Translation Lookaside Buffer (TLB)

- **Problem:**
  - Unless we have hardware support for address translation, CPU would have to go to memory to access the page table on every memory access!
- **In Blitz, this is what happens**
Problem:

Unless we have hardware support for address translation, CPU would have to go to memory to access the page table on every memory access!

In real computers there is a TLB:

- Cache the page table entries in a hardware cache
- Small number of entries (e.g., 64)
- Each entry contains
  - Page number
  - Other stuff from page table entry
- Associatively indexed on page number
## Hardware operation of TLB

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Key**

D: Delete, R: Read, W: Write, V: Valid
# Hardware operation of TLB

## Page Number

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused</td>
<td>D</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused</td>
<td>D</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused</td>
<td>D</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused</td>
<td>D</td>
</tr>
</tbody>
</table>

## Virtual Address

<table>
<thead>
<tr>
<th>virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 1312 0</td>
</tr>
</tbody>
</table>

## Physical Address

<table>
<thead>
<tr>
<th>physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 1312 0</td>
</tr>
</tbody>
</table>

## Key

- **D**: Present
- **R**: Read
- **W**: Write
- **V**: Valid
**Hardware operation of TLB**

![Diagram of TLB operation]

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused D R W V</td>
</tr>
</tbody>
</table>

**Key**
- **Page Number**: Determines the page to be accessed.
- **Frame Number**: Specifies the frame within the page to be accessed.
- **Other**: Controls access permissions and visibility.

**Virtual Address**
- **Page Number**: 012 1323
- **Offset**: 012 1331

**Physical Address**
- **Frame Number**: 0x0
- **Offset**: 0x0

**Physical Address Calculation**
- The physical address is derived by combining the frame number and offset from the virtual address.
- For example, the physical address for virtual address 012 1323 012 1331 is calculated as:
  
  \[
  \text{Physical Address} = \text{Frame Number} \times \text{Frame Size} + \text{Offset}
  \]
Hardware operation of TLB

virtual address

page number offset

physical address

frame number offset

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused</td>
</tr>
</tbody>
</table>
## Hardware operation of TLB

A TLB (Translation Lookaside Buffer) is a hardware cache for virtual to physical address translation. It speeds up the process of address translation by caching recently used translations.

### Virtual Address to Physical Address Translation

The virtual address is divided into a page number and an offset. The page number is used to look up the corresponding page frame in the TLB. The offset is added to the base address of the page to get the physical address.

### TLB Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused</td>
</tr>
</tbody>
</table>

The table shows the page frame numbers for different page numbers. The other column indicates the permissions (D: Read, R: Write, W: Execute, V: Valid) for each page.
Hardware operation of TLB

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused D R W V</td>
</tr>
</tbody>
</table>

Key

virtual address

<table>
<thead>
<tr>
<th>23</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

physical address

<table>
<thead>
<tr>
<th>31</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame number</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>
Software operation of TLB

- What if the entry is not in the TLB?
  - Go to page table (how?)
  - Find the right entry
  - Move it into the TLB
  - Which entry to replace?
Software operation of TLB

- **Hardware TLB refill**
  - Page tables in specific location and format
  - TLB hardware handles its own misses

- **Software refill**
  - Hardware generates trap (**TLB miss fault**)
  - Lets the OS deal with the problem
  - Page tables become entirely a OS data structure!
Software operation of TLB

- What should we do with the TLB on a context switch?

- How can we prevent the next process from using the last process's address mappings?
  - Option 1: empty the TLB
    - New process will generate faults until its pulls enough of its own entries into the TLB
  - Option 2: just clear the “Valid Bit”
    - New process will generate faults until its pulls enough of its own entries into the TLB
  - Option 3: the hardware maintains a process id tag on each TLB entry
    - Hardware compares this to a process id held in a specific register ... on every translation
Page tables

- Do we access a page table when a process allocates or frees memory?
Page tables

