Overview

Objective:
• To discuss how paging works in contemporary computer systems.

• Paging
• Structure of Page Table
Recap

• Contiguous Memory Allocation
  • Hardware Support, Multiple-partition Allocation, Fragmentation

• Segmentation
  • User’s view of a program, segmentation architecture.
Questions

1. What is fragmentation? What are its different types? (Easy)

2. What is segmentation? (Easy)

3. What is segment table? What does it contain? (Easy)
Overview

• Paging
• Structure of Page Table
Overview

• Paging

• Structure of Page Table
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames
- To run a program of size $N$ pages, need to find $N$ free frames and load program
- Set up a page table to translate logical to physical addresses
Paging: Address Translation Scheme

• Address generated by CPU is divided into:
  • Page number \((p)\) – used as an index into a page table which contains base address of each page in physical memory
  • Page offset \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit

\[
\begin{array}{c|c}
\text{page number} & \text{page offset} \\
p & d \\
m - n & n \\
\end{array}
\]

• For given logical address space \(2^m\) and page size \(2^n\)
Paging : Paging Hardware
Paging:

Paging Model of Logical and Physical Memory

logical memory

page 0
page 1
page 2
page 3

page table

frame number

0
1
2
3

0 1
1 4
2 3
3 7

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

physical memory
Paging Example

$n=2$ and $m=4$

32-byte memory and 4-byte pages
Paging: Free Frames

Before allocation

After allocation
Paging:
Implementation of Page Table

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called
  associative memory or translation look-aside buffers (TLBs)
Paging:
Translation Look-aside Buffer

• TLB (aka associative register) – parallel search

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Address translation \((p, d)\)
  • If \(p\) is in TLB, get frame # out
  • Otherwise get frame # from page table in memory
Paging:
Paging Hardware with TLB
Paging: Shared Pages Example

• Shared code
  • One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  • Useful for inter-process communication if sharing of read-write pages is allowed
Paging: Shared Pages Example
Structure of the Page Table

• Memory structures for paging can get huge using straight-forward methods
  • Consider a 32-bit logical address space as on modern computers
  • Page size of 4 KB ($2^{12}$)
  • Page table would have 1 million entries ($2^{32} / 2^{12}$)
  • If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    • That amount of memory used to cost a lot
    • Don’t want to allocate that contiguously in main memory

• Solutions:
  • Hierarchical Paging
  • Hashed Page Tables
  • Inverted Page Tables
Structure of the Page Table: Hierarchical Page Table

• Break up the logical address space into multiple page tables
• A simple technique is a two-level page table
• We then page the page table
Structure of the Page Table:
Two-level Page-Table Scheme
Structure of the 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

• 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>
<tr>
<td>10</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 inner page table
Structure of the Page Table: Address-Translation Scheme

logical address

\[ p_1 \quad p_2 \quad d \]

outer page table

\[ p_1 \]

page of page table

\[ p_2 \]

d
Structure of the Page Table: 64-bit Logical Address Space

• Even two-level paging scheme not sufficient

• If page size is 4 KB ($2^{12}$)
  • Then page table has $2^{52}$ entries
  • If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  • Address would look like

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

• Outer page table has $2^{42}$ entries
• One solution is to add a 2$^\text{nd}$ outer page table
• But in the following example the 2$^\text{nd}$ outer page table has still $2^{32}$ entries
  • And possibly 4 memory access to get to one physical memory location
Structure of the Page Table: Three-level Paging Scheme

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$d$</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Structure of the Page Table: Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
Structure of the Page Table: Hashed Page Tables
Structure of the Page Table: Inverted Page Table

• Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages

• One entry for each real page of memory

• Entry consists of the virtual address of the page, 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
  • TLB can accelerate access
Structure of the Page Table: Inverted Page Table Architecture
Credits for slides

Silberschatz, Galvin and Gagne