Paging

Allocate VA & PA memory in fixed-sized chunks (pages and frames, respectively)
- free frames can be tracked using a simple bitmap
  - 0011111001111011110000 one bit/frame
- no more external fragmentation!
- but now internal fragmentation (you just can’t win…)
  - when memory needs are not a multiple of a page
  - typical size of page/frame: 4KB to 16KB

Adjacent pages in VA (say, within the stack) need not map to contiguous frames in PA!

Virtual address

Interpret VA as comprised of two components
- page: which page?
- offset: which byte within that page?

Virtual address

Interpret VA as comprised of two components
- page: which page?
  - no. of bits specifies no. of pages are in the VA space
- offset: which byte within that page?

Virtual address

Interpret VA as comprised of two components
- page: which page?
  - no. of bits specifies size of page/frame
- offset: which byte within that page?
Virtual address

- To access a byte
  - extract page number
  - map that page number into a frame number using a page table
  - extract offset
  - access byte at offset in frame

Page Table

- Frame number
- Valid/Invalid bit
  - Set if entry stores a valid mapping, if not, page fault
- Cache modified bit
  - Set for memory-mapped I/O
- Referenced bit
  - Set if page has been referenced
- Modified bit
  - Set if page has been modified
- Protection bits (R/W/X)

Basic Paging

- The Page Table lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR saved/restored on context switch

Page Table Entries

- Frame number
- Valid/Invalid bit
- Cache modified bit
- Referenced bit
- Modified bit
- Protection bits (R/W/X)

Sharing

- By now, it's old hat:
  - Processes share pages by mapping virtual pages to the same frame
  - Fine tuning using protection bits (RWX)
- We can refine COW to operate at the granularity of pages
  - on fork, mark all pages read only
  - on write, copy only the affected page

By now, it’s old hat:

- Processes share pages by mapping virtual pages to the same frame
- Fine tuning using protection bits (RWX)
- We can refine COW to operate at the granularity of pages
- on fork, mark all pages read only
- on write, copy only the affected page
Example

Page size: 4 bytes

<table>
<thead>
<tr>
<th>VA Space</th>
<th>Page Table</th>
<th>PA Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  A  B  C  D</td>
<td>0  4</td>
<td>0</td>
</tr>
<tr>
<td>1  E  F  G  H</td>
<td>1  3</td>
<td>2</td>
</tr>
<tr>
<td>2  I  J  K  L</td>
<td>2  1</td>
<td>3</td>
</tr>
</tbody>
</table>

Space overhead

- Two sources, in tension:
  - data structure overhead (the Page Table itself)
  - fragmentation
    - How large should a page be?

- Overhead for paging:
  \[
  \text{(entries x sizeofEntry) + (segments x pageSize/2)} = \text{(VA_Size/pagesize) x sizeofEntry) + (segments x pageSize/2)}
  \]

- Size of entry
  - enough bits to identify physical page (\(\log_2(PA\_Size / \text{page size})\))
  - should include control bits (valid, modified, referenced, etc)
  - usually word or byte aligned

Computing paging overhead

- 1 MB maximum VA, 1 KB page, 3 segments (program, stack, heap)
  \[
  \left(\frac{2^{20}}{2^{10}}\right) \times \text{sizeofEntry} + (3 \times 2^9)
  \]

- If I know PA is 64 KB then sizeofEntry = 6 bits (2^6 frames) + control bits
  - if 7 control bits, byte aligned size of entry: 16 bits

What’s not to love?

- Space overhead
  - With a 64-bit address space, size of page table can be huge

- Time overhead
  - Accessing data now requires two memory accesses
    - must also access page table, to find mapped frame
Reducing the Storage Overhead of Page Tables

- Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
  - an array of $2^{52}$ entries!
- Good news
  - most space is unused
- Use a better data structure to express the Page Table
  - a tree!

Example
- 32 bit address space
- 4Kb pages
- 4 bytes PTE

Multi-level Paging

Example
- What is the page size?
Example

What is the page size? Page size is 256 bytes
What is the Page Table size for a process with a 256KB VAS starting at address 0?

The process needs to map 1024 pages (256KB/256 bytes)
To do so, it is sufficient to materialize the following tree since $1024/64 = 16$
So, assuming each PTE is 2 bytes

$1024 \times 2 + 256 \times 2 + 16 \times 64 \times 2 = 4608$ bytes