- Do we access a page table when a process allocates or frees memory?
  - Not necessarily
  - Library routines (malloc) can service small requests from a pool of free memory already allocated within a process address space
  - When these routines run out of space a new page must be allocated and its entry inserted into the page table
    - This allocation is requested using a system call
    - Malloc is not a system call
Page tables

- We also access a page table during swapping/paging to disk
  - We know the page frame we want to clear
  - We need to find the right place on disk to store this process memory
  - Hence, page table needs to be searchable using physical addresses
    - Fastest approach, index page table using page frame numbers
Page tables

- In a well provisioned system, TLB miss faults will be the most frequently occurring event
- TLB miss fault
  - Given a virtual page number we must find the right page table entry
    - Fastest approach – index the page table using virtual page numbers
    - ...hmm, isn’t that the opposite of what we said last side?
  - Problem ... how to structure our page tables for efficient access and low space overhead
Page table design issues

- **Page table size depends on**
  - Page size
  - Virtual address length

- **Memory used for page tables is overhead!**
  - How can we save space?
  - ... and still find entries quickly?

- **Three options**
  - Single-level page tables
  - Multi-level page tables
  - Inverted page tables
Single-level page tables

A Virtual Address:

<table>
<thead>
<tr>
<th>21-bits</th>
<th>11-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td>offset</td>
</tr>
</tbody>
</table>

Single-level page table

frames in memory
Single-level page tables

- Single-level page table
- Page number (21-bits)
- Offset (11-bits)
- Frames in memory
Single-level page tables

Problem: requires one page table entry per virtual page!
Single-level page tables

32 bit addresses and 2KB pages means $2^{21}$ page table entries per process
Multi-level page tables

- Top-level Page table
- 2nd-level tables
- Frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

Top-level Page table → 2nd-level tables → Frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

- Top-level Page table
- 2nd-level tables
- Frames in memory
Multi-level page tables

A Virtual Address:

- Top-level Page table
- 2nd-level tables
- Frames in memory

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>
Multi-level page tables

- A Virtual Address:

10-bits 10-bits 12-bits

PT1  PT2  offset

frames in memory

Top-level Page table

2nd-level tables
Multi-level page tables

A Virtual Address:

10-bits 10-bits 12-bits

PT1  PT2  offset

Top-level Page table

2nd-level tables

frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

- Top-level Page table
- 2nd-level tables
- Frames in memory
Multi-level page tables

- Ok, but how exactly does this save space?
Multi-level page tables

- Ok, but how exactly does this save space?
- Not all pages within a virtual address space are allocated
  - Not only do they not have a page frame, but that range of virtual addresses is not being used
  - So no need to maintain complete information about it
  - Some intermediate page tables are empty and not needed

- We could also page the page table
  - This saves space but slows access ... a lot!
Inverted page tables

- **Problem:**
  - Page table overhead increases with address space size
  - Page tables get too big to fit in memory!

- **Consider a computer with 64 bit addresses**
  - Assume 4 Kbyte pages (12 bits for the offset)
  - Virtual address space = $2^{52}$ pages!
  - Page table needs $2^{52}$ entries!
  - This page table is much too large for memory!

- **But we only need mappings for pages that are in memory!**
  - A 256 Mbyte memory can only hold 64 4Kbyte pages
  - Only need 64 page table entries on this computer!
Inverted page tables

- An inverted page table
  - Has one entry for every frame of memory
  - Tells which page is in that frame
  - Is indexed by frame number not page number!

- So how can we search it on a TLB miss fault?
Inverted page tables

- If we have a page number (from a faulting address) and want to find it page table entry, do we
  - Do an exhaustive search of all entries?
Inverted page tables

- If we have a page number (from a faulting address) and want to find its page table entry, do we
  - Do an exhaustive search of all entries?
  - No, that’s too slow!
  - Why not maintain a hash table to allow fast access given a page number?
    - $O(1)$ lookup time with a good hash function
Inverted page table

Traditional page table with an entry for each of the $2^{52}$ pages

$2^{52} - 1$

Indexed by virtual page

256-GB physical memory has $2^{16}$ 4-KB page frames

$2^{16} - 1$

Indexed by hash on virtual page

Hash table

Virtual page

Page frame
Which page table design is best?

