[537] Paging

Tyler Harter
9/17/14
Overview

Review Segmentation

Paging (Chapter 18)

Break + Announcements?

Memory Allocators (Chapter 17)

Discuss P2
Review: Segmentation
Virtual Memory Approaches

Approaches (covered Monday):

- **Time Sharing**: one process uses RAM at a time
- **Static Relocation**: rewrite code before run
- **Base**: add a base to virt addr to get phys
- **Base+Bounds**: also check phys is in range
- **Segmentation**: many base+bound pairs
Virtual Memory Approaches

Approaches (covered Monday):
- **Time Sharing**: one process uses RAM at a time
- **Static Relocation**: rewrite code before run
- **Base**: add a base to virt addr to get phys
- **Base+Bounds**: also check phys is in range
- **Segmentation**: many base+bound pairs
Segmentation Example

Assume a 14-bit virtual addresses, with the high 2 bits indicating the segment.

Assume 0=>code, 1=>heap, and 2=>stack.

What virtual addresses could be valid for each segment?
Segments:
0 => code
1 => heap
2 => stack.
Segments:
0 => code
1 => heap
2 => stack.
Segments:
0 => code
1 => heap
2 => stack.
Segments:
0 => code
1 => heap
2 => stack.
Memory Accesses:

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

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses

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

Memory Accesses:
Fetch instruction at addr 0x4010

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses

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

Memory Accesses:
Fetch instruction at addr 0x4010
Exec, load from addr 0x5900

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses

Memory Accesses:
- Fetch instruction at addr 0x4010
- Exec, load from addr 0x5900

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>

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

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

Memory Accesses:
Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses

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

Memory Accesses:
Fetch instruction at addr 0x4010
Exec, load from addr 0x5900
Fetch instruction at addr 0x4013
Exec, no load

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses:

- **Fetch** instruction at addr 0x4010
- Exec, **load** from addr 0x5900
- Fetch instruction at addr 0x4013
- Exec, no load

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses:

- **Fetch** instruction at addr 0x4010
  - Exec, **load** from addr 0x5900
- **Fetch** instruction at addr 0x4013
  - Exec, no load
- **Fetch** instruction at addr 0x4019

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Memory Accesses

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

Memory Accesses:
- Fetch instruction at addr 0x4010
  Exec, load from addr 0x5900
- Fetch instruction at addr 0x4013
  Exec, no load
- Fetch instruction at addr 0x4019
  Exec, store to addr 0x5900

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>16KB (0x4000)</td>
<td>20KB (0x5000)</td>
</tr>
<tr>
<td>heap</td>
<td>22KB (0x5800)</td>
<td>26KB (0x6800)</td>
</tr>
<tr>
<td>stack</td>
<td>26KB (0x6800)</td>
<td>28KB (0x7000)</td>
</tr>
</tbody>
</table>
Segments:
0 => code
1 => heap
2 => stack.
How to grow stack?

Segments:
0 => code
1 => heap
2 => stack.

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16KB</td>
<td>20KB</td>
</tr>
<tr>
<td>1</td>
<td>22KB</td>
<td>26KB</td>
</tr>
<tr>
<td>2</td>
<td>26KB</td>
<td>28KB</td>
</tr>
</tbody>
</table>
Segments:
0 => code
1 => heap
2 => stack.
Segments:
0 => code
1 => heap
2 => stack.

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16KB</td>
<td>20KB</td>
</tr>
<tr>
<td>1</td>
<td>22KB</td>
<td>26KB</td>
</tr>
<tr>
<td>2</td>
<td>28KB</td>
<td>30KB</td>
</tr>
</tbody>
</table>
Segments:
0 => code
1 => heap
2 => stack.
Paging
Paging

Segmentation is too coarse-grained. Either waste space OR memcpy often.

We need a fine-grained alternative!

