Virtual Memory

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Have you ever thought …

• Why the text segment (Do you remember heap, stack, text, and data?) of any process starts at the same address?
• Why don’t you run out of memory even if the sum of memory requirement of each program you are running at the same time exceeds the amount of memory you have in your machine?
• The address is 64-bit, in 64-bit machines. It accesses memory from 0 to \(2^{64} - 1\) which is way more than the amount of memory you have on your machine?

How come?
Virtual Addressing

- Used in all modern servers, desktops, and laptops
Why Virtual Memory (VM)?

- **Simplified memory management**
  - Each process gets an *exclusive* linear address space

- **Process Isolation**
  - Different processes have different virtual address spaces
  - One process can't interfere with another's memory

- **Uses main memory efficiently**
  - Parts of a process not needed are sent to the disk
Address translation

- Key idea of VM: each process has its own virtual address space

Strawman view of addr translation

<table>
<thead>
<tr>
<th>VA</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>..</td>
</tr>
<tr>
<td>0x08</td>
<td>0x98</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>invalid</td>
</tr>
</tbody>
</table>

Granularity of mapping?

- **Byte-level**: Map each byte in VA to a byte in PA → too expensive
- **Page-level**: Map each consecutive $2^p$-byte address range in VA to a $2^p$-byte address range in PA

Valid address:
- [0, $2^{64}$) for 64-bit machine

Valid address:
- [0, $2^m$) depending on how much memory your machine has
Address Translation with a Page Table

Virtual address

<table>
<thead>
<tr>
<th>n-1</th>
<th>p</th>
<th>p-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual page number (VPN)</td>
<td>Virtual page offset (VPO)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page table

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number (PPN)</th>
</tr>
</thead>
</table>

PTE (page table entry)

<table>
<thead>
<tr>
<th>m-1</th>
<th>p</th>
<th>p-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number (PPN)</td>
<td>Physical page offset (PPO)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page table base register (PTBR)

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

Page table address for process

How kernel tells h/w where to find the page table
Page Table Base Register (PTBR)

• The operating system maintains information about each process in a process control block.
• The page table base address for the process is stored there.
• The operating system loads this address into the PTBR whenever a process is scheduled for execution.
• Only the kernel can access PTBR
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
PA: Physical Address
PTE: Page Table Entry
PTEA: PTE 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 in kernel**
   
If VA is invalid, then kill process (SIGSEGV)
   
If VA has been paged out to disk, then swaps in faulted page, update page table, resume faulted process
There are two challenges

• **Speed**: VA to PA translation means we need to access the memory twice for each CPU memory request!

• **Size**: page table can be huge. And we have a page table per process.
Tackling the Speed Problem
Speeding up Translation with a TLB

• VA->PA translation can be expensive
  – One additional memory reference for every normal memory reference!
  – How about this: Page table entries (PTEs) are cached in L1 like others? .... Not very good idea.
    • PTEs may be evicted by other data references
    • PTE hit still requires a small L1 delay

• Solution: Translation Lookaside Buffer (TLB)
  – Small hardware cache inside the MMU (i.e. on chip)
  – 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?
Tackling the size (i.e. memory requirement of page tables) problem
Reduce Page Table Size

• 4KB-page, 48-bit address space, 8-byte PTE
• Size of page table needed?
  – $2^{48-12} \times 2^3 = 2^{39} = 512 \text{ GB}$
• Wasteful: most PTEs are invalid...
• Solution: multi-level page table
  – Example: 2-level page table
    • Level 1 table: each PTE points to a page table
    • Level 2 table: each PTE points to a page
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 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8

1023 null PTEs

PTE 1023

Virtual memory

- VP 0
- ... VP 1023
- VP 1024
- ... VP 2047

Gap

1023 unallocated pages

VP 9215

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.
We can now say that VM is useful for:

- Memory management
- Simplified linking and loading
- Memory protection
- Caching
Memory management and protection

- Each process has an **exclusive VA space**
  - One process cannot overwrite another one's memory!
- **Sharing** among processes
  - Map different virtual pages to the same physical page

