CSC 252: Computer Organization
Spring 2018: Lecture 22

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:
• Programming Assignment 5 is due Monday
Announcement

- Programming Assignment 5 is out
  - Main assignment: 11:59pm, **Monday, April 16**.
- No office hours today. Had already moved to this Tuesday.
A System Using Virtual Addressing

CPU Chip

- Virtual address (VA) 4100
- Physical address (PA)
- Main memory
- Data word
A System Using Virtual Addressing

- On a 64-bit machine, virtual memory size = $2^{64}$
- Physical memory size is much much smaller:
  - iPhone 8: 2 GB ($2^{31}$)
  - 15-inch Macbook Pro 2017: 16 GB ($2^{34}$)
VM Concepts

• Conceptually, *virtual memory* is an array of N *pages* stored on disk. The contents of the array on disk are cached in *physical memory*, which is an array of M pages (M << N).
VM Concepts

- Conceptually, *virtual memory* is an array of N *pages* stored on disk. The contents of the array on disk are cached in *physical memory*, which is an array of M pages (M << N).
VM Concepts

• Page size is the same in VM and PM

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page Number</td>
<td>offset</td>
</tr>
</tbody>
</table>
VM Concepts

- Page size is the same in VM and PM
- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4KB page size.
VM Concepts

- Page size is the same in VM and PM
- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4KB page size.
- How many bits for offset?
  - 12. Same for VM and PM
VM Concepts

- Page size is the same in VM and PM
- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4KB page size.
- How many bits for offset?
  - 12. Same for VM and PM
- How many bits for Virtual Page Number?
  - 52

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page Number</td>
<td>offset</td>
</tr>
</tbody>
</table>
VM Concepts

- Page size is the same in VM and PM
- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4KB page size.
- How many bits for offset?
  - 12. Same for VM and PM
- How many bits for Virtual Page Number?
  - 52
- How many bits for Physical Page Number?
  - 20

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page Number</td>
<td>offset</td>
</tr>
</tbody>
</table>
Page Table: Enabling VA to PA Translation

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0</td>
<td>0</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>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PTE 7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Page Table: Enabling VA to PA Translation

