Address Translation

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

• A4 is due 11:59pm, Thursday, November 18

• Remember to let us know about slip days

• Also remember to type the dot when running the turn-in script

• A5 will be out sometime in the next couple of days, I’ll send an announcement when it is ready and go over it on Monday
Virtual Memory

Virtual Memory of Process 1

99  Unallocated
100  Data 1
101  ...
102  Data 2
103  ...
104  ...
105  ...

Virtual Memory of Process 2

99  ...
100  Unallocated
101  ...
102  Data 3
103  Data 2
104  ...
105  ...

Page Table of Process 1

...  ...
...
...
...
...

Page Table of Process 2

...  ...
...
...
...
...

Physical/Main Memory

1  Data 1
2  ...  ...
3  Data 3
4  Data 2

Hard Drive

A  ...
B  ...
C  ...
D  ...
O  ...
P  ...
Q  ...
R  ...

Process 1

Data 1
...
...
...
...
Unallocated
Data 2
...
...
...
...
...
Process 2

Data 3
...
...
...
...
...
...
...
Unallocated
...
...
...
...
...
...
...
A System Using Virtual Memory

• The memory management unit (MMU) does the VA to PA translation, and moves data between physical memory and disk.
Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
- How many bits for page offset?
  - 12. Same for 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>
Calculate Bits in VA and PA

• In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
• How many bits for page offset?
  • 12. Same for VM and PM
• How many bits for Virtual Page Number?
  • 52, i.e., $2^{52}$ virtual pages
Calculate Bits in VA and PA

• In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
• How many bits for page offset?
  • 12. Same for VM and PM
• How many bits for Virtual Page Number?
  • 52, i.e., $2^{52}$ virtual pages
• How many bits for Physical Page Number?
  • 20, i.e., $2^{20}$ physical pages

<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>
Today

- VM basic concepts and operation
- Other critical benefits of VM
- Address translation
VM as a Tool for Memory Management

• Each process has its own virtual address space
  • It can view memory as a simple linear array
  • Mapping scatters addresses through physical memory
    • Well-chosen mappings can improve locality
Virtual Memory Enables Sharing

- Simplifying 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)
VM Provides Further Protection Opportunities

- Extend PTEs with permission bits
- MMU checks these bits on each access (read/write/executable accessible only in supervisor mode?)
- Remember buffer overflow attack?

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</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>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
Today

• VM basic concepts and operation
• Other critical benefits of VM
• Address translation
So Far…

What does an MMU do?

- Translate address from a VA to PA
  - Enforce permissions
  - Fetch from disk
Address Translation With a Page Table

Virtual address (issued by CPU)

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

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

Physical address (what will be used to access the DRAM)
Address Translation With a Page Table

Virtual address *(issued by CPU)*

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

Page table *(in the physical memory)*

Valid Physical page number (PPN)

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

Physical address *(what will be used to access the DRAM)*
Address Translation With a Page Table

Virtual address *(issued by CPU)*

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

Page table *(in the physical memory)*

Valid Physical page number (PPN)

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

Physical address *(what will be used to access the DRAM)*

Page table base register (PTBR)

Physical page table address for the current process
Address Translation With a Page Table

Virtual address (issued by CPU)

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

Page table (in the physical memory)

Valid Physical page number (PPN)

Physical page table address for the current process

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

Physical address (what will be used to access the DRAM)
Address Translation With a Page Table

Virtual address *(issued by CPU)*

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

Page table *(in the physical memory)*

Valid Physical page number (PPN)

Physical page table address for the current process

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

Physical address *(what will be used to access the DRAM)*

Page table base register (PTBR)

PTEA = PTBR + VPN * sizeof (PTE)
Address Translation With a Page Table

**Virtual address (issued by CPU)**

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

**Page table (in the physical memory)**

Valid Physical page number (PPN)

PTEA = PTBR + VPN * sizeof(PTE)

Physical page table address for the current process

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

**Physical address (what will be used to access the DRAM)**

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

Virtual address (issued by CPU)

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

Page table (in the physical memory)

Valid Physical page number (PPN)

Physical page table address for the current process

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

Physical address (what will be used to access the DRAM)

