CSC2/452 Computer Organization
Virtual Memory

Sreepathi Pai

URCS

November 2, 2022
Outline

Administrivia

Recap

Memory Virtualization

x86-64 Implementation of Virtual Memory

OS implementation of virtual memory
Outline

Administrivia

Recap

Memory Virtualization

x86-64 Implementation of Virtual Memory

OS implementation of virtual memory
Administrivia

- Assignment #3 out today
  - Due Nov 13, 2022
- Homework #6 out today
  - Due Wed Nov 9
  - I have printed copies with me today
Outline

Administrivia

Recap

Memory Virtualization

x86-64 Implementation of Virtual Memory

OS implementation of virtual memory
The OS and virtualization

- The operating system and CPU cooperate to perform virtualization
- CPU virtualization
  - Time sharing
- Memory virtualization
  - Today
Outline

Administrivia

Recap

Memory Virtualization

x86-64 Implementation of Virtual Memory

OS implementation of virtual memory
The High-Level Problem

How do you make every program believe it has access to the full RAM?
Time Sharing

- Program A starts executing with full access to memory
- Timer interrupt
- All memory for Program A is copied to “swap area”
  - swap area could be hard disk, for example
- All memory for Program B is loaded from “swap area”
- Program B starts executing
- Repeat
The Problem with Time Sharing

- My laptop has 8GB of RAM
  - Worst case save and restore data size
- My HDD writes about 500MB/s
- 16s to save full contents of memory
- 16s to load full contents of memory
- 32s to switch between programs
## Space Sharing

<table>
<thead>
<tr>
<th>Memory</th>
<th>Program A</th>
<th>Program B</th>
<th>Empty</th>
</tr>
</thead>
</table>

- Divide memory into portions
- Each program gets some portion of memory
The Problems with Space Sharing: #1

- How do we size each portion?
  - Fixed-size allocations waste space
  - Known as the “trapped-capacity” problem
Problem #2: Contiguous address space requirements

- Can’t change size of allocations as programs are running
  - At least not easily
  - Need contiguous address spaces
- Think of an array that is bigger than each portion, but smaller than two portions combined
Problem #3: Can’t move allocations

- Can’t move allocations
  - Pointers in programs would need to be updated
Adding a Translation Layer

- Programs need to see one contiguous address space
  - We will call this the virtual address space
- We will translate from this virtual address space to actual physical address space
- Programs use virtual addresses, only the OS sees physical addresses
Virtual and Physical Addresses

- A virtual address and a physical address are “physically” indistinguishable
  - Both are 64-bit
  - However, virtual addresses span the whole 64-bit range
  - Physical addresses only span the actual amount of physical memory present
- All addresses used in programs are virtual
  - Except in very special cases when values of virtual addresses and its translated physical address are the same
  - Usually, when doing I/O with devices that don’t understand virtual addresses
  - (Devices like this are increasingly uncommon)
Translation Granularity

- We could translate any virtual address to any physical address
  - i.e. at byte level granularity
- But, it is more efficient to translate larger regions of memory
- Memory is divided into non-overlapping contiguous regions called pages
  - Most common page size is 4096 bytes (or 4KB)
  - But modern systems support large (or huge) pages (2MB or more)
  - The new M1 Macs have a 16384 byte page size
A load or store instruction uses virtual addresses

The memory management unit (MMU) translates this virtual address to a physical address

Nearly everything “downstream” of the MMU sees physical addresses
Who maintains the translations?

- Although the CPU performs the translations, they are actually set up by the OS
- Page translations can change
  - Virtual address remains the same
  - Physical address changes
- This allows:
  - Allocation sizes to shrink and grow at page size granularity
  - Physical addresses can be non-contiguous
Swapping Pages Out

- The OS can mark virtual addresses as “not present”
  - The pages corresponding to these virtual addresses are not “mapped in”
  - Their contents may be on disk
  - No physical addresses are assigned to these virtual addresses
- Accessing these “swapped out” pages causes a page fault
  - CPU “suspending” processing of load/store instruction that caused fault
  - MMU notifies OS
Swapping Pages Back In

- When the OS receives a page fault notification, it can:
  - identify a page in physical memory
  - create a new mapping from the faulting virtual address to this page
  - load the contents of the newly mapped page from disk (if it was swapped out)
  - tell MMU that a new mapping has been set up
- CPU can then resume processing of load/store instruction
Summary

- Virtual memory uses a virtual address space
  - One, contiguous, linear address space
- Addresses are in the virtual address space are translated to physical addresses at page granularity
- Translations are setup by OS
- CPU MMU performs the translation on every load/store
- Virtual addresses can be marked as not present
  - Allows system to support allocating more physical memory than actually present!
  - CPU notifies OS whenever these addresses are accessed
- Programs do not notice these translations (except as loss in performance)
Outline

Administrivia

Recap

Memory Virtualization

x86-64 Implementation of Virtual Memory

OS implementation of virtual memory
Pages are 4KB (4096 bytes) in size
▶ How many bits?
▶ Also supports 2MB and 1GB pages, but we will not discuss these

Uses a structure called a page table to maintain translations
▶ Note, current implementations only use 48-bit to 52-bit virtual addresses
▶ How many entries in page table?
x86-64 Page Table Design

▶ 12 bits for offset within page (4096 bytes)
▶ 36 bits remaining (if using 48-bit virtual addresses)
  ▶ 16 bits not used in current x86-64 implementations
▶ Page table will contain $2^{36}$ entries
  ▶ Each program will require $8 \times 2^{36}$ bytes for its page table
  ▶ How much is this?
Space requirements for the page table