- The best choice depends on CPU architecture
- 64 bit systems need inverted page tables
- Some systems use a combination of regular page tables together with segmentation (later)
Memory protection

- At what granularity should protection be implemented?
  - page-level?
    ✷ A lot of overhead for storing protection information for non-resident pages
  - segment level?
    ✷ Coarser grain than pages
    ✷ Makes sense if contiguous groups of pages share the same protection status
Memory protection

- How is protection checking implemented?
  - compare page protection bits with process capabilities and operation types on every load/store
  - sounds expensive!
  - Requires hardware support!
- How can protection checking be done efficiently?
  - Use the TLB as a protection look-aside buffer
  - Use special segment registers
Protection lookaside buffer

- A TLB is often used for more than just “translation”
- Memory accesses need to be checked for validity
  - Does the address refer to an allocated segment of the address space?
    - If not: segmentation fault!
  - Is this process allowed to access this memory segment?
    - If not: segmentation/protection fault!
  - Is the type of access valid for this segment?
    - Read, write, execute ...?
    - If not: protection fault!
Page-grain protection checking with a TLB

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</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>R X</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>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</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>
Page grain protection in a page table

A typical page table entry with support for page grain protection
Segment-grain protection

- All pages within a segment usually share the same protection status
  - So we should be able to batch the protection information

- Why not just use segment-size pages?
  - Segments vary in size
  - Segments change size dynamically (stack, heap etc)
Segmented address spaces

- **Traditional Virtual Address Space**
  - “flat” address space (1 dimensional)

- **Segmented Address Space**
  - Program made of several “pieces”
  - Each segment is like a mini-address space
  - Addresses within a segment start at zero
  - The program must always say which segment it means
    - either embed a segment id in an address
    - or load a value into a segment register
  - Addresses:
    - Segment + Offset
  - Each segment can grow independently of others
Segmentation in a single address space

Example: A compiler

- Call stack
- Parse tree
- Constant table
- Source text
- Symbol table

Address space allocated to the parse tree

Free
Space currently being used by the parse tree
Symbol table has bumped into the source text table
Segmented memory

- Each space grows, shrinks independently!
Separate instruction and data spaces

* One address space
* Separate I and D spaces
Comparison of paging and segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection</td>
</tr>
</tbody>
</table>
Segmentation with paging (MULTICS)

- Each segment is divided up into pages.
- Each segment descriptor points to a page table.
Segmentation with paging (MULTICS)

- Each entry in segment table...

<table>
<thead>
<tr>
<th>18</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory address of the page table</td>
<td>Segment length (in pages)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page size:
0 = 1024 words
1 = 64 words

0 = segment is paged
1 = segment is not paged

Miscellaneous bits

Protection bits
Segmentation with paging: MULTICS

MULTICS virtual address

Segment number

Page number

Offset

Conversion of a 2-part MULTICS address into a main memory address
### Segmentation with Paging: TLB operation

- **Simplified version of the MULTICS TLB**
- **Existence of 2 page sizes makes actual TLB more complicated**

<table>
<thead>
<tr>
<th>Comparison field</th>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
<th>Is this entry used?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Execute only</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>Execute only</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
Spare Slides
Anatomy of a page fault

1. TLB miss
2. TLB miss
3. Page fault
4. Find Frame
5. Get page from backing store
6. Bring in page
7. Update PTE
Conversion of a (selector, offset) pair to a linear address
Segmentation & paging in the Pentium

A Pentium segment selector

bits

13

1 2

Index

0 = GDT/1 = LDT

Privilege level (0-3)
Segmentation & paging in the Pentium

- Pentium segment descriptor
Implementation Issues
Operating System Involvement with Paging

Four times when OS involved with paging

- **Process creation**
  - determine program size
  - create page table

- **Process execution**
  - MMU reset for new process
  - TLB flushed

- **Page fault time**
  - determine virtual address causing fault
  - swap target page out, needed page in

- **Process termination time**
  - release page table, pages