Operating Systems
ECE344

Translation Lookaside Buffer

Ashvin Goel
ECE
University of Toronto
Outline

- Translation lookaside buffer -- TLB
- Memory protection
Recall: Paging MMU

- Partition virtual memory into pages & physical memory into frames
- Translate virtual page to physical frame on each memory access
- Page and frame size are always a power of 2
  - Low n bits identify offset within page
- E.g., page size = 8 bytes = $2^3$

<table>
<thead>
<tr>
<th>page nr./frame nr.</th>
<th>offset 3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>11</td>
<td>.</td>
</tr>
<tr>
<td>11</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>111</td>
</tr>
<tr>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>.</td>
</tr>
<tr>
<td>01</td>
<td>.</td>
</tr>
<tr>
<td>00</td>
<td>111</td>
</tr>
<tr>
<td>00</td>
<td>.</td>
</tr>
<tr>
<td>00</td>
<td>.</td>
</tr>
<tr>
<td>00</td>
<td>000</td>
</tr>
</tbody>
</table>
Recall: Paging MMU

- Partition virtual memory into pages & physical memory into frames
- Translate virtual page to physical frame on each memory access
- Page and frame size are always a power of 2
  - Low n bits identify offset within page
- E.g., page size = 4096 bytes = $2^{12}$
Recall: Linear Page Table

32 bit virtual address: 0x1005a6f

frames in memory

Example architecture
- Virtual address size: 32 bits
- Page size: $2^{12}$ B (4KB)

Example vaddr->paddr translation:

\[
\begin{align*}
vaddr &= 0x01005a6f \\
\text{offset} &= vaddr \& 0xffff = 0xa6f \\
\text{page} &= vaddr >> 12 = 0x1005 \\
// \text{pte is page_table}[\text{page}] \\
\text{fr} &= \text{page_table}[\text{page}].\text{frame} = 0xe \\
// (\text{fr} \ll 12) \text{ is frame phys address} \\
paddr &= (\text{fr} \ll 12) | \text{offset} = 0xea6f
\end{align*}
\]
Recall: Multi-Level Page Table

32 bit virtual address: 0x1005a6f

10 bits 10 bits 12 bits

PT1 PT2 offset

frames in memory

0xa000

0xe000

0xe100

0xe500

0x0

0xa

0

0x0

0xa6f

0x0

0xa0

0xa

0xa000

0xa6f

0xe

0x1005a6f
Speeding up Address Translation

- Paging MMU maps virtual to physical address on each memory access using page tables in memory
  - Every memory access requires additional accesses

Linear Page Table
- \(\text{vaddr} = 0x01005a6f\)
- \(\text{offset} = \text{vaddr} \& 0xffff = 0xa6f\)
- \(\text{pg} = \text{vaddr} >> 12 = 0x1005\)
- \(\text{fr} = \text{ptr}[\text{pg}].frame = 0xe\)
- \(\text{paddr} = (\text{fr} << 12) | \text{offset} = 0xea6f\)

Two-Level Page Table
- \(\text{vaddr} = 0x01005a6f\)
- \(\text{offset} = \text{vaddr} \& 0xffff = 0xa6f\)
- \(\text{pg2} = (\text{vaddr} >> 12) \& 0x3ff = 0x5\)
- \(\text{pg1} = \text{vaddr} >> (12+10) = 0x4\)
- \(\text{pt}_2 = \text{ptr}[\text{pg1}].frame << 12 = 0xa000\)
- \(\text{fr} = \text{pt}_2[\text{pg2}].frame = 0xe\)
- \(\text{paddr} = (\text{fr} << 12) | \text{offset} = 0xea6f\)

- This address translation can be sped up by using a cache of page mappings
Translation Lookaside Buffer (TLB)

- TLB is a h/w cache of page table entries (PTE)
  - TLB has small nr. of entries (e.g., 64)
  - TLB exploits locality, programs need few pages at a time
- Each TLB entry contains
  - key: page number
  - Data: page table entry

<table>
<thead>
<tr>
<th>virtual page number (VPN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frame number (PFN)</th>
<th>unused</th>
<th>R</th>
<th>D</th>
<th>C</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
</table>

V: valid
W: writeable
C: Cacheable
D: Dirty (set by hardware)
R: Referenced (set by hardware)
TLB Operations

- Similar to operations on any cache
  - TLB lookup
  - TLB cache miss handling
  - TLB cache invalidate
TLB Lookup

Diagram showing the process of TLB lookup:

1. CPU sends a request to the TLB.
2. If there is a TLB hit, the page number and frame number are matched.
3. If there is a TLB miss, the page table is consulted to find the page frame number.
4. The page frame number is used to access physical memory.
If TLB lookup fails, then page table is looked up for correct entry, and TLB cache is filled.

