Page Tables Revisited

Learning Outcomes

• An understanding of virtual linear array page tables, and their use on the MIPS R3000.
• Exposure to alternative page table structures beyond two-level and inverted page tables.

Two-level Translation

R3000 TLB Refill

• Can be optimised for TLB refill only
  – Does not need to check the exception type
  – Does not need to save any registers
    • It uses a specialised assembly routine that only uses k0 and k1.
    – Does not check if PTE exists
  – Assumes virtual linear array – see extended OS notes
• With careful data structure choice, exception handler can be made very fast

Virtual Linear Array page table

• Assume a 2-level PT
• Assume 2nd-level PT nodes are in virtual memory
• Assume all 2nd-level nodes are allocated contiguously ⇒ 2nd-level nodes form a contiguous array indexed by page number

How does this work?

An example routine

```
  mfc0 k1, C0_CONTEXT
  mfc0 k0, C0_EPC  # mfc0 delay
  lw k1, 0(k1)  # slot
  # fault (k1 = orig EPC)
  noc
  mtc0 k1, C0_EFI
  tlbwr
  jr k0
  rfe
```
Virtual Linear Array Operation

- Index into 2nd level page table without referring to root PT
- Simply use the full page number as the PT index!
- Leave unused parts of PT unmapped!
- If access is attempted to unmapped part of PT, a secondary page fault is triggered
  - This will load the mapping for the PT from the root PT
  - Root PT is kept in physical memory (cannot trigger page faults)

Virtual Linear Array Page Table

- Use Context register to simply load PTE by indexing a PTE array in virtual memory
- Occasionally, will get double faults
  - A TLB miss, while servicing a TLB miss
  - Handled by general exception handler

c0 Context Register

- c0_Context = PTEBase + 4 * PageNumber
  - PTEs are 4 bytes
  - PTEBase is the base local of the page table array (note: aligned on 4 MB boundary)
  - PTEBase is (re)initialised by the OS whenever the page table array is changed
    - E.g. on a context switch
  - After an exception, c0_Context contains the address of the PTE required to refill the TLB.

Code for VLA TLB refill handler

- Load PTE address from context register
- Move the PTE into EntryLo
- Write EntryLo into random TLB entry
- Load PTE
  - Note: this load can cause a TLB refill miss itself, but this miss is handled by the general exception vector.
  - The general exception vector has to understand this situation and deal with it appropriately

Software-loaded TLB

- Pros
  - Can simplify hardware design
  - Provide greater flexibility in page table structure
- Cons
  - Typically have slower refill times than hardware managed TLBs.
**Design Tradeoffs for Software-Managed TLBs**

David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge & Richard Brown

ISCA '93 Proceedings of the 20th annual international symposium on computer architecture

---

**Trends at the time**

- Operating systems
  - Moving functionality into user processes
  - Making greater use of virtual memory for mapping data structures held within the kernel.
- RAM is increasing
  - TLB capacity is relatively static

**Statement:**
- Trends place greater stress upon the TLB by increasing miss rates and hence, decreasing overall system performance.
- True/False? How to evaluate?

---

**Software Trap on TLB Miss**

- Figure 1: Tapeworm
- The Tapeworm TLB simulator is built into the operating system and is invoked whenever there is a real TLB miss. The simulator uses the real TLB misses to simulate its own TLB configurations. Because the simulator resides in the operating system, Tapeworm captures the dynamic nature of the system and avoids the problems associated with simulators driven by static traces.

---

**Table 3: Costs for Different TLB Miss Types**

<table>
<thead>
<tr>
<th>TLB Miss Type</th>
<th>Ulrix</th>
<th>OS/91</th>
<th>Mach 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1U</td>
<td>114</td>
<td>121</td>
<td>114</td>
</tr>
<tr>
<td>L2</td>
<td>494</td>
<td>511</td>
<td>497</td>
</tr>
<tr>
<td>L3</td>
<td>354</td>
<td>386</td>
<td>385</td>
</tr>
<tr>
<td>Not-Hit</td>
<td>575</td>
<td>430</td>
<td>466</td>
</tr>
<tr>
<td>Miss</td>
<td>535</td>
<td>590</td>
<td>535</td>
</tr>
</tbody>
</table>

---

**Note the TLB miss costs**

- What is expected to be the common case?
Specialising the L2/L1K miss vector

<table>
<thead>
<tr>
<th>Type of PTE Miss</th>
<th>Count</th>
<th>Previous Total Cost (ms)</th>
<th>New Total Cost (ms)</th>
<th>Time Saved (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach 3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1D</td>
<td>30,123,212</td>
<td>30.12</td>
<td>30.12</td>
<td>0.00</td>
</tr>
<tr>
<td>L1K</td>
<td>30,095,894</td>
<td>6.09</td>
<td>6.09</td>
<td>0.00</td>
</tr>
<tr>
<td>L1K</td>
<td>2,493,250</td>
<td>43.04</td>
<td>2.89</td>
<td>40.15</td>
</tr>
<tr>
<td>L3</td>
<td>880,441</td>
<td>11.85</td>
<td>11.85</td>
<td>0.00</td>
</tr>
<tr>
<td>Invalid</td>
<td>127,245</td>
<td>3.81</td>
<td>3.81</td>
<td>0.00</td>
</tr>
<tr>
<td>Invalid</td>
<td>119,429</td>
<td>2.70</td>
<td>2.70</td>
<td>0.00</td>
</tr>
<tr>
<td>Total</td>
<td>39,052,019</td>
<td>46.25</td>
<td>46.25</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 7: Recomputing Cost of TLB Misses Given Additional Miss Vectors (Mach 3.2)

Other performance improvements?

- In Paper
  - Pinned slots
  - Increased TLB size
  - TLB associativity
- Other options
  - Bigger page sizes
  - Multiple page sizes
Itanium Page Table

- Takes a bet each way
- Loading
  - software
  - two different format hardware walkers
- Page table
  - software defined
  - Virtual linear array
  - Hashed

what about the P4?

- i.e. 32-bit x86 architecture.

P4

- Sophisticated, supports:
  - demand paging
  - pure segmentation
  - segmentation with paging
- Heart of the VM architecture
  - Local Descriptor Table (LDT)
  - Global Descriptor Table (GDT)
- LDT
  - 1 per process
  - describes segments local to each process (code, stack, data, etc.)
- GDT
  - shared by all programs
  - describes system segments (including OS itself)

P4

- To access a segment P4
  - loads a selector in 1 of the segment registers

P4

- a P4 selector:
• a P4 selector:

- when selector is in register, corresponding segment descriptor is
  - fetched by MMU
  - loaded in internal MMU registers
- Next, segment descriptor is used to handle memory reference (discussed later)

- finds a a P4 code segment descriptor

- calculating a linear address from selector+offset

- every process has page directory
  - 1024 32-bit entries
  - each entry points to page table
  - page table contains 1024 32-bit entries
  - each entry points to page frame

IF no paging used: we are done

⇒ this is the physical address

ELSE

⇒ linear address interpreted as virtual address
⇒ paging again!
P4

• Many OSs:
  – BASE=0
  – LIMIT=MAX

• ➔ no segmentation at all

That is it!