Who Handles The TLB Miss?

- Hardware handles the TLB miss entirely on CISC processors.
  - The hardware know where the page tables are located
  - … “walks” the page table, finding the correct entry and extracting the desired translation, update and retry instruction.
  - this is a hardware-managed TLB.

- RISC processors often manage TLBs in software.
  - On a TLB miss, the hardware raises an exception
    - Trap handler is code within the OS that is written with the express purpose of handling TLB misses.
TLB Control Flow algorithm (OS Handled)

- The hardware would do the following:

```plaintext
1: VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2: (Success, TlbEntry) = TLB_Lookup(VPN)
3: if (Success == True) // TLB Hit
4:   if (CanAccess(TlbEntry.ProtectBits) == True)
5:     Offset = VirtualAddress & OFFSET_MASK
6:     PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:     Register = AccessMemory(PhysAddr)
8:   else
9:     RaiseException(PROTECTION_FAULT)
10: else // TLB Miss
11:   RaiseException(TLB_MISS)
```

- But might be slow, why not just use the hardware approach?

TLB entry

- TLB entries are often fully associative (any entry for any mapping)
  - A typical TLB might have 32, 64, or 128 entries.
  - Hardware searches the TLB in parallel to find the translation.
  - other bits: valid, protection, address-space identifier, dirty bit

![Typical TLB entry](image_url)
TLB Issue: Context Switching

Process A

access VPN10

Virtual Memory

Page 0
Page 1
Page 2
Page n

Insert TLB Entry

TLB Table

VPN | PFN | valid | prot
---|-----|-------|-----
10  | 100 | 1     | rwx
-   | -   | -     | -
-   | -   | -     | -
-   | -   | -     | -

Process B

Virtual Memory

Page 0
Page 1
Page 2
Page n

Context Switching

TLB Issue: Context Switching

Process A

access VPN10

Virtual Memory

Page 0
Page 1
Page 2
Page n

TLB Table

VPN | PFN | valid | prot
---|-----|-------|-----
10  | 100 | 1     | rwx
-   | -   | -     | -
10  | 170 | 1     | rwx
-   | -   | -     | -
TLB Issue: Context Switching

Virtual Memory

TLB Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>1</td>
<td>rwx</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>170</td>
<td>1</td>
<td>rwx</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Can’t Distinguish which entry is meant for which process

Could just flush the TLB on every context switch...

Disambiguating Address Spaces

- Provide an address space identifier (ASID) field in the TLB.

Virtual Memory

TLB Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>1</td>
<td>rwx</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>170</td>
<td>1</td>
<td>rwx</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Another Case

- Two processes share a page.
  - Process 1 is sharing physical page 101 with Process 2.
  - P1 maps this page into the 10th page of its address space.
  - P2 maps this page to the 50th page of its address space.

<table>
<thead>
<tr>
<th>VPN</th>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>101</td>
<td>1</td>
<td>rwx</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>50</td>
<td>101</td>
<td>1</td>
<td>rwx</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Replacement Policy

- LRU (Least Recently Used)
  - Evict an entry that has not recently been used.
  - Take advantage of locality in the memory-reference stream.

Reference Row

Page Frame:

- 6 hits, 11 misses
A Real TLB Entry

64-bit MIPS R4000 TLB entry

<table>
<thead>
<tr>
<th>Flag</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>19-bit VPN</td>
<td>The rest reserved for the kernel.</td>
</tr>
<tr>
<td>24-bit PFN</td>
<td>Systems can support with up to 64GB of main memory (pages).</td>
</tr>
<tr>
<td>Global bit(G)</td>
<td>Used for pages that are globally-shared among processes.</td>
</tr>
<tr>
<td>ASID</td>
<td>OS can use to distinguish between address spaces.</td>
</tr>
<tr>
<td>Coherence bit(C)</td>
<td>determine how a page is cached by the hardware.</td>
</tr>
<tr>
<td>Dirty bit(D)</td>
<td>marking when the page has been written.</td>
</tr>
<tr>
<td>Valid bit(V)</td>
<td>tells the hardware if there is a valid translation present in the entry.</td>
</tr>
</tbody>
</table>

Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy
Paging: Linear (Single-Level) Tables

- Usually one page table for every process in the system.
- Example:
  - 32-bit address space, 4KB pages, 4-byte page-table entries

\[
\text{Page table size} = \frac{2^{32}}{2^{12}} \times 4\text{Byte} = 4\text{MByte}
\]

Paging: Smaller Tables

- Larger pages mean fewer entries
- 32-bit address space, 16KB pages, 4-byte entries.

\[
\frac{2^{32}}{2^{16}} \times 4 = 1\text{MByte} \text{ per page table}
\]

Big pages lead to internal fragmentation.
**Problem**

A Page Table For 16KB Address Space

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
<th>present</th>
<th>dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>rw-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>rw-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Most of the page table is **unused**

A Page Table For 16KB Address Space
Hybrid: Page Table Per Segment

- Each process has three page tables associated with it.
  - Base register for each segment is physical address of its page table.

