CS202 - Virtual Memory

Where We Were

\[
\text{Po} \quad \text{mov} \ $7, 0xffff\ldots20
\]

\[
\text{Pl} \quad \text{mov} \ $9, 0xffff\ldots20
\]

How?

\[
\text{mov} \ $9, 0xffff\ldots20
\]

Virtual Address

\text{\textquotedblleft Deep copy\textquotedblright}"

Physical Address

\[
\text{movq}
\]

\text{Page Table Address}

% CR3, Physical Addr.
Today: Page Tables

Virtual Address (for now assume 48-bits)

![Diagram of page table]

Note: Physical & Virtual Addresses might have different lengths.

And 64 (some) - Virtual 48 bits
Physical 52 bits
Note: Remaining 64 - 48 = 16 bits?
All 0 or All 1

Page Tables

Core Problem: Lots of unused virtual addresses. Don't want to waste space.

Solution: Build a Tree

(Switch to sheet)

What is important:
- Faults: Not present or violate access control permissions.
- How to translate virtual address
  - VPN, Offset
  - Use VPN to walk page table
- Why the tree helps.

RESOLVING ADDRESSES

\[
\text{uint64_t } d[512]
\]

5x Extra accesses

Want to reduce overheads.
TRANSLATION LOOKASIDE BUFFER → SIZE Dictated by HARDWARE Usually B/W 64-1024 entries

- ON Miss?

TLB QUESTION

16 Entries.

Program that (after initial start) misses TLB on each instruction? [4096 x 1]

```c
for (i=0; p;)
    while (!c)
        `pul((a [i])
    i++
```
Process Memory Layout.
\[ 2^0 \quad \text{Byte} \]\n\[ 2^{10} \quad \text{Kilobyte} \quad (1024) \]
\[ 2^{20} \quad \text{MEGA Byte} \]
\[ 2^{30} \quad \text{GIGA Byte} \]

\[ 2^3, 2^9 \quad \gtrless 1 \text{ PT} \]

\[ \sim \text{En(Grey)} \]

\[ 12 \text{ bytes} \]

\[ = 2^{12} \]

\[ = 2^8 \cdot 2^4 \]

\[ = 4 \text{ k} \]
## Core i7 Level 4 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 physical base address</td>
<td>Unused</td>
<td>G</td>
<td>D</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>

Available for OS (for example, if 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 this 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: if bit set, we have 2 MB or 1 GB pages (bit can be set in Level 2 and 3 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.
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>PK</td>
</tr>
<tr>
<td>4</td>
<td>I/D</td>
</tr>
<tr>
<td>3</td>
<td>RSVD</td>
</tr>
<tr>
<td>2</td>
<td>WR</td>
</tr>
<tr>
<td>1</td>
<td>P</td>
</tr>
<tr>
<td>0</td>
<td>The fault was caused by a non-present page.</td>
</tr>
<tr>
<td>1</td>
<td>The fault was caused by a page-level protection violation.</td>
</tr>
<tr>
<td>0</td>
<td>The access causing the fault was a read.</td>
</tr>
<tr>
<td>1</td>
<td>The access causing the fault was a write.</td>
</tr>
<tr>
<td>0</td>
<td>A supervisor-mode access caused the fault.</td>
</tr>
<tr>
<td>1</td>
<td>A user-mode access caused the fault.</td>
</tr>
<tr>
<td>0</td>
<td>The fault was not caused by reserved bit violation.</td>
</tr>
<tr>
<td>1</td>
<td>The fault was caused by a reserved bit set to 1 in some paging-structure entry.</td>
</tr>
<tr>
<td>0</td>
<td>The fault was not caused by an instruction fetch.</td>
</tr>
<tr>
<td>1</td>
<td>The fault was caused by an instruction fetch.</td>
</tr>
<tr>
<td>0</td>
<td>The fault was not caused by protection keys.</td>
</tr>
<tr>
<td>1</td>
<td>There was a protection-key violation.</td>
</tr>
<tr>
<td>0</td>
<td>The fault is not related to SGX.</td>
</tr>
<tr>
<td>1</td>
<td>The fault resulted from violation of SGX-specific access-control requirements.</td>
</tr>
</tbody>
</table>

**Figure 4-12. Page-Fault Error Code**
End-to-end Core i7 Address Translation

Virtual address (VA) → VPN → TLBT → TLBI → TLB → L1 TLB (16 sets, 4 entries/set) → TLB hit → L1 hit → L1 d-cache (64 sets, 8 lines/set) → Physical address (PA) → CR3 → Page tables → PTE → PPN → PPO → Result → L2, L3, and main memory

CPU

VPN

VPO

TLBT

TLBI

36

12

32

4

9

9

9

9

40

12

40

6

6

VPN1

VPN2

VPN3

VPN4

PTE

Page tables

TLB

miss

TLB

hit

L1

hit

L1

miss

32/64

Result
Cute Trick for Speeding Up L1 Access

■ Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Cache carefully sized to make this possible: 64 sets, 64-byte cache blocks
- Means 6 bits for cache index, 6 for cache offset
- That’s 12 bits; matches VPO, PPO \( \rightarrow \) One reason pages are \( 2^{12} \) bits = 4 KB
Virtual Address Space of a Linux Process

- Process-specific data structs (ptables, task and mm structs, kernel stack)
- Physical memory
- Kernel code and data
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Identical for each process

Kernel virtual memory

Process virtual memory

Physical memory

brk

%rsp
Q. Pages Required To Allocate

- 1 Byte

- 4 KB  \( (\text{Page Size}) = 2^{12} \) Bytes

- \( 512 \cdot 4 \text{ KB} = 2^9 \cdot 2^{12} \) Bytes \( \equiv 2 \text{ MB} \)

- \( 512 \cdot 2 \text{ MB} = 1 \text{ GB} = 2^9 \cdot 2^9 \cdot 2^{12} \) Bytes