Address Space Management

- **Base and bounds**
  - Unit of translation and swapping is entire address space

- **Segmentation**
  - Unit of translation and swapping is each segment (few per address space)

- **Paging**
  - Unit of translation and swapping is a fixed-size page

Paging

- Translating virtual address to physical address:
  - Split address into virtual page # and offset
  - Look up physical page corresponding to virtual page
  - if (virtual page is invalid or non-resident or protected) {
    - trap to OS fault handler
  } else {
    - physical page # = pageTable[virtual page #].physPageNum
  }

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Physical page #</th>
<th>Resident</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>105</td>
<td>0</td>
<td>RX</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>283</td>
<td>1</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>Invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1048575</td>
<td>Invalid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valid versus Resident

- **Valid** → virtual page is legal for process to access
- **Resident** → virtual page is in physical memory
- Error to access invalid page, but not to access non-resident page

- Who makes a virtual page valid/invalid?
- Who makes a virtual page resident/non-resident?
- Why would a process want one of its virtual pages to be invalid?

Picking Page Size

- What happens if page size is really small?
- What happens if page size is really big?

- Typically a compromise, e.g., 4 KB or 8 KB
  - Some architectures support multiple page sizes

Growing Address Space

Doesn’t this waste space like base and bounds?

<table>
<thead>
<tr>
<th>Stack</th>
<th>Virtual page #</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Invalid</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
<th>Virtual page #</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1048572</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>1048573</td>
<td>1078</td>
<td></td>
</tr>
<tr>
<td>1048574</td>
<td>48136</td>
<td></td>
</tr>
<tr>
<td>1048575</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Paging

- **Pros?**
  - Simple memory allocation
  - Flexible sharing
  - Easy to grow address space

- **Cons?**
  - Large page tables

- How to modify paging to reduce space needed for translation data?
Multi-level Paging

- Standard page table is a simple array
- Multi-level paging generalizes this into a tree
- Example: Two-level page table with 4KB pages
  - Index into level 1 page table: virtual address bits 31-22
  - Index into level 2 page table: virtual address bits 21-12
  - Page offset: bits 11-0

How does this let translation data take less space?

Sparse Address Space

- Stack
- Heap
- Code

Translation lookaside buffer

- TLB caches virtual page # to PTE mapping
  - Cache hit \(\rightarrow\) Skip all the translation steps
  - Cache miss \(\rightarrow\) Get PTE, store in TLB, restart instruction

- Does this change what happens on a context switch?
**Administrivia**

- Fill out peer evaluations
- Keep track of interesting bugs for interviews
- Project 3 posted: Due on Nov. 12th
- Midterm review: 7-9pm on Sunday in 1013 Dow

**Page Replacement**

- Not all valid pages may fit in physical memory
  - Some pages are swapped out to disk
- To read in a page from disk, some resident page must be swapped out to disk
- Which page to evict when you need a free page?
  - Goal: minimize page faults

**Replacement policies**

- Random
- FIFO
  - Replace page brought into memory longest time ago
  - May replace pages that continue to be frequently used
- Optimal
  - Replace page that won’t be used for the longest time in the future
  - Minimizes misses, but requires knowledge of the future

**Replacement policies**

- LRU (least recently used)
  - Approximates OPT by using past reference pattern
    » If page hasn’t been used for a while, it probably won’t be used for a long time in the future
  - Why would this work well?
    » Temporal locality: Recently accessed pages are often accessed again
- LRU is hard to implement exactly
  - Can we simplify LRU by approximating it?

**Clock replacement algorithm**

- Most MMUs maintain a "referenced" bit for each resident page
  - Set by MMU when page is read or written
  - Can be cleared by OS
- How to use reference bit to identify old pages?
- Why maintain reference bit in hardware?
- How do we do work incrementally, rather than all at once?