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.

```
<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PTE 0</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>
</tbody>
</table>

|       | PTE 7                                 |
| 0     |                                       |
| 0     |                                       |
| 1     |                                       |

<table>
<thead>
<tr>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
</tr>
<tr>
<td>VP 3</td>
</tr>
<tr>
<td>VP 4</td>
</tr>
<tr>
<td>VP 6</td>
</tr>
<tr>
<td>VP 7</td>
</tr>
</tbody>
</table>
Page Table: Enabling VA to PA Translation

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.
A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.
Page Table: Enabling VA to PA Translation

- A **page table** is an array of **page table entries (PTEs)** that maps every virtual page to its physical page.
- 64-bit machine, 4KB page size, how many PTEs?
  - Every page has a PTE, so $2^{52}$ PTEs.
Address Translation With a Page Table

Virtual address

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

Physical address

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

Physical address
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 table address for the current process

Page table base register (PTBR)

Physical address

Physical page number (PPN)  Physical page offset (PPO)
Address Translation With a Page Table

Virtual address

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

Page table base register (PTBR)

Physical page table address for the current process

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

Physical address

Page table

Valid  Physical page number (PPN)
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 table address for the current process

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

Physical address

Physical page number (PPN)  Physical page offset (PPO)
Address Translation With a Page Table

Virtual address

Page table base register (PTBR)

Physical page table address for the current process

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

Valid bit = 1

Physical address

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

Physical page number (PPN)
Physical page offset (PPO)
Address Translation: Page Hit

CPU Chip

CPU  MMU

Memory
1) Processor sends virtual address to MMU
Address Translation: Page Hit

1) Processor sends virtual address to MMU
Address Translation: Page Hit

1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory
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
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

CPU Chip

CPU

MMU

Memory

Disk
1) Processor sends virtual address to MMU
Address Translation: Page Fault

1) Processor sends virtual address to MMU
Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
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
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)
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
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
Integrating VM and Cache

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

**CPU Chip**

- **CPU**
- **VA**

**MMU**

- **PTEA**
- **PTE**

**Memory**

- **L1 cache**

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

**VA**: virtual address, **PA**: physical address, **PTE**: page table entry, **PTEA** = PTE address
Integrating VM and Cache

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

**CPU Chip**

- CPU
- MMU
- Memory

**L1 cache**

**VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address**
Integrating VM and Cache

CPU Chip

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

CPU Chip

CPU

VA

MMU

PTE

PTEE

L1 cache

Memory

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

CPU Chip

CPU

MMU

L1 cache

Memory

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Integrating VM and Cache

CPU Chip

CPU ➔ VA ➔ MMU ➔ PTE ➔ Memory

Data

L1 cache

PTE

PTEA

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Today

• Three Virtual Memory Optimizations
  • TLB
    • Page the page table (a.k.a., multi-level page table)
    • Virtually-indexed, physically-tagged cache
• Case-study: Intel Core i7/Linux example
Speeding up Address Translation
Speeding up Address Translation

• Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  • The PTE access is kind of an overhead
  • Can we speed it up?
Speeding up Address Translation

• Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  • The PTE access is kind of an overhead
  • Can we speed it up?
• Page table entries (PTEs) are already cached in L1 data cache like any other memory data. But:
  • PTEs may be evicted by other data references
  • PTE hit still requires a small L1 delay
Speeding up Translation with a TLB

• Solution: *Translation Lookaside Buffer* (TLB)
  • Think of it as a dedicated cache for page table
  • Small set-associative hardware cache in MMU
  • Contains complete page table entries for a small number of pages
Speeding up Translation with a TLB

• Solution: *Translation Lookaside Buffer* (TLB)
  • Think of it as a dedicated cache for page table
  • Small set-associative hardware cache in MMU
  • Contains complete page table entries for a small number of pages

| Tag | Set Index |
Speeding up Translation with a TLB

• Solution: *Translation Lookaside Buffer* (TLB)
  • Think of it as a dedicated cache for page table
  • Small set-associative hardware cache in MMU
  • Contains complete page table entries for a small number of pages

![A Conventional Data Cache]
Speeding up Translation with a TLB

• Solution: *Translation Lookaside Buffer* (TLB)
  • Think of it as a dedicated cache for page table
  • Small set-associative hardware cache in MMU
  • Contains complete page table entries for a small number of pages

![Diagram of TLB and cache]

- A Conventional Data Cache
Speeding up Translation with a TLB

• Solution: *Translation Lookaside Buffer* (TLB)
  • Think of it as a dedicated cache for page table
  • Small set-associative hardware cache in MMU
  • Contains complete page table entries for a small number of pages

A Conventional Data Cache
Accessing the TLB

• MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

```
<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-1</td>
<td>p</td>
</tr>
<tr>
<td>n-1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

A Page Table Cache

**Diagram:**
- Set 0
- Set 1
- Set T-1
  - `v` for virtual page
  - `tag` for tag
  - `PTE` for page table entry
Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

**Diagram:**

```
              p+t-1  p+t  p   p-1  0
Virtual Page Number

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>...</th>
<th>Set T-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>tag</td>
<td>tag</td>
<td>tag</td>
<td>tag</td>
</tr>
<tr>
<td>PTE</td>
<td>PTE</td>
<td></td>
<td>PTE</td>
</tr>
</tbody>
</table>
```

A Page Table Cache
Accessing the TLB

• MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

```
Virtual Page Number
n-1  p+t  p+t-1  p  p-1  0
```

TLB tag (TLBT)  TLB index (TLBI)  Offset

TLBI selects the set

A Page Table Cache
Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

```
<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set T-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>tag</td>
<td>tag</td>
<td>tag</td>
</tr>
<tr>
<td>PTE</td>
<td>PTE</td>
<td>PTE</td>
</tr>
</tbody>
</table>
```

TLBT matches tag of line within set

```
Virtual Page Number
n-1  p+t  p+t-1  p  p-1  0
```

