Virtual Memory: Systems

Instructors:
Alvin R. Lebeck

Slides provided by Randy Bryant and Dave O’Hallaron

Administrivia

- Work on Project (Due Wed Dec 7 no late submissions)
- HW #6, Due next Monday
- #7 up, Due Monday Dec 5 (no late submissions)
- After VM, one main topic: storage & I/O
- One class on advanced topics
Today

- 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

Review: Computer Memory

- Memory is a large linear array of bytes.
- 32-bit or 64-bit addresses
Review: Memory Hierarchy 101

Very fast <1ns clock
Multiple Instructions per cycle

SRAM, Fast, Small
Expensive
~KB to MB

DRAM, Slow, Big, Cheap
(called physical or main)
~1-4 GB

=> Cost Effective Memory System (Price/Performance)
You should know how to draw block diagram of cache
and how to compute miss ratios, memory stall cycles, CPU time

Any problems with this picture?

Review: A Simple Program’s Memory Layout

- What are the possible addresses generated by the program?
- How big is our DRAM?
- Is there more than one program running?
- If so, how do we allocate memory to each?
Extending the Memory Hierarchy

Very fast 1ns clock
Multiple Instructions per cycle

HW manages movement

SRAM, Fast, Small
Expensive

DRAM, Slow, Big, Cheap
(called physical or main memory)

Magnetic, Really Slow,
Really Big, Really Cheap

Virtual Memory: Motivation

- **Process** = Address Space + thread of control (PC)
- **Address space** = Physical
  - programmer controls movement from disk
  - protection?
  - relocation?
- **Linear Address space**
  - larger than physical address space
  - 32, 64 bits v.s. 28-bit physical (256MB)
- **Want Automatic management**
Virtual Memory

Process = virtual address space + thread of control

Translation
- VA -> PA
- What physical address does virtual address A map to
- Is VA in physical memory?

Protection (access control)
- Do you have permission to access it?

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: Page Table**

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

Memory resident page table (DRAM)

Physical memory (DRAM)

Virtual memory (disk)

Physical page number or disk address

Page Hit

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

**Diagram: Page Hit**

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

Memory resident page table (DRAM)

Physical memory (DRAM)

Virtual memory (disk)
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)

Virtual address

Valid
PTE 0
0 null
1
1
0
1
0
0
PTE 7
1

Physical page number or disk address

Physical memory (DRAM)

VP 1
VP 2
VP 7
VP 4

Physical memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Memory resident page table (DRAM)

Virtual memory

null
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

- Review: VM as a tool for caching
- **Review: VM as a tool for memory management**
- Review: VM as a tool for memory protection
- Review: Address translation
- Core i7 case study
- Simple memory system example
- Case study: Core i7/Linux memory system

**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 Diagram]

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

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

<table>
<thead>
<tr>
<th>Address translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>M-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Address Space (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 2</td>
</tr>
<tr>
<td>PP 6</td>
</tr>
<tr>
<td>PP 8</td>
</tr>
<tr>
<td>(e.g., read-only library code)</td>
</tr>
<tr>
<td>PP 2</td>
</tr>
<tr>
<td>PP 6</td>
</tr>
<tr>
<td>PP 8</td>
</tr>
<tr>
<td>(e.g., read-only library code)</td>
</tr>
<tr>
<td>PP 2</td>
</tr>
<tr>
<td>PP 6</td>
</tr>
<tr>
<td>PP 8</td>
</tr>
<tr>
<td>(e.g., read-only library code)</td>
</tr>
<tr>
<td>PP 2</td>
</tr>
<tr>
<td>PP 6</td>
</tr>
<tr>
<td>PP 8</td>
</tr>
<tr>
<td>(e.g., read-only library code)</td>
</tr>
</tbody>
</table>
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
Today

- Review: VM as a tool for caching
- Review: VM as a tool for memory management
- Review: VM as a tool for memory protection
- Address translation
  - TLBs
  - VM & caches
- Core i7 case study
- Simple memory system example
- Case study: Core i7/Linux 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)

```
Process i:

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

Process j:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
<td></td>
</tr>
</tbody>
</table>
```

---

Physical Address Space

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
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)

Page table

Valid  Physical page number (PPN)

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

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

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
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

TLB Hit

A TLB hit eliminates a memory access
**TLB Miss**

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
- 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

Level 2 page tables
- PTE 0
- ... PTE 1023
- PTE 0
- ... PTE 1023
- 1023 null PTEs
- PTE 1023

Virtual memory
- 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
- 1023 unallocated pages

32 bit addresses, 4KB pages, 4-byte PTEs

Summary, thus far

- **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
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

![Virtual and Physical Address Diagram](attachment:image.png)
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

Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

VPN 0x0F  TLBI 0x3  TLB Hit? Y  Page Fault? N  PPN: 0x0D  Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2E</td>
<td>TBD</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>CT</th>
<th>CI</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Address Translation Example #3

Virtual Address: 0x0020

<table>
<thead>
<tr>
<th>TLBT</th>
<th>TLBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x28</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>CT</th>
<th>CI</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Byte: Mem
Today

- Review: VM as a tool for caching
- Review: VM as a tool for memory management
- Review: VM as a tool for memory protection
- Review: Address translation
- Case study: Core i7/Linux memory system
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
### Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>Page physical base address</th>
<th>Unused</th>
<th>G</th>
<th>PS</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available for OS (page table location on disk)  

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 (1) or enabled (0).  
- **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>XD</th>
<th>Unused</th>
<th>Page physical base address</th>
<th>Unused</th>
<th>G</th>
<th>D</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available for OS (page location on disk)  

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 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

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
Virtual Memory of a Linux Process

- **Kernel virtual memory**
  - Process-specific data structs (ptables, task and mm structs, kernel stack)
  - Physical memory
  - Kernel code and data
- **Process virtual memory**
  - User stack (esp)
  - Memory mapped region for shared libraries (brk)
    - Runtime heap (malloc)
    - Uninitialized data (.bss)
    - Initialized data (.data)
    - Program text (.text)
  - Program text (.text)
  - IniOalized data (.data)
  - UniniOalized data (.bss)

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

- `task_struct` and `mm_struct`
- `vm_area_struct`
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)

Virtualization (VMware)

- **Boot and run multiple operating systems**
  - Windows, MacOS, Linux all at once
  - Not dual booting!
- **Requires special support for efficiency**
  - New privilege mode for hypervisor