Paging idea:
- break mem into small, fix-sized chunks (aka pages)
- each virt page is independently mapped to a phys page
- grow memory segments however we please!
Addressing

For segmentation, high bits $\Rightarrow$ segment, low bits $\Rightarrow$ offset

For paging, high bits $\Rightarrow$ page, low bits $\Rightarrow$ offset

How many low bits do we need?
# Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
</tbody>
</table>
### Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
</tbody>
</table>
# Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
<tr>
<td>512 bytes</td>
<td></td>
</tr>
</tbody>
</table>
Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
</tr>
<tr>
<td>4 KB</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
### Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
</tr>
</tbody>
</table>
# Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
<th>Virt Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
# Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
<th>Virt Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
## Address Examples

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
<th>Virt Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>1 K</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td>4 K</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
<td>1 MB</td>
</tr>
</tbody>
</table>
For segmentation, we used a formula (e.g., $\text{phys} = \text{virt\_offset} + \text{base\_reg}$)

Now, we need a more general mapping mechanism.

What data structure is good?
For segmentation, we used a formula (e.g., \( \text{phys} = \text{virt\_offset} + \text{base\_reg} \))

Now, we need a more general mapping mechanism.

What data structure is good? Big array, called a **pagetable**
The Mapping

Virt Mem

Phys Mem

P1

P2

P3
The Mapping

Virt Mem

Phys Mem

P2

P3
The Mapping
The Mapping

Virt Mem

Phys Mem

P2

P3
The Mapping

Virt Mem

Phys Mem

P2

P3
Where Are Pagetable’s Stored?

How big is a typical page table?
Where Are Pagetable’s Stored?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries (or this could be less)
- \(2^{32 - \log(4\text{KB})} \times 4 = 4\text{ MB}\)
Where Are Pagetable’s Stored?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries (or this could be less)
- $2^{32 - \log(4\text{KB})} \times 4 = 4 \text{ MB}$

Store in memory.
CPU finds it via register (e.g., CR3 on x86)
Final PT example

Memory Accesses:

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

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages
Final PT example

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

Memory Accesses:
PT, load from addr 0x5000

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages
Final PT example

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

Memory Accesses:
PT, load from addr 0x5000
Fetch instruction at addr 0x2010

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages
Final PT example

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages

Memory Accesses:
- PT, load from addr 0x5000
- Fetch instruction at addr 0x2010
- PT, load from addr 0x5004

```
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
```
Final PT example

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages

Memory Accesses:
PT, load from addr 0x5000
Fetch instruction at addr 0x2010
PT, load from addr 0x5004
Exec, load from addr 0x5900

0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
**Final PT example**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010</td>
<td><code>movl 0x1100, %edi</code></td>
</tr>
<tr>
<td>0x0013</td>
<td><code>addl $0x3, %edi</code></td>
</tr>
<tr>
<td>0x0019</td>
<td><code>movl %edi, 0x1100</code></td>
</tr>
</tbody>
</table>

**PT**

| 2 | 0 | 80 | 99 |

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages

**Memory Accesses:**
- PT, load from addr 0x5000
- Fetch instruction at addr 0x2010
- PT, load from addr 0x5004
- Exec, load from addr 0x5900
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100

Final PT example

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages

Memory Accesses:
- PT, load from addr 0x5000
- Fetch instruction at addr 0x2010
- PT, load from addr 0x5004
- Exec, load from addr 0x5900
...
Final PT example

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

Memory Accesses:
- PT, load from addr 0x5000
- Fetch instruction at addr 0x2010
- PT, load from addr 0x5004
- Exec, load from addr 0x0100

...  

Our pagetable is slow!!!

Assume PT is at 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages
Other PT info

What other data should go in pagetable entries besides translation?
Other PT info

What other data should go in pagetable entries besides translation?

- valid bit
- protection bits
- present bit
- reference bit
- dirty bit
Summary

Pros?

Cons?
Summary

Pros?
- very flexible
- no external fragmentation
- no need to shuffle around data

Cons?
- expensive translation
- huge space overheads
Announcements

P1 due tonight!