![Diagram of virtual and physical address spaces for two processes.](image)
Simplified Linking and Loading

• Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

• Loading
  - `execve()` causes kernel to allocate virtual pages
  - Kernel copies `.text` and `.data` sections, page by page, from disk to memory
Memory Protection

- How to **protect shared pages from corruption**?
  - E.g. bad process overwrites shared kernel code/data, shared libc code etc.
- Extend PTEs with permission bits

### 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>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</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>PP 2</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>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

**SUP**: whether processes must be running in kernel (supervisor) mode to access the page.
VM as a Tool for Caching

• Not all processes’ valid VA pages fit in physical memory

• Key idea: treat DRAM-resident pages as a cache of on-disk pages
VM for Caching

• General mechanism:
  – On page fault, load corresponding on-disk page to memory, evict a previously memory-resident to disk, set appropriate PTE entry

• Which entity should be performing this task?
  – User-level process? OS? Hardware?

• VM Caching policy (more sophisticated than CPU cache)
  – Fully associative: any VP can be mapped to any PP
  – Write-back
Page Hit

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

Kernel figures out where to find the corresponding on-disk page
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!
Why should VM caching work?

• 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 (working set sizes > main memory size)
  – *Thrashing*: Performance meltdown where pages are swapped (copied) in and out continuously
Toy Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes
# Toy Memory System Page Table

<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>

1-level page table: How many PTEs?
# Address Translation Example

**Virtual Address:** 0x0354

<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>

What’s the corresponding PPN? Physical address?
Integrating Caches and VM

• Shall we access caches with virtual address? or physical address?

• With virtual address:
  – good: cache accessed as soon as possible
  – bad: aliasing (i.e. two different virtual addresses can map to the same cache block and vice versa) → more complicated cache needed to handle this.

• With physical address:
  – good: no aliasing
  – bad: must wait for address translation (VA → PA) before cache can be accessed.
Case study: Core i7/Linux memory system (Nehalem microarchitecture)
Intel Core i7 Memory System

Processor chip package

One core (4 total)

Registers
L1 d-cache 32 KB
L1 i-cache 32 KB
L2 unified cache 256 KB
L3 unified cache 8 MB, (shared by all cores)

Instruction fetch

MMU (addr translation)
L1 d-TLB 64 entries
L1 i-TLB 128 entries
L2 unified TLB 512 entries

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
i7 Memory Hierarchy

• 48-bit virtual address
• 52-bit physical address
• TLBs are virtually addressed
• Caches are physically addressed
• Page size can be configured at start-up time as either 4KB or 4MB
  – Linux uses 4KB
• i7 uses 4-level page table hierarchy
• Each process has its own private page table hierarchy
Core i7 Page Table Translation

Virtual address

Offset into physical and virtual page

Physical address

Physical address of page

CR3
Physical address of L1 PT

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

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

PPN

PPO
Core i7 Page Table Entry (level-4)

- **63-62**: Unused
- **52-51**: PPN
- **12-11**: Unused
- **9-8**: D (Dirty bit)
- **7-6**: A
- **5-4**: U/S (User or supervisor mode access)
- **3-2**: R/W (Read-only or read-write permission)
- **1-0**: P (Page in memory or not)

Dirty bit (set by MMU on writes, cleared by OS)
Reference bit (set by MMU on reads and writes, cleared by OS)
End-to-end Core i7 Address Translation

Virtual address (VA)

CPU

VPN
VPO

36 12

VPN1 VPN2 VPN3 VPN4

TLB miss

L1 TLB

L1 cache

L2, L3, and main memory

L1 hit

L1 miss

TLB hit

VPN
VPO

32/64

Result

Page tables

Physical address (PA)

CR3

PTE

Page tables

PPP

PPN

40

12

9 9 9 9

VPN1 VPN2 VPN3 VPN4
Conclusions

• In this lecture we have seen VM in action.
• It is important to know how the following pieces interact:
  – CPU, MMU, DRAM, Cache, Kernel
• 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 moving memory pages between disk and memory