![Diagram of page table structure]

32-bit Virtual address space with 4KB pages

<table>
<thead>
<tr>
<th>Seg value</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>unused segment</td>
</tr>
<tr>
<td>01</td>
<td>code</td>
</tr>
<tr>
<td>10</td>
<td>heap</td>
</tr>
<tr>
<td>11</td>
<td>stack</td>
</tr>
</tbody>
</table>

GeekOS!

TLB miss on Hybrid Approach

- Need physical address of entry from page table.
  - Segment bits (SN) determine which base and bounds pair
  - Hardware combines physical address therein and the VPN to form the address of the page table entry (PTE).

01: \[SN = (\text{VirtualAddress} \& \text{SEG\_MASK}) >> \text{SN\_SHIFT}\]
02: \[VPN = (\text{VirtualAddress} \& \text{VPN\_MASK}) >> \text{VPN\_SHIFT}\]
03: \[\text{AddressOfPTE} = \text{Base}[SN] + (\text{VPN} \times \text{sizeof(PTE)})\]
Multi-level Page Tables

- Hybrid Approach is not without problems
  - Sparsely-used heap still leads to external fragmentation
- Turns the linear page table into something like a tree
  - Page the page table
  - Allocate page-table pages as needed
  - Track valid page table pages with page directory
Multi-level Page Tables

- Page directory has:
  - one page directory entry (PDE) per page of the page table
  - Valid bit and page frame number (PFN)

- Advantages
  - Page-table space in proportion to used address space
  - OS can lazily allocate new pages as need
  - Indirection can disperse page-table pages through memory

- Disadvantages
  - Time and space tradeoff
  - Complexity

A Detailed Multi-Level Example

<table>
<thead>
<tr>
<th>Flag</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>16 KB</td>
</tr>
<tr>
<td>Page size</td>
<td>64 byte</td>
</tr>
<tr>
<td>Virtual address</td>
<td>14 bit</td>
</tr>
<tr>
<td>Num pages</td>
<td>$2^{14}/2^6 = 2^8 = 256$ pages</td>
</tr>
<tr>
<td>VPN</td>
<td>8 bit</td>
</tr>
<tr>
<td>Offset</td>
<td>6 bit</td>
</tr>
<tr>
<td>Page table entry</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

A 16-KB Address Space With 64-byte Pages
Detailed Example

- Page directory has one entry per page of the page table
  - 256 pages, 4 bytes for PTE, 64 byte pages
    - $256 \times 4/64 = 16$ pages
  - Accessing invalid page-directory entry raises exception

If PDE is valid:
- Fetch the page table entry (PTE) from the page of the page table pointed to by this page-directory entry.
- This page-table index can then be used to index into the page table itself.
### The Translation Process: Remember the TLB

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>PTIndex = (VPN &amp; PT_MASK) &gt;&gt; PT_SHIFT</td>
</tr>
<tr>
<td>19</td>
<td>PTEAddr = (PDE.PFN &lt;&lt; SHIFT) + (PTIndex * sizeof(PTE))</td>
</tr>
<tr>
<td>20</td>
<td>PTE = AccessMemory(PTEAddr)</td>
</tr>
<tr>
<td>21</td>
<td>if(PTE.Valid == False)</td>
</tr>
<tr>
<td>22</td>
<td>RaiseException(SEGMENTATION_FAULT)</td>
</tr>
<tr>
<td>23</td>
<td>else if(CanAccess(PTE.ProtectBits) == False)</td>
</tr>
<tr>
<td>24</td>
<td>RaiseException(PROTECTION_FAULT);</td>
</tr>
<tr>
<td>25</td>
<td>else</td>
</tr>
<tr>
<td>26</td>
<td>TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)</td>
</tr>
<tr>
<td>27</td>
<td>RetryInstruction()</td>
</tr>
</tbody>
</table>

### Inverted Page Tables

- Keeping only a single page table that has
  - an entry for each **physical page** of the system

- The entry tells us
  - which process is using this page, and
  - which virtual page that maps to this physical page

- Finding translating a virtual address now requires a search!
  - But can use a hash (PowerPC)
Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping and Demand Paging
- 22 - Swapping Policy

Beyond Physical Memory: Mechanisms

- Require an additional level in the memory hierarchy.
  - OS need a place to stash away portions of address space that currently aren’t in great demand.
  - In modern systems, this role is usually served by a hard drive

![Memory Hierarchy in modern system](image)
Single large address for a process

- Need to arrange for the code or data to be in memory before calling a function or accessing data.