TLB tag (TLBT)  TLB index (TLBI)  Offset

TLBI selects the set

A Page Table Cache
TLB Hit

CPU Chip

CPU

MMU

TLB

Cache/Memory
TLB Hit

CPU Chip

CPU → TLB → MMU → Cache/Memory

1 VA
TLB Hit

CPU Chip

1. CPU

2. MMU

3. TLB

4. VPN

Cache/Memory
TLB Hit

CPU Chip

CPU

TLB

MMU

Cache/Memory

1

VA

2

VPN

3

PTE
TLB Hit

CPU Chip

CPU → MMU → Cache/Memory

1. CPU (VA)
2. TLB (VPN)
3. PTE
4. Cache/Memory (PA)
TLB Hit

Carnegie Mellon

TLB Hit
TLB Hit

A TLB hit eliminates a memory access
TLB Miss

CPU Chip

CPU

MMU

TLB

Cache/Memory

1 VA

2 VPN
TLB Miss

CPU Chip

1
VA

2
VPN

3
PTEA

CPU

MMU

TLB

Cache/
Memory
TLB Miss

CPU Chip

CPU

TLB

MMU

VA

VPN

PTE

PTEA

Cache/Memory
TLB Miss

CPU Chip

1. CPU → VA
2. VPN
3. PTEA
4. PTE
5. PA

TLB

MMU

Cache/Memory
TLB Miss

CPU Chip

1. VA

2. VPN

3. PTEA

4. PTE

5. PA

6. Data

Carnegie Mellon

CPU

MMU

TLB

Cache/Memory
A TLB miss incurs an additional memory access (the PTE).
Fortunately, TLB misses are rare.
Today

• Three Virtual Memory Optimizations
  • TLB
  • Page the page table (a.k.a., multi-level page table)
    • Virtually-indexed, physically-tagged cache
• Case-study: Intel Core i7/Linux example
Where Does Page Table Live?
Where Does Page Table Live?

• It needs to be at a specific location where we can find it
  • In main memory, with its start address stored in a special register (PTBR)
Where Does Page Table Live?

• It needs to be at a specific location where we can find it
  • In main memory, with its start address stored in a special register (PTBR)
• Assume 4KB page, 48-bit virtual memory, each PTE is 8 Bytes
  • $2^{36}$ PTEs in a page table
  • 512 GB total size per page table??!!
Where Does Page Table Live?

• It needs to be at a specific location where we can find it
  • In main memory, with its start address stored in a special register (PTBR)

• Assume 4KB page, 48-bit virtual memory, each PTE is 8 Bytes
  • $2^{36}$ PTEs in a page table
  • 512 GB total size per page table??!!

• Problem: Page tables are huge
  • One table per process!
  • Storing them all in main memory wastes space
Solution: Page the Page Table

• Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
• Put only the relevant page table entries in main memory
• Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

• Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
• Put only the relevant page table entries in main memory
• Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

• Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
• Put only the relevant page table entries in main memory
• Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

• Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
• Put only the relevant page table entries in main memory
• Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

• Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
• Put only the relevant page table entries in main memory
• Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entries in main memory
- Idea: Put Page Table in Virtual Memory and swap it just like data
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entries in main memory
- Idea: Put Page Table in Virtual Memory and swap it just like data
Effectively: A 2-Level Page Table

- **Level 1 table:**
  - Always in memory at a known location.
  - Each L1 PTE points to the start address of a L2 page table.
  - Bring that table to memory on-demand.

- **Level 2 table:**
  - Each PTE points to an actual data page
A Two-Level Page Table Hierarchy

32 bit addresses, 4KB pages, 4-byte PTEs
A Two-Level Page Table Hierarchy

32 bit addresses, 4KB pages, 4-byte PTEs
A Two-Level Page Table Hierarchy

Level 1 page table

Level 2 page tables

Virtual memory

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

1023 null PTEs

PTE 1023

PTE 0

PTE 1023

VP 0

VP 1023

VP 1024

VP 2047

unallocated pages

unallocated pages

VP 9215

...
A Two-Level Page Table Hierarchy

- **Level 2 page table size:**
  \[ \frac{2^{32}}{2^{12}} \times 4 = 4 \text{ MB} \]

- **Level 1 page table size:**
  \[ \frac{\left(\frac{2^{32}}{2^{12}}\times 4\right)}{2^{12}} \times 4 = 4 \text{ KB} \]

32 bit addresses, 4KB pages, 4-byte PTEs
How to Access a 2-Level Page Table?

<table>
<thead>
<tr>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>0011</td>
<td>0100</td>
</tr>
<tr>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>0101</td>
<td>0110</td>
</tr>
<tr>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>0111</td>
<td>1100</td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
</tr>
<tr>
<td>1101</td>
<td>1110</td>
</tr>
<tr>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

Page Table
How to Access a 2-Level Page Table?

Level 1 Table

Level 2 Tables

VPN | VPO
---|---
00 | 00
01 | 01
10 | 10
11 | 11

0001
How to Access a 2-Level Page Table?

<table>
<thead>
<tr>
<th>VPN 1</th>
<th>VPN 2</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0001</td>
</tr>
</tbody>
</table>

Level 2 Tables

- Level 1 Table
  - 00
  - 01
  - ... (indicated by '...')
  - 11

- Level 2 Tables
  - 00
  - 01
  - 10
  - 11
How to Access a 2-Level Page Table?