Tests: sorry.

P2 released.

Office hours, 1-2pm today (room 7373).
Memory Allocators
Free-Space Management

Many systems need to manage/allocate space

1. physical space for process segments
2. virtual space for malloc calls
3. disk blocks for files
Free-Space Management

Many systems need to manage/allocate space

1. physical space for process segments
2. virtual space for malloc calls → today
3. disk blocks for files
Allocation API

void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
Allocation API

void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
Malloc/Free Basics

malloc(3072)
malloc(3072) = 15KB
Malloc/Free Basics
Malloc/Free Basics

free(15KB)
Malloc/Free Basics

free(15KB)
Malloc/Free Basics

```plaintext
free(15KB)
```

How do we know the size to free?
Odd Object Sizes

An object may not fit free space exactly: 
\textbf{split} memory 

Free areas may be adjacent: 
\textbf{coalesce} memory
Splitting

free used

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
free used
Splitting

malloc(2048)
Splitting

\[
\text{malloc}(2048) = 15\text{KB}
\]
Coalescing

free(15KB)
Coalescing

free(15KB)
Coalescing
Coalescing
Bookkeeping

Need to know size + location of free spaces
- for malloc()

Need to know size of used spaces
- for free()
Bookkeeping

Need to know size+location of free spaces
  - for malloc()

Need to know size of used spaces
  - for free()
Free List: malloc

head

addr: 0KB
len: 1KB

addr: 15KB
len: 3KB

NULL

free
used
free
used

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
Free List: malloc

malloc(1KB)
Free List: malloc

malloc(1KB) = 15KB
Free List: malloc

head → addr: 0KB
    len: 1KB → addr: 16KB
          len: 2KB → NULL

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
free  used  used  used  free  used
Start Over

head → addr: 0KB  
len: 1KB → addr: 15KB  
len: 3KB → NULL

free  used  free  used
10KB  11KB  12KB  13KB  14KB  15KB  16KB  17KB  18KB  19KB  20KB
Start Over

head → addr: 0KB len: 1KB → addr: 15KB len: 3KB → NULL

malloc(1KB)
Free List: malloc

malloc(1KB) = 17KB
Free List: malloc

- head: addr: 0KB, len: 1KB
- addr: 15KB, len: 2KB
- NULL

Address space:
- free
- used
- free
- used
- used

Address range:
- 10KB to 20KB
Free List: free

head → addr: 0KB  len: 1KB → addr: 15KB  len: 2KB → NULL

free(17KB)
Free List: free

head ▸ addr: 17KB, len: 1KB ▸ addr: 0KB, len: 1KB ▸ addr: 15KB, len: 2KB ▸ NULL

free(17KB)
Free List:

- addr: 17KB, len: 1KB
- addr: 0KB, len: 1KB
- addr: 15KB, len: 2KB

head → NULL

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
Free List: coalesce

```
head: addr 17KB, len 1KB
addr 0KB, len 1KB
addr 15KB, len 2KB
```

Diagram:
- Free list: `addr 17KB, len 1KB`
- Used block: `addr 0KB, len 1KB`
- Free block: `addr 15KB, len 2KB`

Memory map:
```
+----------+----------+----------+
| free     | used     | free     |
+----------+----------+----------+
| 10KB     | 11KB     | 12KB     | 13KB     | 14KB     | 15KB     | 16KB     | 17KB     | 18KB     | 19KB     | 20KB     |
+----------+----------+----------+
```
Free List: coalesce

- Head: addr: 15KB, len: 3KB
- Next: addr: 0KB, len: 1KB, NULL
When coalescing is even trickier

Do we ever have to coalesce multiple areas?
When coalescing is even trickier

Do we ever have to coalesce multiple areas?

free order may be arbitrary:

```cpp
free(17KB);
free(15KB);
free(16KB)
```
Double Coalesce

head → NULL

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
Double Coalesce

head → addr: 17KB
len: 1KB → NULL

used
used
free
used
Bookkeeping: free list

What fields do we need for each node in linked list?
What fields do we need for each node in linked list?

```c
struct node {
    int size;
    void *addr;
    struct node* next;
};
```
Bookkeeping: free list

