Paging
Review

Virtual addresses

Physical addresses

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bounds</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x2000</td>
<td>0x6ff</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xffff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

External Fragmentation

Paging

VPN to PFN Translation
Virtual => Physical PAGE Mapping

How should OS translate VPN to PPN?
For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)
For paging, OS needs more general mapping mechanism
What data structure is good?

Big array: page table
The Mapping
Let’s fill in the Page Table

Page Tables:
Where are page tables stored?

Ideally, put it in fast hardware (MMU)…

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = $2^{\text{bits for vpn}}$
- Bits for vpn = 32– number of bits for page offset
  $= 32 – \log(4\text{KB}) = 32 – 12 = 20$
- Num entries = $2^{20} = 1 \text{ MB}$
- Page table size = Num entries * 4 bytes = 4 MB per process
Where are page tables stored?

Implication: Store each page table in memory
• Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?
• Change contents of page table base register to newly scheduled process
• Save old page table base register in PCB of descheduled process
Other PT info

What other info is in pagetable entries besides translation?
- valid bit
- protection bits
- present bit (needed later)
- reference bit (needed later)
- dirty bit (needed later)

Page table entries are just bits stored in memory
- Agreement between hardware and OS about interpretation
Memory Accesses with Pages

0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100

Assume PT is at phys addr 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Earlier: How many mem refs with segmentation?

5 (3 instrs, 2 movl)

Physical Memory Accesses with Paging?

1) Fetch instruction at logical addr 0x0010; vpn?
   • Access page table to get ppn for vpn 0
   • Mem ref 1: 0x5000
   • Learn vpn 0 is at ppn 2
   • Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from logical addr 0x1100; vpn?
   • Access page table to get ppn for vpn 1
   • Mem ref 3: 0x5004
   • Learn vpn 1 is at ppn 0
   • Movl from 0x0100 into reg (Mem ref 4)

Use of a page table doubles memory references
Advantages of Paging

No external fragmentation
- Any page can be placed in any frame in physical memory

Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn’t have to coalesce with adjacent free space
- Just use bitmap to show free/allocated page frames

Simple to swap-out portions of memory to disk (later lecture)
- Page size matches disk block size
- Can run process when some pages are on disk
- Add “present” bit to PTE
Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process
• Wasted memory grows with larger pages

Additional memory references → time-inefficient!
• Page table must be stored in memory
• MMU stores only base address of page table

Storage for page tables substantial → space-inefficient!
• Simple page table: Requires PTE for all pages in address space
  • Naively, page table entry needed even if page not allocated
• Problematic with dynamic stack and heap within address space
• Page tables must be allocated contiguously in memory
  • Due to linear access of page table entries
Reducing Page Table sizes
How big are page tables?

1. PTE’s are **2 bytes**, and **32** possible virtual page numbers

   \[32 \times 2 \text{ bytes} = 64 \text{ bytes}\]

2. PTE’s are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

   \[2 \text{ bytes} \times 2^{(24 - \lg 16)} = 2^{21} \text{ bytes (2 MB)}\]

3. PTE’s are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**

   \[4 \text{ bytes} \times 2^{(32 - \lg 4K)} = 2^{22} \text{ bytes (2 MB)}\]

4. PTE’s are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

   \[4 \text{ bytes} \times 2^{(64 - \lg 4K)} = 2^{54} \text{ bytes}\]

How big is each page table?
Why ARE Page Tables so Large?
Many invalid page table entries

Format of linear page tables:

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

...many more invalid...

how to avoid storing these?
Avoid the simple linear page table

Use more efficient (but complex) data structures, instead of the simple big array

Any data structure is possible in principle*

*assuming software managed TLB
Some approaches

1. Inverted Pagetables
2. Segmented Pagetables
3. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
Approach 1: Inverted Page Table

Inverted Page Tables
• Only need entries for virtual pages w/ valid physical mappings

Naïve approach:
Search through data structure <ppn, vpn+ASID> to find match
• Too much time to search entire table

Better: Find possible matches entries by hashing vpn+ASID
• Smaller number of entries to search for exact match
Valid PTEs are Contiguous

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>...many more invalid...</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
</tr>
</tbody>
</table>

Note “hole” in addr space: valids vs. invalids are clustered

How did OS avoid allocating holes in phys memory?