1. Use the VPN to access the Level 1 Table.
2. Use the VPN from the Level 1 Table to access the Level 2 Table.
3. Use the VPO to access the desired page.
How to Access a 2-Level Page Table?

![Diagram of a 2-level page table]

- **VPN 1**
  - Level 1 Table
    - 00
    - 01
    - 11
  - 00
  - 01
  - 10
  - 11

- **VPN 2**
  - Level 2 Tables
    - 00
    - 01
    - 10
    - 11

- **VPO**
  - 00
  - 01
How to Access a 2-Level Page Table?

Page table base register (PTBR)

VIRTUAL ADDRESS

m-1

VPN

n-1

PPO

PPN

PHYSICAL ADDRESS

p-1

0

VPO

page table

0

p-1

m-1

n-1
How to Access a 2-Level Page Table?

Page table base register (PTBR)

VIRTUAL ADDRESS

PHYSICAL ADDRESS

VPN 1

Level 1 page table

VPN 2

Level 2 page table

VPN

PPN

PPO

m-1

p-1

0

n-1

p-1

0

Virtual Address

Physical Address
Translating with a k-level Page Table
Today

• Three Virtual Memory Optimizations
  • TLB
  • Page the page table (a.k.a., multi-level page table)
  • Virtually-indexed, physically-tagged cache
• Case-study: Intel Core i7/Linux example
Performance Issue in VM

- Address translation and cache accesses are serialized
  - First translate from VA to PA
  - Then use PA to access cache
  - Slow! Can we speed it up?

![Diagram of address translation and cache access]

**CPU Chip**

- CPU
- MMU
- L1 cache
- Memory
- VA
- PA
- Data
- PA hit
- PA miss
Performance Issue in VM

Virtual Address

Virtual page number (VPN) → Page Offset

Physical Address

Physical page number (PPN) → Page Offset

Tag → Set Index → Cache Line Offset

L1 cache
Performance Issue in VM

Virtual Address

Virtual page number (VPN)  Page Offset

Physical page number (PPN)  Page Offset

Tag  Set Index  Cache Line Offset

Unchanged!!

L1 cache
Performance Issue in VM

Virtual Address
- Virtual page number (VPN)
- Page Offset

Physical Address
- Physical page number (PPN)
- Page Offset

Unchanged!!

Tag
Set Index
Cache Line Offset

L1 cache
Performance Issue in VM

- Set Index + Cache Line Offset = Page Offset
- Indexing into cache in parallel with translation (TLB access)
- If TLB hits, can get the data back in one cycle
Performance Issue in VM

- Set Index + Cache Line Offset = Page Offset
- Indexing into cache in parallel with translation (TLB access)
- If TLB hits, can get the data back in one cycle
Any Implications?

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Virtual page number (VPN)</th>
<th>Page Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Tag</th>
<th>Set Index</th>
<th>Cache Line Offset</th>
</tr>
</thead>
</table>
Any Implications?

- Assuming 4K page size, cache line size is 16 bytes.
Any Implications?

- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
Any Implications?

- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
- Increasing cache size then requires increasing associativity
Any Implications?

• Assuming 4K page size, cache line size is 16 bytes.
• Set Index = 8 bits. Can only have 256 Sets => Limit cache size
• Increasing cache size then requires increasing associativity
  • Not ideal because that requires comparing more tags
Any Implications?

- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
- Increasing cache size then requires increasing associativity
  - Not ideal because that requires comparing more tags
- Solutions?
Any Implications?

Virtual Address

Virtual page number (VPN)

Page Offset

Physical Address

Tag

Set Index

Cache Line Offset

12 bits

9 bits

4 bits

• What if we use 9 bits for Set Index? More Sets now.
Any Implications?

- What if we use 9 bits for Set Index? More Sets now.
- How can this still work???
Any Implications?

• What if we use 9 bits for Set Index? More Sets now.
• How can this still work???
• The least significant bit in VPN and PPN must be the same
Any Implications?