Choice of which TLB entry is replaced is called TLB replacement policy.

TLB cache misses can be handled by hardware or OS.
TLB Miss in Hardware or Software

- **Hardware Managed TLB (x86)**
  - TLB handles misses in hardware
  - Hardware defines page table format and uses page table register to locate page table in physical memory
  - TLB replacement policy fixed by hardware

- **Software Managed TLB (MIPS, SPARC, HP PA)**
  - Hardware generates trap called **TLB miss fault**
  - OS handles TLB miss, similar to exception handling
    - OS figures out the correct page table entry, adds it in TLB
    - CPU has instructions for modifying TLB
  - Page tables become entirely an OS data structure
    - H/w doesn’t have a page table register
  - TLB replacement policy managed in software
TLB Cache Invalidate

- TLB is a cache of page table entries, needs to be kept consistent with page table
- Whenever OS modifies any page table entry, it needs to invalidate TLB entry
- On a context switch to another address space: TLB entries must be invalidated to prevent use of mappings of last address space
  - This is what really adds to the cost of context switching
    - Why?
    - We need a way to reduce this cost
TLB Cache Invalidate Options

- **Clear TLB**
  - Empty TLB by clearing the valid bit of all entries
  - Next thread will generate misses initially
  - Eventually, its caches enough of its own entries in the TLB

- **Tagged TLB**
  - Hardware maintains the current process id in a specific register
    - OS updates register on context switch
  - Hardware maintains an *id tag* with each TLB entry
  - On TLB fill, TLB tag is assigned current process id
  - On TLB lookup, TLB hit occurs only when TLB tag matches current process id
    - Enables space multiplexing of cache entries
    - Reduces need for invalidation significantly
Memory Protection

- Each memory region may need different protection
  - Text region: read, execute, no write
  - Stack and data regions: read, write, no execute

- We can use MMU to implement page-level protection
  - Page table entry has protection bits (e.g., page is writable)

- Page protection is enforced during address translation
  - E.g., generate protection fault when read-only page is written
Speeding Up Protection Enforcement

- Checking page table on each memory access is slow
  - TLB caches page-level protection bits
- TLB checks whether memory accesses are valid on each memory access
  - If memory access is inconsistent with protection bits, TLB generates protection fault
  - OS needs to invalidate TLB entry when page protection is changed

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Dirty</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>RX</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>RX</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>RX</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>
Software Managed TLB Faults

- When a TLB fault occurs, it can be:
  - TLB miss fault: no matching page number was found
    - Read fault: A read was attempted but no entry matched
    - Write fault: A write was attempted but no entry matched
  - TLB protection fault: a matching page number was found, but protection bits are inconsistent with operation
    - Read-only fault: A write to a word in a page was attempted but write-bit was not set (page is marked read-only)
    - No-execute fault: The execution of an instruction in a page was attempted but the execute-bit was not set (page is marked non-executable)
Summary

- Paging MMU adds significant performance overhead because each memory access requires additional memory accesses to the page table.

- TLB is a cache of page table entries:
  - Indexed by page number, returns frame number.

- TLB miss handling can be performed in h/w or s/w:
  - When performed in software, hardware only knows about TLB, not page table.

- OS needs to invalidate a TLB entry when it modifies the corresponding page table entry.

- Paging MMU can be used to provide page level memory protection.
Think Time

- What is the purpose of a TLB?
- When is TLB-miss fault generated?
- What are the benefits of using a hardware managed TLB?
- What are the benefits of a software managed TLB?
- What is a protection fault?
Think Time Answers

- **What is the purpose of a TLB?**
  - TLB speeds up address translation by keeping cache of page table entries, or virtual page -> physical frame mappings

- **When is TLB-miss fault generated?**
  - When the TLB doesn't contain a valid page mapping

- **What are the benefits of using a hardware managed TLB?**
  - TLB-miss handling is fast
Think Time Answers

- What are the benefits of a software managed TLB?
  - More flexibility with TLB-miss handling (e.g., OS can implement different TLB replacement policies)

- What is a protection fault?
  - Page access is inconsistent with protection bits associated with page