Use ideas from segmentation!

how to avoid storing these?
Approach 2: Segmented Page Tables

Divide address space into segments (code, heap, stack)
  • Segments can be variable length

Divide each segment into fixed-sized pages

Logical address divided into three portions

<table>
<thead>
<tr>
<th>seg #  (4 bits)</th>
<th>page number (8 bits)</th>
<th>page offset (12 bits)</th>
</tr>
</thead>
</table>

Ideas

• Each segment has a page table
• Each segment tracks the base (physical address) and bounds of the page table for that segment
## Combining Paging and Segmentation

<table>
<thead>
<tr>
<th>seg # (4 bits)</th>
<th>page number (8 bits)</th>
<th>page offset (12 bits)</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x002000</td>
<td>0xff (255)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x000000</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0x001000</td>
<td>0x0f (15)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Read/Write</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x002070</td>
<td>read</td>
<td>0x004070</td>
</tr>
<tr>
<td>0x202016</td>
<td>read</td>
<td>0x003016</td>
</tr>
<tr>
<td>0x104c84</td>
<td>read</td>
<td>error</td>
</tr>
<tr>
<td>0x010424</td>
<td>write</td>
<td>error</td>
</tr>
<tr>
<td>0x210014</td>
<td>write</td>
<td>error</td>
</tr>
<tr>
<td>0x203568</td>
<td>read</td>
<td>0x02a568</td>
</tr>
</tbody>
</table>

Page table:

```
0x001000
  ... 0x01f
  0x011
  0x003
  0x02a
  0x013
  ... 0x00c
  0x007
  0x004
  0x00b
  0x006
  ... 0x02000
```
Advantages of Segments
- Supports sparse address spaces
  - Decreases size of page tables
  - If segment not used, not needed for page table

Advantages of Pages
- No external fragmentation
- Segments can grow without any reshuffling
- Can run process when some pages are swapped to disk (next lecture)

Advantages of Both
- Increases flexibility of sharing
  - Share either single page or entire segment. How?
Disadvantages of Paging with Segmentation

Potentially large page tables (for each segment)

• Must allocate each page table contiguously
• More problematic with more address bits
• Page table size?
  • Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:
  = Number of entries * size of each entry
  = Number of pages * 4 bytes
  = $2^{18} \times 4$ bytes = $2^{20}$ bytes = 1 MB!!!
Other Approaches

1. Inverted Pagetables
2. Segmented Pagetables
3. Multi-level Pagetables
   • Page the page tables
   • Page the pages of page tables…
3) Multilevel Page Tables

Goal: Allow page tables to be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level page directory
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)
## Multilevel example

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>valid</th>
<th>VPN</th>
<th>PPN</th>
<th>valid</th>
<th>VPN</th>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x3</td>
<td>1</td>
<td>0</td>
<td>0x10</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0x23</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>0x59</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>0x80</td>
<td>1</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>0x92</td>
<td>1</td>
<td></td>
<td>-</td>
<td>0</td>
<td></td>
<td>0x55</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0x45</td>
<td>1</td>
<td></td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

### 20-bit address:

- **outer page (4 bits)**
- **inner page (4 bits)**
- **page offset (12 bits)**

- translate 0x01ABC
- 0x23ABC
- translate 0x00000
- 0x10000
- translate 0xFEED0
- 0x55ED0
Address format for Multilevel Paging

How should logical address be structured?
• How many bits for each paging level?

Goal?
• Each page table fits within a page
• PTE size * number PTE = page size
  • Assume PTE size = 4 bytes
  • Page size = 2^12 bytes = 4KB
  • number PTE per page = (2^12 bytes per page) / (4 bytes per PTE)
  • → number PTE = 2^10
• → # bits for selecting inner page = 10

Remaining bits for outer page:
• 30 – 10 – 12 = 8 bits