Valid bit = 1

PTEA = PTBR + VPN * sizeof (PTE)

Page table base register (PTBR)

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

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Address Translation: Page Hit

1) Processor sends virtual address to MMU

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Address Translation: Page Hit

1) Processor sends virtual address to MMU

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Address Translation: Page Hit

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

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

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

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Address Translation: Page Fault

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Address Translation: Page Fault

1) Processor sends virtual address to MMU

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Address Translation: Page Fault

1) Processor sends virtual address to MMU

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

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

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
5) Handler identifies victim (and, if dirty, pages it out to disk)

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
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory

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
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

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

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

Memory

L1 cache

PTEA

PTE

PTEA

miss

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

PTEA

PTEA hit

PTEA miss

PA

L1 cache

Memory

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 Chip

CPU

MMU

PTE

PA

L1 cache

Memory

PTEA

PTEA

PTEA

PA

PA

PA

Data

VA

Data

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

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

![Diagram of TLB sets](image)

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 sets with tags and data]

- 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:**
  - 3 sets (Set 0, Set 1, Set T-1)
  - Each set contains a tag and data
  - Tag is compared to decide cache hit/miss
  - Set Index selects a set

A Conventional Data Cache
Accessing the TLB

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

```
Virtual Page Number
n-1    p+t   p+t-1   p    p-1   0
| TLB tag (TLBT) | TLB index (TLBI) | Offset |
```

A Page Table Cache
Accessing the TLB

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

```
+---+---+---+---+
| 0 | 1 | 2 | 3 |
+---+---+---+---+
| 0 | 1 | 2 | 3 |
```

**TLB tag (TLBT)**

**TLB index (TLBI)**

**Offset**

- TLBI selects the set

```
Set 0
  v  tag  PTE
  v  tag  PTE

Set 1
  v  tag  PTE
  v  tag  PTE

Set T-1
  v  tag  PTE
  v  tag  PTE
```

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
```

```
TLBT matches tag of line within set
```

```
Set 0

```

```
Set 1

```

```
Set T-1

```

```
A Page Table Cache

TLBI selects the set
```
TLB Hit

CPU Chip

CPU

MMU

TLB

Cache/Memory
TLB Hit

![Diagram of CPU Chip, TLB, MMU, and Cache/Memory]
TLB Hit

CPU Chip

CPU

TLB

MMU

VPN

Cache/Memory
TLB Hit

CPU Chip

CPU

MMU

TLB

VPN

PTE

VA

Cache/Memory
TLB Hit
**TLB Hit**

**CPU Chip**

1. VA
2. VPN
3. PTE
4. PA
5. Data

**Diagram:**
- CPU
- TLB
- MMU
- Cache/Memory
- Data

**Abbreviations:**
- CPU: Central Processing Unit
- VA: Virtual Address
- VPN: Virtual Page Number
- PTE: Page Table Entry
- PA: Physical Address
- MMU: Memory Management Unit
- TLB: Translation Lookaside Buffer
A TLB hit eliminates a memory access
TLB Miss

CPU Chip

- CPU
- MMU
- TLB

1 VA

2 VPN

Cache/Memory
TLB Miss

CPU Chip

1. CPU
2. VPN
3. PTEA

TLB

MMU

Cache/Memory
TLB Miss

CPU Chip

CPU

MMU

TLB

VPN

PTE

PTEA

Cache/Memory

1 VA

2

3

4
TLB Miss

