Managing Free space

- Many segments, different processes, different sizes
- OS tracks free memory blocks ("holes")
  - Initially, one big hole
- Many strategies to fit segment into free memory (think "assigning classrooms to courses")
  - First Fit: first big-enough hole
  - Next Fit: Like First Fit, but starting from where you left off
  - Best Fit: smallest big-enough hole
  - Worst Fit: largest big-enough hole

External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes

External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes
    - Copying eats up a lot of CPU time!
      - if 4 bytes in 10ns, 8 GB in 20s!
    - But what if a segment wants to grow?
Eliminating External Fragmentation: Swapping

- Preempt processes and reclaim their memory
- Move images of suspended processes to swap space on backing store

Paging

- Allocate VA & PA memory in fixed-sized chunks (pages and frames, respectively)
- Free frames can be tracked using a simple bitmap
  - 00111110011101110000 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

- 32 bits
  - Interpret VA as comprised of two components
    - Page: which page?
    - Offset: which byte within that page?

Virtual address

- p (20 bits)
  - Interpret VA as comprised of two components
    - Page: which page?
      - No. of bits specifies no. of pages in VA space
    - 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 in VA space
  - offset: which byte within that page?
    - no. of bits specifies size of page/frame

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

Basic Paging

- CPU
- Page Table
  - stores frame nos
  - lives in memory
  - at the physical address stored in the Page Table Base Register
  - PTBR saved/restored on context switch
- Physical memory

Page Table Entries

- Frame number
- Valid/Invalid bit
  - Set if process can reference that portion of VA space
- Present bit
  - Set if page is mapped to a frame
- Referenced bit
  - Set if page has been referenced
- Dirty bit
  - Set if page has been modified
- Cache disable bit
  - Set if page can’t be cached
- Protection bits (R/W/X)
Page Table Entries

- **Frame number**
- **Valid/Invalid bit**
  - Set if process can reference that portion of VA space
- **Present bit**
  - Set if page is mapped to a frame
- **Referenced bit**
  - Set if page has been referenced
- **Dirty bit**
  - Set if page has been modified
- **Cache disable bit**
  - Set if page can't be cached
- **Protection bits (R/W/X)**

```
<table>
<thead>
<tr>
<th>Frame no.</th>
<th>Valid</th>
<th>Present</th>
<th>Referenced</th>
<th>Cache Disable</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>R/W/X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R/W/X</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R/W/X</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>R/W/X</td>
</tr>
</tbody>
</table>
```

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
  - set W bit in both PTEs

Sharing

Example

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

```
<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 A B C D</td>
<td>3</td>
</tr>
<tr>
<td>1 E F G H</td>
<td>1</td>
</tr>
<tr>
<td>2 I J K L</td>
<td>2</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:

\[
(\text{#PTEs} \times \text{sizeofEntry}) + (\text{#segments} \times \text{pageSize}/2) = ((\text{VA Size}/\text{pageSize}) \times \text{sizeofEntry}) + (\text{#segments} \times \text{pageSize}/2)
\]

What makes up sizeofEntry?
- bits to identify physical page (log2 (PA_Size / frame (aka page) size))
- control bits (Valid, Present, Dirty, Referenced, etc)
- usually word or byte aligned (so, however many bits are needed to make it so)
Computing Paging Overhead

- 1 MB maximum VA, 1 KB page, 3 "segments" (program, stack, heap)
- PA space is 64KB and PTE has 7 control bits

What is the Paging Overhead?

- \((\frac{2^{20}}{2^{10}}) \times \text{sizeofEntry} + (3 \times 2^9) \) bytes
- sizeofEntry = 6 bits (2^6 frames) + 7 control bits
  - byte aligned size of PTE entry: 16 bits

\[ \text{Overhead: } 2^{10} \times 2 + 3 \times 2^9 = (2^{11} + 3 \times 2^9) \text{ bytes} \]

What’s not to love?

- **Space overhead**
  - With a 64-bit address space, size of page table can be huge
- **Time overhead**
  - What before used to require one memory access, now needs two
    - one to access the correct PTE and retrieve the correct frame number
    - one to access the actual physical address that contains the data of interest