Today’s Topics

- How to reduce the size of page tables?
- How to reduce the time for address translation?
Managing page tables

Space overhead of page tables
- The size of the page table for a 32-bit address space with 4KB pages = 4MB (per process)

How can we reduce this overhead?
- Only need to map the portion of the address space actually being used (tiny fraction of entire address space)

How do we only map what is being used?
- Make the page table structure dynamically extensible
- Use another level of indirection
  - Two-level, hierarchical, hashed, etc.
Two-level Page Tables
Two-level Page Tables

- Two-level page tables
  - Virtual addresses have 3 parts
    - Master page table
      - Master page number $\rightarrow$ secondary page table
    - Secondary page table
      - Secondary page number $\rightarrow$ page frame number

<table>
<thead>
<tr>
<th>Master page #</th>
<th>Secondary page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Logical address</th>
<th>Master page table</th>
<th>Secondary page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- outer page table
- page of page table
Two-level Page Tables

- Example
- 32-bit address space, 4KB pages, 4 bytes/PTE
- Want master page table in one page

<table>
<thead>
<tr>
<th>Master page #</th>
<th>Secondary page #</th>
<th>Offset</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Physical address:
- Page frame N
- Page frame 6
- Page frame 5
- Page frame 4
- Page frame 3
- Page frame 2
- Page frame 1
- Page frame 0
Multi-level Page Tables

- Address translation in Alpha AXP Architecture
  - Three-level page tables
  - 64-bit address divided into 3 segments (coded in bits 63/62)
    - seg0 (0x): user code
    - seg1 (11): user stack
    - kseg (10): kernel

- Alpha 21064
  - Page size: 8KB
  - Virtual address: 43bits
  - Each page table is one page long
Hashed Page Tables

- Hashed page tables
  - When the address space is larger than 32 bits
  - Virtual page number is hashed into the hash table
  - Each hash table entry contains a linked list of elements that hash to the same location

- Each elements contains
  - The virtual page number
  - The value of the mapped page frame
  - A pointer to the next element in the linked list
Hashed Page Tables

Example

Following is a diagram of how a hashed page table works: Given a logical address (p, d), the hash function maps it to a hash table. The hash table then maps to a physical address (r, d), which corresponds to a location in physical memory.
Hashed Page Tables

- Clustered page tables
  - A variant of hash page tables with the difference that each entry stores mapping information for a block of consecutive page tables.
Inverted Page Tables

- Inverted page tables
  - One entry for each real page of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
  - Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
  - Use hash table to limit the search to one, or at most a few, page-table entries
Inverted Page Tables

- **Example**

![Diagram of Inverted Page Tables]

- CPU → Logical address (pid, p, d) → Page table → Physical address → Physical memory

- Process ID (pid) and page (p) for memory access.
Paging Page Tables

- **Addressing page tables**
  - Where are page tables stored? (and which address space?)
  - Physical memory
    - Easy to address, no translation required
    - But, allocated page tables consume memory for lifetime of VAS
  - Virtual memory (OS virtual address space)
    - Cold (unused) page table pages can be paged out to disk
    - But, addressing page tables requires translation
    - Do not page the outer page table (called wiring)
- Now we’ve paged the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (e.g., interrupt and exception handlers)
Original page table scheme doubled the cost of memory lookups
- One lookup into the page table, another to fetch the data

Two-level page tables triple the cost!
- Two lookups into the page tables, a third to fetch the data
- And this assumes the page table is in memory

How can we make this more efficient?
- Goal: make fetching from a virtual address about as efficient as fetching from a physical address

Solutions
- Cache the virtual-to-physical translation in hardware
- Translation Lookaside Buffer (TLB)
- TLB managed by the Memory Management Unit (MMU)
TLBs

- Translate virtual page #s into PTEs (not physical address)
- Can be done in a single machine cycle

<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>
TLBs

- TLB is implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE+offset, MMU can directly calculate the physical address

- TLBs exploit locality
  - Processes only use a handful of pages at a time
    - 16-48 entries in TLB is typical (64-192KB)
    - Can hold the “hot set” or “working set” of process
  - Hit rates are therefore really important
Address translation with TLB
Handling TLB Misses

- Address translations are mostly handled by the TLB
  - > 99% of translations, but there are TLB misses occasionally
  - In case of a miss, who places translations into the TLB?

- Hardware (MMU): Intel x86
  - Knows where page tables are in memory
  - OS maintains tables, HW access them directly
  - Page tables have to be in hardware-defined format

- Software loaded TLB (OS)
  - TLB miss faults to OS, OS finds right PTE and loads TLB
  - Must be fast (but, 20-200 cycles typically)
  - CPU ISA has instructions for TLB manipulation
  - Page tables can be in any format convenient for OS (flexible)
Managing TLBs

- OS ensures that TLB and page tables are consistent
  - When OS changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB

- Reload TLB on a process context switch
  - Remember, each process typically has its own page tables
  - Need to invalidate all the entries in TLB. (flush TLB)
  - In IA-32, TLB is flushed automatically when the contents of CR3 (page directory base register) is changed
  - (cf.) Alternatively, we can store the PID as part of the TLB entry, but this is expensive

- When the TLB misses, and a new PTE is loaded, a cached PTE must be evicted
  - Choosing a victim PTE called the “TLB replacement policy”
  - Implemented in hardware, usually simple (e.g., LRU)
Situation

Process is executing on the CPU, and it issues a read to a (virtual) address.
Memory Reference

- The common case
  - The read goes to the TLB in the MMU
  - TLB does a lookup using the page number of the address
  - The page number matches, returning a PTE
  - TLB validates that the PTE protection allows reads
  - PTE specifies which physical frame holds the page
  - MMU combines the physical frame and offset into a physical address
  - MMU then reads from that physical address, returns value to CPU
TLB misses: two possibilities

(1) MMU loads PTE from page table in memory
- Hardware managed TLB, OS not involved in this step
- OS has already set up the page tables so that the hardware can access it directly

(2) Trap to the OS
- Software managed TLB, OS intervenes at this point
- OS does lookup in page tables, loads PTE into TLB
- OS returns from exception, TLB continues

At this point, there is a valid PTE for the address in the TLB
Memory Reference

- **TLB misses**
  - Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
    - Assuming page tables are in OS virtual address space
    - Not a problem if tables are in physical memory
  - When TLB has PTE, it restarts translation
    - Common case is that the PTE refers to a valid page in memory
    - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
Memory Reference

- Page faults
  - PTE can indicate a protection fault
    - Read/Write/Execute – operation not permitted on page
    - Invalid – virtual page not allocated, or page not in physical memory
  - TLB traps to the OS (software takes over)
    - Read/Write/Execute – OS usually will send fault back to the process, or might be playing tricks (e.g., copy on write, mapped files)
    - Invalid (Not allocated) – OS sends fault to the process (e.g., segmentation fault)
    - Invalid (Not in physical memory) – OS allocates a frame, reads from disk, and maps PTE to physical frame