid | Valid | Valid | Valid | Valid | Valid | Valid | Valid | Valid | Valid | Valid |
        +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
        |     |     |     |     |     |     |     |     |     |     |     |     |
        | 0   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
        +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
        |     |     |     |     |     |     |     |     |     |     |     |     |
        | null |     |     |     |     |     |     |     |     |     |     |     |
        +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

Physical page number or disk address

Physical memory (DRAM)

Virtual memory (disk)

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 moved (copied) in and out continuously
Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
(2) 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:

0       Address translation       0
VP 1    PP 2
VP 2    PP 6
...    PP 8
N-1

Virtual Address Space for Process 2:

0
VP 1
VP 2
...
N-1

Physical Address Space (DRAM)

(e.g., read-only library code)
```
Simplifying allocation and sharing

- 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 multiple 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
Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
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)
Virtual memory review

- **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
Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
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)**
  - **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
  - **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)

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

Page table address for process

m-1  p  p-1  0

n-1  p  p-1  0

0  p-1  p  m-1

p-1  p  0

0  p-1  p  m-1

33
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

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

CPU Chip

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Question #1

- Are the PTEs cached like other memory accesses?

- Yes (and no: see next question)
Page tables in memory, like other data

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Question #2

- Isn’t it slow to have to go to memory twice every time?

- Yes, it would be... so, real MMUs don’t
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, dedicated, super-fast hardware cache of PTEs in MMU
  - Contains complete page table entries for small number of pages
TLB Hit

A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Question #3

- Isn’t the page table huge? How can it be stored in RAM?

- Yes, it would be... so, real page tables aren’t simple arrays
Multi-Level Page Tables

- **Suppose:**
  - 4KB ($2^{12}$) page size, 64-bit address space, 8-byte PTE

- **Problem:**
  - Would need a 32,000 TB page table!
    - $2^{64} \times 2^{12} \times 2^3 = 2^{55}$ 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 0
... PTE 0
... PTE 0
... PTE 0
... PTE 0
... PTE 0
... PTE 0
VP 0
... VP 1023
... VP 1024
... VP 2047
Gap

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
Translating with a k-level Page Table

![Diagram showing a k-level page table translation process.](image)

- **VPNs**: Virtual Page Numbers
- **VPO**: Virtual Page Offset
- **PPNs**: Physical Page Numbers
- **PPO**: Physical Page Offset
- **Level 1, Level 2, ..., Level k**: Page tables at different levels of the translation hierarchy
- **Virtual Address**: Input address
- **Physical Address**: Output address

The diagram illustrates how a virtual address is translated into a physical address through a sequence of page table lookups, starting from the VPO and ending with the PPO.
Question #4

- Shouldn’t fork() be really slow, since the child needs a copy of the parent’s address space?

- Yes, it would be... so, fork() doesn’t really work that way
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!
The `fork` Function Revisited

- `fork` provides private address space for each process

- To create virtual address for new process
  - Create exact copies of parent page tables
  - Flag each page in both processes (parent and child) as read-only
  - Flag writeable areas in both processes as private COW

- On return, each process has exact copy of virtual memory

- Subsequent writes create new physical pages using COW mechanism

- Perfect approach for common case of `fork()` followed by `exec()`
  - Why?
Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
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)**
  - **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
  - **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

![Virtual and Physical Address Diagram]

- **Virtual Page Number (VPN)**
- **Virtual Page Offset (VPO)**
- **Physical Page Number (PPN)**
- **Physical Page Offset (PPO)
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

![Cache Diagram]

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

VPN 0xF  TLBI 0x3  TLBT 0x03  TLB Hit? Y  Page Fault? N  PPN: 0xD

Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

Virtual Address: 0x0B8F

VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? N Page Fault? Y PPN: TBD

Physical Address

CO ____ Cl____ CT ____ Hit? ____ Byte: ____
Address Translation Example #3

Virtual Address: 0x00020

Physical Address

Byte: Mem
Diverse workloads and a wide range of hardware configurations compound the complexity of an operating system’s memory management policies.

Virtual Memory Management in the VAX/VMS Operating System

Henry M. Levy and Peter H. Lipman, Digital Equipment Corporation

Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
Intel Core i7 Memory System

Processor package

Core x4

- **Registers**
- **Instruction fetch**
- **MMU (addr translation)**

- **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

- **L3 unified cache**
  - 8 MB, 16-way
  - (shared by all cores)

- **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)**
  - 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
## End-to-end Core i7 Address Translation

![Diagram of address translation process]

1. **CPU** takes a virtual address (VA) as input.
2. **VPN** and **VPO** are used to determine the corresponding **TLB** entry. If the **TLB** entry is not found (miss), the process continues to the next level. If found (hit), the translation is completed.
3. **TLBT** and **TLBI** are used to determine if the **TLB** entry is a hit or a miss. If it is a hit, the translated address is returned. If a miss, it goes to the next level.
4. **L1 TLB** (16 sets, 4 entries/set) is queried. If hit, the address is translated. If miss, it goes to the next level.
5. **Page tables** consist of **PTE** entries. If a page table entry is found, the address is translated. If not found, it goes to the next level.
6. **Physical address (PA)** is calculated using the **CR3** and **PTE** entries.
7. The final **Result** is generated and sent back to the **CPU**.

**Key Points**:
- **L1 d-cache**: 32/64 bit access, 16-way set-associative, 8 lines per set.
- **L2, L3, and main memory**: Accesses are direct with no cache level involved.
- **Physical address (PA)** is calculated using the **CR3** and **PTE** entries.
Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page table physical base address</td>
<td>Unused</td>
<td>G</td>
<td>PS</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available for OS (page table location on disk) P=0

**Each entry references a 4K child page table**

**P:** Child page table present in physical memory (1) or not (0).

**R/W:** Read-only or read-write access access permission for all reachable pages.

**U/S:** user or supervisor (kernel) mode access permission for all reachable pages.

**WT:** Write-through or write-back cache policy for the child page table.

**CD:** Caching disabled or enabled for the child page table.

**A:** Reference bit (set by MMU on reads and writes, cleared by software).

**PS:** Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

**G:** Global page (don’t evict from TLB on task switch)

**Page table physical base address:** 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page physical base address</td>
<td>Unused</td>
<td>G</td>
<td>D</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available for OS (page location on disk)  P=0

Each entry references a 4K child page

**P:** Child page is present in memory (1) or not (0)

**R/W:** Read-only or read-write access permission for child page

**U/S:** User or supervisor mode access

**WT:** Write-through or write-back cache policy for this page

**CD:** Cache disabled (1) or enabled (0)

**A:** Reference bit (set by MMU on reads and writes, cleared by software)

**D:** Dirty bit (set by MMU on writes, cleared by software)

**G:** Global page (don’t evict from TLB on task switch)

**Page physical base address:** 40 most significant bits of physical page address (forces pages to be 4KB aligned)
Core i7 Page Table Translation

Virtual address

Offset into physical and virtual page

Physical address

CR3 Physical address of L1 PT

VPN 1 | VPN 2 | VPN 3 | VPN 4 | VPO

L1 PT Page global directory

L2 PT Page upper directory

L3 PT Page middle directory

L4 PT Page table

L1 PTE  L2 PTE  L3 PTE  L4 PTE

512 GB region per entry

1 GB region per entry

2 MB region per entry

4 KB region per entry

VPN 1  VPN 2  VPN 3  VPN 4  VPO

PPN  PPO
Cute Trick for Speeding Up L1 Access

- **Observation**
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Generally we hit in TLB, so PPN bits (CT bits) available next
  - “Virtually indexed, physically tagged”
  - Cache carefully sized to make this possible
Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
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
User-Level Memory Mapping

void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)

- Map len bytes starting at offset offset of the file specified by file description fd, preferably at address start
  - start: may be 0 for “pick an address”
  - prot: PROT_READ, PROT_WRITE, ...
  - flags: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be start)
User-Level Memory Mapping