512GB
- Ideally, we only need to store as many translations as there are physical pages
  - e.g., if 8GB physical RAM, then 2097152 pages, so 16MB for page table entries
  - Called an *inverted page table* design
- Not used by x86-64
Hierarchical Page Tables

▶ Instead of a single page table, multi-level page tables are used
▶ On the x86-64, each level contains 512 entries
  ▶ How many bits required to index into each level?
  ▶ How many levels (given we have 36 bits)?
▶ Each entry is 64-bits wide
  ▶ Total size of each level?
▶ NOTE: x86-64 supports 52-bits in physical addresses
Translation: Goal
Translation: First level

Linear Address

47 39 38 30 29 21 20 12 11 0

PML4 Directory Ptr Directory Table Offset

4-KByte Page

Physical Addr

40

CR3

9

PML4E

30
Translation: Second level

Linear Address

47 39 38 30 29 21 20 12 11 0

PML4  Directory Ptr  Directory  Table  Offset

Page-Directory-Pointer Table

PDPTE

PML4E

CR3

4-KByte Page

Physical Addr
Translation: Third level

<table>
<thead>
<tr>
<th>47</th>
<th>39</th>
<th>38</th>
<th>30</th>
<th>29</th>
<th>21</th>
<th>20</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PML4</td>
<td>Directory Ptr</td>
<td>Directory</td>
<td>Table</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`PDE with PS=0`  
`Directory Ptr`  
`Linear Address`  
`4-KByte Page`  
`Physical Addr`  

- **CR3**: Page-Directory Pointer Table
- **PDPTE**: PDE with PS=0
- **PML4**: Page-Directory
- **PML4E**: Physical Addr
- **Physical Addr**: 4-KByte Page

Diagram:

- CR3
- PML4
- Directory
- PDE with PS=0
- Page-Directory
- 4-KByte Page
- Physical Addr
Translation: Fourth level

```
PML4  Directory Ptr  Directory  Table  Offset
39  38  30  29  21  20  12  11  0

47  9

Page-Directory-Pointer Table

PDPTE

Page-Directory

PDE with PS=0

Page Table

PTE

4-KByte Page

Physical Addr
```

40  9

Page-Directory-Pointer Table

PDPTE

Page-Directory

PDE with PS=0

Page Table

PTE

4-KByte Page

Physical Addr
Space requirements for multi-level page tables

- Each level contains 512 8-byte entries containing physical addresses
  - 4096 bytes
- A minimal program could get away with 4096*4 bytes for the page tables
  - No need for 512GB or even 16MB
- Note some of these levels can be “paged out”
  - i.e. each entry in these tables contains a present bit
Translation Overheads

- Translating one memory address requires reading 4 other addresses!
  - This is called a “page table walk”, performed by the MMU
- Can we avoid reading the page table on every read access?
The Translation Look-aside Buffer (TLB)

- The TLB is a small cache used by the MMU
  - Usually fewer than 10 entries, fully associative
- It caches the contents of final translation
  - Must be invalidated whenever the translation changes (the `inv1pg` instruction)
- MMU checks TLB if it contains translation
  - If it does, no page table walk is performed
## Entry Formats

<table>
<thead>
<tr>
<th>X</th>
<th>D</th>
<th>Ignored</th>
<th>Rsvd.</th>
<th>Address of PML4 table</th>
<th>Ignored</th>
<th>CR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-1</td>
<td>Ignored</td>
<td>Address of page-director-pointer table</td>
<td>Ignored</td>
<td>PML4E: present</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Ignored</td>
<td>Address of page directory</td>
<td>Ignored</td>
<td>PDE: not present</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ignored</td>
<td>Address of page table</td>
<td>Ignored</td>
<td>PDE: not present</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ignored</td>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>PTE: not present</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ignored</td>
<td>Address of 2MB page frame</td>
<td>Ignored</td>
<td>PDE: 2MB page</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Ignored</td>
<td>Address of 1GB page frame</td>
<td>Ignored</td>
<td>PDPE: page directory</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Reserved
- PML4 table
- Page-directory-pointer table
- Page directory
- Page table
- Page frame

### X, D
- Present
- Not present

### Prot.
- Present
- Not present

### Key
- Present
- Not present

### Ignored
- Present
- Not present

### Address Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Present</th>
<th>Not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2MB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4KB</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### CR3
- Present
- Not present
Page Permissions

- **U/S: User/Supervisor**
  - if this bit is 0, a user-space process cannot read/write this page
  - this is how the kernel protects itself from user programs (among other “defenses”)

- **R/W: Read/Write**
  - if this bit is 0, this page cannot be written
  - e.g. pages containing code or read-only data

- **XD: eXecute Disable (Intel terminology)**
  - if this bit is 1 and the machine supports XD, then instructions cannot be fetched from this page

Attempting to perform forbidden actions causes the CPU to generate a fault.
How Caches Change with Virtual Memory

- Should you use virtual addresses to index the cache?
  - Should you do this at all levels?
- Which address should the tag be constructed from?
  - Virtual or physical?
How Prefetchers Change with Virtual Memory

- Recall, a prefetcher fetches data before it is required.
- What happens if a prefetcher tries to fetch data from a page that is swapped out?
Outline

- Administrivia
- Recap
- Memory Virtualization
- x86-64 Implementation of Virtual Memory
- OS implementation of virtual memory
OS Implementations of Virtual Memory

- mmap
- mprotect
References and Acknowledgements

- Chapter 9
- Acknowledgements
  - Figure of memory layout from the textbook