Page Table Implementation
Readings

- Silbershatz et al: 8.4-8.6
Outline

- Translation Lookaside buffer (TLB)
- Protection and Shared Pages
- Dealing with Large Page Tables
TLB
Implementation of Page Table

- The simplest approach is to have the page table implemented as a set of dedicated registers

- Note: Not feasible to keep page table in registers
  - Why? Page tables can be very large
  - Context switching would be very expensive
Implementation of Page Table

- Each process has a page table
- Page table is kept in main memory
- **Page-table base register** (PTBR) points to the page table
- **Page-table length register** (PRLR) indicates size of the page table
- Changing page tables requires changing only this one register during a context switch
Implementation of Page Table

Problem:

- Want to access location $i$
- Two memory accesses:
  - Index into the page table using the value in the PTBR offset by the location $i$. This gives the frame number
  - Use the frame number to access the desired address

Solution: Use a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
TLB

- Associative memory - parallel search

- Address translation \((p, d)\)
  - If \(p\) is in associative memory, get frame number out
  - Otherwise get frame number from page table in memory
Paging Hardware With TLB
The TLB contains only a few of the page-table entries.

When a logical address is generated by the CPU its page number is presented to the TLB.

If found the frame number is immediately available.
TLB

- If page number is not in TLB then a TLB miss occurs
  - The page table is consulted
  - The page number and frame number is added to the TLB
  - If the TLB is full then one of the entries is replaced
  - Example replacement policy:
    - Least Recently Used (LRU)

- A high hit rate has a high impact can dramatically reduce lookup time
Effective Access Time

- TLB Lookup = $\varepsilon$ time unit
- Memory cycle is 1
- Hit ratio = $\alpha$
  - If hit the time to get page is $1 + \varepsilon$
- Miss ratio = $1 - \alpha$
  - If miss the time to get page is $2 + \varepsilon$
- Effective Access Time (EAT)
  \[
  \text{EAT} = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\
  = 2 + \varepsilon - \alpha
  \]
Effective Access Time

- TLB Lookup = 20 nanoseconds
- Hit ratio (percentage of times that a particular page is found in the TLB) = 80%
- Time to access memory: 100 nanoseconds

- TLB hit:
  - Time to get data: 120

- TLB miss:
  - Time to get data: 220

- Effective access time:
  - $0.80 \times 120 + 0.2 \times 220 = 140$
Effective Access Time

- Assume hit ratio is 98% (typical)
- Effective access time:
  - $0.98 \times 120 + 0.2 \times 220 = 122$
Protection and Shared Pages
Protection

- Memory protection implemented by associating protection bit with each frame
- One protection bit can define a page to be read-write or read-only
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.

- **Private code and data**
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.
Shared Pages Example
Dealing with Large Page Tables
Structure of the Page Table

- Typically systems have large logical address spaces
- Page table could be excessively large

Example:
- 32-bit logical address space
  - The number of possible addresses in the logical address space is $2^{32}$
- Page size is 4 KB (4096 bytes or $2^{12}$)
- Number of pages is $2^{20}$ (20 bits for page number)
- Page table may consist of up to 1 million entries

Take away message: Page table could be large
Structure of Page Table

- Several approaches
  - MultiLevel Page tables
  - Hashed Page Tables
  - Inverted Page Tables
Two-Level Page-Table Scheme

- Page table is also paged
- Need to be able to index the outer page table
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - A page number consisting of 22 bits
  - A page offset consisting of 10 bits

- Since the page table is paged, the page number is further divided into:
  - A 12-bit page number
  - A 10-bit page offset
Two-Level Paging Example

Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table.
Address-Translation Scheme

- $p_1$ is used to index the outer page table
- $p_2$ is used to index the page table
Multi-level Paging

- What if you have a 64-bit architecture?
- Do you think hierarchical paging is a good idea?
  - How many levels of paging are needed?
  - What is the relationship between paging and memory accesses?
  - 64-bit means that even the outer page is large
    - The 64-bit UltraSPARC requires 7 levels of paging
Hashed Page Tables

- Common in address spaces larger than 32 bits
- The page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
Examples

- Intel 64 bit architecture
  - 48-bit virtual address with support for page sizes of 4KB, 2 MB, or 1GB using 4 levels of paging hierarchy

- ARM
  - 32-bit
  - Page sizes: 4-KB, 16-KB pages
  - Two levels of TLB

- Oracle Sparc
  - 64 bit
  - Hash table
    - Kernel and user processes
Summary

- This section studied implementation issues related to hash tables