```
CPU Chip

1. CPU (VA) → MMU
2. MMU (VPN) → TLB
3. TLB (PTEA) → Cache/Memory
4. MMU (PTE) → Cache/Memory
5. MMU (PA) → Cache/Memory
```

TLB Miss
TLB Miss

CPU Chip

CPU

1 VA

MMU

2 VPN

TLB

3 PTEA

Cache/Memory

4 PTE

5 PA

Data

6
Today

• Three Virtual Memory Optimizations
  • TLB
  • Virtually-indexed, physically-tagged cache
  • Page the page table (a.k.a., multi-level page table)
• 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 CPU, MMU, and Memory with VA, PA, and Data paths]
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 Address

Physical page number (PPN) | Page Offset

Tag | Set Index | Cache Line Offset

L1 cache

Unchanged!!
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

Unchanged!!
Performance Issue in VM

Virtual Address

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

Physical Address

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

- 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?

Virtual Address

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

Physical Address

<table>
<thead>
<tr>
<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
• 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
• 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?

- 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
  • Virtually-indexed, physically-tagged cache
  • Page the page table (a.k.a., multi-level page table)
• 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 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
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

Virtual memory

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

- unallocated pages

- unallocated pages

- VP 9215

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

Virtual memory

Level 2 page tables

VP 0
...
VP 1023
VP 1024
...
VP 2047
unallocated pages
unallocated pages
VP 9215

PTE 0
...
PTE 1023

PTE 0
...
PTE 1023

1023 null PTEs
PTE 1023

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 0
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

PTE 1023

VP 0

VP 1023
VP 1024

VP 2047

unallocated pages

unallocated pages

VP 9215

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

Level 1 page table

Level 2 page tables

Virtual memory

- Level 2 page table size:
  - $2^{32} / 2^{12} \times 4 = 4$ MB
- Level 1 page table size:
  - $(2^{32} / 2^{12} \times 4) / 2^{12} \times 4 = 4$ KB

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

Page table base register (PTBR)

VIRTUAL ADDRESS

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

Page table base register (PTBR)

VIRTUAL ADDRESS

VPN 1

Level 1 page table

VPN 2

Level 2 page table

PPN

PHYSICAL ADDRESS

PPN

PPO

VPO

n-1

m-1

p-1
Translating with a k-level Page Table

Page table base register (PTBR)

VIRTUAL ADDRESS

PHYSICAL ADDRESS

VPN 1  VPN 2  ...  VPN k

Level 1 page table  Level 2 page table  ...  Level k page table

PPN  PPO

m-1  n-1  p-1  0 0
Today

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

• 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
  - **Instruction fetch**
    - **L2 unified cache**
      - 256 KB, 8-way
  - **MMU**
    - (addr translation)
      - **L1 d-TLB**
        - 64 entries, 4-way
      - **L1 i-TLB**
        - 128 entries, 4-way
    - **L2 unified TLB**
      - 512 entries, 4-way
  - **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

- CPU
- Virtual address (VA)
- VPN
- VPO

36
12
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN

VPO

36

12

32

4

TLBT

TLBI
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

TLB

hit

L1 TLB (16 sets, 4 entries/set)

PPN

40
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

VPN

VPO

TLBT

TLBI

TLB

PTE

PPN
End-to-End Core i7 Address Translation

Virtual address (VA)

VPN
VPO

36
12

32
4

TLBT TLBI

TLB
hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

9
9
9
9

CR3

PTE

Page tables

PPN PPO

40
12

37
End-to-End Core i7 Address Translation

Virtual address (VA)

CPU

VPN

VPO

36

12

PPO

40

12

 Physical address (PA)

TLBT TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

CR3

PTE

Page tables

VPN

VPO

32

4

VPN1

VPN2

VPN3

VPN4

9

9

9

9

40

12

PTE

PTE

PTE

PTE
End-to-End Core i7 Address Translation

Virtual address (VA)

CPU

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

VPN3 VPN4

9

9

9

9

PPN PPO

40

12

CT CI CO

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

Virtual address (VA)

L1 TLB (16 sets, 4 entries/set)

TLB hit

TLB miss

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

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

CPU

Virtual address (VA)

VPN VPO

36 12

32 4

TLBT TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

9 9 9 9

CR3

PTE

Page tables

VPN1 VPN2 VPN3 VPN4

32/64

Result

L1

hit

PPT PPO

PPN

L1 d-cache

(64 sets, 8 lines/set)

CT CI CO

Physical address (PA)
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

VPN

VPO

32

4

TLB

hit

Result

32/64

L1

hit

L1 miss

L1 d-cache

(64 sets, 8 lines/set)

VPN1 VPN2 VPN3 VPN4

9

9

9

9

PPN PPO

CT CI CO

Physical address (PA)
Core i7 Level 4 Page Table Entries

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.