Address Translation

Sarah Diesburg
Operating Systems
CS 3430
Recall from Last Time...

- Translation tables are implemented in HW, controlled by SW
This Lecture...

- Different translation schemes
  - Base-and-bound translation
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Hashed page tables
  - Inverted page tables
Assumptions

• 32-bit machines
• 1-GB RAM max

\[ \log_2(1\text{GB}) = 30 \text{ bits for 1GB of RAM} \]

1GB RAM = \(2^{30}\)
Base-and-Bound Translation

- Each process is loaded into a contiguous region of physical memory
- Processes are protected from one another
Base-and-Bound Translation

- Each process “thinks” that it is running on its own dedicated machine, with memory addresses from 0 to bound.
Base-and-Bound Translation

• An OS can move a process around
  – By copying bits
  – Changing the base and bound registers
Pros and Cons of Base-and-Bound Translation

+ Simplicity
+ Speed
- **External fragmentation**: memory is wasted because the available memory is not contiguous for allocation
- Difficult to share programs
  – Each instance of a program needs to have a copy of the code segment
Pros and Cons of Base-and-Bound Translation

- Memory allocation is complex
  – Need to find contiguous chunks of free memory
  – Reorganization involves copying

- Does not work well when address spaces grow and shrink dynamically
Segmentation

- **Segment**: a region of logically contiguous memory
- **Segmentation-based translation**: use a table of base-and-bound pairs
Segmentation Diagram

30 bits

Physical seg base  Seg bound

0

1

2

Physical seg base  Seg bound

Virt seg #  Offset

32 - 30 = 2 bits for 32-bit machines

up to 30 bits

0

1

2

up to 30 bits

Phy addr

Error

log_2(1GB) = 30 bits for 1GB of RAM
### Segmentation Diagram

<table>
<thead>
<tr>
<th></th>
<th>code</th>
<th>data</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000</td>
<td>0x0</td>
<td>0x2000</td>
</tr>
<tr>
<td>1</td>
<td>0x700</td>
<td>0x500</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

- **2**: 0x200
- **>**: 0x200
- **+**: 0x200
- **0x2200**: Result
Segmentation Translation

- virtual_address = virtual_segment_number:offset
- physical_base_address = segment_table[virtual_segment_number]
- physical_address = physical_base_address + offset
Pros and Cons of Segmentation

+ Easier to grow and shrink individual segments
+ Finer control of segment accesses
  - e.g., read-only for shared code segment
+ More efficient use of physical space
+ Multiple processes can share the same code segment

- Memory allocation is still complex
  - Requires contiguous allocation
Paging

- **Paging-based translation**: memory allocation via fixed-size chunks of memory, or *pages*
  - Translation granularity is a page
- The memory manager uses a *bitmap* to track the allocation status of memory pages
  - Array of bits (0 or 1) to signify free or used pages
Paging Diagram

32 – 12 = 20 bits for 32-bit machines

$log_2(4\text{KB}) = 12$ bits for 4-KB pages

$log_2(1\text{GB}) = 30$ bits for 1GB of RAM

Virtual page number

Offset

Page table size - 1

Physical page number

Physical page number

Physical page number

Physical page number

Offset

$log_2(1\text{GB}) = 30$ bits for 1GB of RAM

Error
Paging Example
Paging Translation

• virtual_address = virtual_page_number:offset
• physical_page_number = page_table[virtual_page_number]
• physical_address = physical_page_number:offset
Pros and Cons of Paging

+ Easier memory allocation
+ Allows code sharing

- **Internal fragmentation**: allocated pages are not fully used

- The page table size can potentially be very large
  
  – 32-bit architecture with 1-KB pages can require 4 million table entries
Multi-Level Translation

- **Segmented-paging translation**: breaks the page table into segments
- **Paged page tables**: Two-level tree of page tables
Segmented Paging

1. Start with a virtual memory address
2. Look up the page table address in the segment table
3. Index into the page table to get the physical page number
4. Concatenate the physical page number to the offset
Segmented Paging

30 bits for 1-GB RAM \[32 - 3 - 12 = 17\] bits

<table>
<thead>
<tr>
<th>Seg #</th>
<th>Virt page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

Page table base | Page table bound
Page table base | Page table bound
Page table base | Page table bound

\[\log_2(6\text{ segments}) = 3\] bits
12 bits for 4-KB pages

<table>
<thead>
<tr>
<th>Phy page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phy page #</td>
</tr>
<tr>
<td>Phy page #</td>
</tr>
</tbody>
</table>

18 bits
num of entries defined by bound; up to \(2^{17}\) entries
Segmented Paging (Cont.)

32 – 3 – 12 = 17 bits

Page table base + Seg # Virt page # Offset

Phy page # Phy page # Phy page #

\[ \log_2(1\text{GB}) = 30 \text{ bits for 1GB of RAM} \]

Page table bound-1

Error

2^{17}
Segmented Paging Translation

- virtual_address = segment_number:page_number:offset
- page_table (base address) = segment_table[segment_number]
- physical_page_number = page_table[virtual_page_number]
- physical_address = physical_page_number:offset
Pros and Cons of Segmented Paging

+ Code sharing
+ Reduced memory requirements for page tables
- Higher overhead and complexity
- Page tables still need to be contiguous
- Each memory reference now takes two lookups
Paged Page Tables

1. Start with a virtual memory address
2. Index into the first page table to find the address to the second page table
3. Index into the second page table to find the physical page number
4. Concatenate the physical page number to the offset
Paged Page Tables

Page table num | Virt page num | Offset
---|---|---

Page table address (30 bits)

$2^{12}$ entries

Page table address

Page table address

$2^{8}$ entries

Phy page num

Phy page num

Phy page num

Phy page num (18 bits) | Offset
---|---

12 bits
12 bits for 4-KB pages
Paged Page Table Translation

- `virtual_address = outer_page_num:inner_page_num:offset`
- `page_table = outer_page_table[outer_page_num]`
- `physical_page_num = inner_page_table[inner_page_num]`
- `physical_address = physical_page_num:offset`
Pros and Cons of Paged Page Tables

+ Can be generalized into multi-level paging

- Multiple memory lookups are required to translate a virtual address
  – Can be accelerated with *translation lookaside buffers* (TLBs)
    • Stores recently translated memory addresses for short-term reuses
Hashed Page Tables

- Physical_address
  \[= \text{hash(virtual\_page\_num)}:\text{offset}\]

+ Conceptually simple

- Need to handle collisions

- Need one hash table per address space
Inverted Page Table

- One hash entry per physical page
- physical_address
  - $= \text{hash}(\text{pid}, \text{virtual\_page\_num}):\text{offset}$

+ The number of page table entries is proportional to the size of physical RAM

- Collision handling