```c
void *mmap(void *start, int len,
        int prot, int flags, int fd, int offset)
```

- `len` bytes
- `start` (or address chosen by kernel)
- `offset` (bytes)

Disk file specified by file descriptor `fd`

Process virtual memory
#include "csapp.h"

/*
 * mmapcopy - uses mmap to copy
 * file fd to stdout
 */

void mmapcopy(int fd, int size)
{
    /* Ptr to mem-mapped VM area */
    char *bufp;

    bufp = Mmap(NULL, size,
                 PROT_READ,
                 MAP_PRIVATE, fd, 0);
    Write(1, bufp, size);
    return;
}

/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmdline arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }

    /* Copy the input arg to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
Virtual Memory of a Linux Process

- **Process-specific data structs** (ptables, task and mm structs, kernel stack)
- **Physical memory**
- **Kernel code and data**
- **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**

Different for each process

Identical for each process

%esp

brk

Different for each process

Identical for each process

---

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

Process virtual memory

- shared libraries
- data
- text

vm_area_struct
- vm_end
- vm_start
- vm_prot
- vm_flags

1. **Segmentation fault:**
   accessing a non-existing page

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

3. **Normal page fault**
The `execve` Function Revisited

- To load and run a new program a.out in the current process using `execve`:
  - Free `vm_area_struct`'s and page tables for old areas
  - Create `vm_area_struct`'s and page tables for new areas
    - Programs and initialized data backed by object files.
    - `.bss` and stack backed by anonymous files.
  - Set PC to entry point in `.text`
    - Linux will fault in code and data pages as needed.

```
To load and run a new program a.out in the current process using execve:

- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files.
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed.
```