- Beyond just a single process.
  - The addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running process.

Swap Space

- Reserve some space on the disk for swapping pages
Present Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk.
- When the hardware looks in the PTE, it may find that the page is not present in physical memory.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>page is present in physical memory</td>
</tr>
<tr>
<td>0</td>
<td>The page is not in memory but rather on disk.</td>
</tr>
</tbody>
</table>

- OS often needs to make room for the new pages
  - The process of picking a page to replace is known as page-replacement or victim-selection policy

The Page Fault

- Accessing page that is not in physical memory.
- A page with false present bit has either:
  - never been in-core (lazily loaded), or
  - has been swapped out to disk
Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk address.

When Replacements Really Occur

- Wait until memory entirely full?
  - No, proactively try to keep small portion of memory free.

- Swap or Page Daemon
  - Frees/evicts page frames if fewer than a low-water threshold available.
  - ...until a high-water threshold pages available.
Beyond Physical Memory: Policies

- **Memory pressure forces the OS to start** paging out pages to make room for actively-used pages.
- Deciding which page to **evict** is encapsulated within the replacement policy of the OS.
The Optimal Replacement Policy

- Leads to the fewest number of misses overall
- Replaces the page that will be accessed furthest in the future
- Resulting in the fewest-possible cache misses
- Serve only as a comparison point, to know how close we are to perfect

Tracing the Optimal Policy

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>0,1,3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td>0,1,3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td></td>
</tr>
</tbody>
</table>

Reference Row:

0 1 2 0 1 1 1 0 3 1 2 1

Hit rate is \( \frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 54.6\% \)
A Simple Policy: FIFO

- Pages were placed in a queue when they enter the system.
- When a replacement occurs, the page on the tail of the queue (the “First-in” pages) is evicted.
- It is simple to implement, but can’t determine the importance of blocks.

Tracing the FIFO Policy

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>0</td>
<td>1,2,3</td>
</tr>
<tr>
<td>0</td>
<td>Miss</td>
<td>1</td>
<td>2,3,0</td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>3,0,1</td>
<td>3,0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>3</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
</tbody>
</table>

| Reference Stream | Hit rate is \( \frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 36.4\% \) } |
BELADY’S ANOMALY

- We would expect the cache hit rate to increase when the cache gets larger.
  But with FIFO, it gets worse:

  - FIFO does not have stack policy
    - i.e. pages with $n$ frames always subset of pages with $n+1$ frames

Using History

- Lean on the past and use history.
  - Two type of historical information.

<table>
<thead>
<tr>
<th>Historical Information</th>
<th>Meaning</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>recency</td>
<td>The more recently a page has been accessed, the more likely it will be accessed again</td>
<td>LRU</td>
</tr>
<tr>
<td>frequency</td>
<td>If a page has been accessed many times, it should not be replaced as it clearly has some value</td>
<td>LFU</td>
</tr>
</tbody>
</table>
Using History : LRU

- Replaces the least-recently-used page.

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>1,2,0</td>
<td>1,2,0</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>2,0,1</td>
<td>2,0,1</td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>2</td>
<td>0,1,3</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>1,3,0</td>
<td>1,3,0</td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td>1,0,3</td>
<td>1,0,3</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,3,1</td>
<td>0,3,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0</td>
<td>3,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>3,2,1</td>
<td>3,2,1</td>
</tr>
</tbody>
</table>

The Exam

- GeekOS
- Synchronization
  - Deadlock mitigation
  - Trylocks, TestAndSet
  - Writing code
  - Semantics
- Queueing
  - Characteristics
  - Deriving queue lengths
  - changing my terminology: turnaround time, not response
- Paging and memory systems
  - segmentation
  - paging
  - multi-level page tables
  - victim-replacement policies: LRU, FIFO, MIN
The Exam (all point totals approximate)

- **GeekOS**: 5 pts
- **Synchronization**: 45 pts
  - Deadlock mitigation
  - Trylocks, TestAndSet
  - Writing code
  - Semantics
- **Queueing**: 25 pts
  - Characteristics
  - Deriving queue lengths
  - changing my terminology: turnaround time, not response
- **Paging and memory systems**: 25 pts
  - segmentation
  - paging
  - multi-level page tables
  - victim-replacement policies: LRU, FIFO, MIN