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 machines.

**CPU Chip**

CPU

Virtual address (VA) 4100

MMU: Memory Management Unit

Physical address (PA) 4

Main memory

0: 
1: 
2: 
3: 
4: (shaded)
5: 
6: 
7: 
8: 
...
M-1:

Data word
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

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
Address Translation w/ a Page Table

Virtual address

Page table base register (PTBR)

Page table address for process

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

How kernel tells hardware where to find the page table

Physical address

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

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

PTE (page table entry)
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
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
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 showing virtual memory spaces for two processes and their corresponding physical pages.]
Simplified Linking and Loading

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

• **Loading**
  - Starting a program execution 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>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 6</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 9</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
<td></td>
</tr>
</tbody>
</table>

**SUP**: whether processes must be running in kernel (supervisor) mode to access the page.
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>

<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>

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
- Instruction fetch
- L1 d-cache 32 KB
- L1 i-cache 32 KB
- L2 unified cache 256 KB
- 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

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

VPN 1

VPN 2

VPN 3

VPN 4

VPO

VPN 1

VPN 2

VPN 3

VPN 4

VPO

512 GB region per entry

1 GB region per entry

2 MB region per entry

4 KB region per entry

40

40

40

40

Physical address of page

40

12

Physical address

40

12

Physical address

40

12

Physical address

40

12

Physical address

40

12

Physical address
Core i7 Page Table Entry (level-4)

- **Unused**
- **PPN**
- **Unused**
- **D**
- **A**
- **U/S**
- **R/W**
- **P**

- Dirty bit (set by MMU on writes, cleared by OS)
- Reference bit (set by MMU on reads and writes, cleared by OS)
- User or supervisor mode access
- Read-only or read-write permission
- Page in memory or not
End-to-end Core i7 Address Translation

- Virtual address (VA)
  - VPN
  - VPO
  - TLB miss

- L1 TLB
  - VPN1
  - VPN2
  - VPN3
  - VPN4
  - TLB hit
  - PTE
  - Page tables

- L1 cache
  - L1 hit
  - L1 miss

- Physical address (PA)
  - PPN
  - PPO
  - CR3
  - 32/64
  - Result
  - L2, L3, and main memory
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