- What if we use 9 bits for Set Index? More Sets now.
- How can this still work???
- The least significant bit in VPN and PPN must be the same
- That is: an even VA must be mapped to an even PA, and an odd VA must be mapped to an odd PA
Today

• Three Virtual Memory Optimizations
  • TLB
  • Page the page table (a.k.a., multi-level page table)
  • Virtually-indexed, physically-tagged cache

• Case-study: Intel Core i7/Linux example
Intel Core i7 Memory System

Processor package

Core x4

- Registers
- L1 d-cache: 32 KB, 8-way
- L1 i-cache: 32 KB, 8-way
- L2 unified cache: 256 KB, 8-way
- Instruction fetch
- L1 d-TLB: 64 entries, 4-way
- L1 i-TLB: 128 entries, 4-way
- L2 unified TLB: 512 entries, 4-way
- MMU (addr translation)
- 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
End-to-End Core i7 Address Translation

Virtual address (VA)

CPU

36

12

VPN  VPO
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN
VPO

TLBT
TLBI

36
12
32
4
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN VPO

36 12

32 4

TLBT TLBI

L1 TLB (16 sets, 4 entries/set)
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN VPO

36 12

TLBT TLBI

32 4

L1 TLB (16 sets, 4 entries/set)

hit

TLB

PPN

40
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN

VPO

36

12

TLBT

TLBI

32

4

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

9

9

9

9

CR3

PTE

Page tables

40

VPN3

VPN4

Page tables

PTE

PTE

PTE

PTE
End-to-End Core i7 Address Translation

Virtual address (VA)

CPU

VPN

VPO

TLBT

TLBI

TLB hit

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

CR3

Page tables

PTE

PTE

PTE

PTE

PPN

PPO
End-to-End Core i7 Address Translation

Virtual address (VA)

CPU

VPN

VPO

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

Physical address (PA)

VPN

VPO

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

Physical address (PA)
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN VPO

TLBT TLBI

36 12

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

9 9 9 9

CR3

PTE

Page tables

PTE

PTE

PTE

40 12

PPN PPO

CT CI CO

Physical address (PA)
End-to-End Core i7 Address Translation

![Diagram showing address translation process]

- **CPU**
- **Virtual address (VA)**
  - **VPN**
  - **VPO**
  - **TLBT**
  - **TLBI**
  - **TLB hit**
  - **TLB miss**
  - **L1 TLB (16 sets, 4 entries/set)**
  - **L1 d-cache (64 sets, 8 lines/set)**

- **VPN1**
- **VPN2**
- **VPN3**
- **VPN4**
- **VPN1**
- **VPN2**
- **VPN3**
- **VPN4**
- **PPN**
- **PPO**
- **Physical address (PA)**
- **CR3**
- **Page tables**
- **PTE**

- **CT**
- **CI**
- **CO**
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN

VPO

36

12

TLBT

TLBI

32

4

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

9

9

9

9

CR3

PTE

PTE

PTE

PTE

Page tables

VPN1

VPN2

VPN3

VPN4

9

9

9

9

PPN

PPO

40

12

32/64

Result

L1

hit

L1 d-cache

(64 sets, 8 lines/set)

CT

CI

CO

Physical address (PA)
End-to-End Core i7 Address Translation

Virtual address (VA) → CPU

VPN → VPO

TLB:
- TLBT
- TLBI

TLB hit → L1 TLB (16 sets, 4 entries/set)

TLB miss → L1 d-cache (64 sets, 8 lines/set)

L1 hit → Result

L1 miss → L1 TLB (16 sets, 4 entries/set)

VPN1 → VPN2 → VPN3 → VPN4

CR3 → PTE

Page tables

PPN → PPO

Physical address (PA) → CT → CI → CO
### 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) | P=0 |

**Each entry references a 4K child page. Significant fields:**

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

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

**XD:** Disable or enable instruction fetches from this page.
### 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>

Each entry references a 4K child page table. Significant fields:

- **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.
- **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).

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

**XD**: Disable or enable instruction fetches from all pages reachable from this PTE.

Available for OS (page table location on disk) | P=0
Core i7 Page Table Translation

CR3
Physical address of L1 PT

L1 PT
Page global directory

L1 PTE
512 GB region per entry

VPN 1
9

L2 PT
Page upper directory

L2 PTE
1 GB region per entry

VPN 2
9

L3 PT
Page middle directory

L3 PTE
2 MB region per entry

VPN 3
9

L4 PT
Page table

L4 PTE
4 KB region per entry

VPN 4
9

VPO
12

VPN
Virtual address

Offset into physical and virtual page

Physical address

PPN

PPO

40

49

12

1/12