What fields do we need for each node in linked list?

```c
struct node {
    int size;
    void *addr;
    struct node* next;
}
```

How do we allocate memory for new nodes?
What fields do we need for each node in linked list?

```c
struct node {
    int size;
    void *addr;
    struct node* next;
}
```

How do we allocate memory for new nodes?
- store them in free space!
Bookkeeping: free list

What fields do we need for each node in linked list?

```c
struct node {
    int size;
    void *addr;
    struct node* next;
};
```

How do we allocate memory for new nodes?
- store them in free space!
- `addr = ((void *)node + sizeof(*node)`
Free List

10KB free
11KB used
12KB used
13KB free
14KB free
15KB used
16KB free
17KB used
18KB used
19KB used
20KB free
Bookkeeping

Need to know size+location of free spaces - for malloc()

Need to know size of used spaces - for free()
Bookkeeping

Need to know **size+location** of **free spaces**
- for `malloc()`

Need to know **size of** **used spaces**
- for `free()`
Used Size

malloc(2KB)
`malloc(2KB) = ?`
Used Size

\[ \text{malloc}(3\text{KB}) = 15\text{KB} + \text{sizeof(int)} \]
Magic Numbers

Can malloc/free catch bugs for you?
- double frees?
- overflows

Add magic number to each allocated segment, if it is overwritten, there’s a bug!

What can you spell with 0 - 1 and A - F?
Magic Numbers

Can malloc/free catch bugs for you?
- double frees?
- overflows

Add magic number to each allocated segment, if it is overwritten, there’s a bug!

What can you spell with 0 - 1 and A - F?
0xDEADBEEF, 0xFEEDFACE, …
buggy overwrite

free 4KB BEEF used free 2425533 used

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
free(18KB) doesn’t see 0xBEEF, and crashes with a warning
Allocator Policy

Which free space to consider?
Of those considered, which to use?

No perfect solutions!
Where to allocate?

Workload 1
1) need: 2KB
Where to allocate?

Workload 1
1) need: 2KB
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB

Diagram: Used and free spaces in memory.
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
   (fail)
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
   (fail)
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
   (fail)

Workload 2
1) need: 2KB
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
   (fail)

Workload 2
1) need: 2KB
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB (fail)

Workload 2
1) need: 2KB
2) need: 4KB

10KB 11KB 12KB 13KB 14KB 15KB 16KB 17KB 18KB 19KB 20KB
  used  free  used  free  used
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB
   (fail)

Workload 2
1) need: 2KB
2) need: 4KB
   (fail)
Where to allocate?

Workload 1
1) need: 2KB
2) need: 3KB
3) need: 2KB (fail)

Workload 2
1) need: 2KB
2) need: 4KB (fail)

no right answer
Review: Scheduler Vocabulary

**Workload**: set of job descriptions

**Scheduler**: logic that decides when jobs run

**Metric**: measurement of scheduling quality

Scheduler “algebra”, given 2 variables, find the 3rd:

\[ f(W, S) = M \]
Allocator Vocabulary

**Workload**: series of malloc()’s and free()’s

**Allocator**: logic that gives memory to processes

**Metric**: measurement of allocation quality

Allocator “algebra”, given 2 variables, find the 3rd:

\[ f(W, A) = M \]
<table>
<thead>
<tr>
<th>Workload:</th>
<th>Allocators:</th>
<th>Metrics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td>Best fit</td>
<td>internal fragmentation</td>
</tr>
<tr>
<td>addresses</td>
<td>Worst fit</td>
<td>external fragmentation</td>
</tr>
<tr>
<td>sizes</td>
<td>First fit</td>
<td>search time</td>
</tr>
<tr>
<td></td>
<td>Next fit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Slab</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Buddy</td>
<td></td>
</tr>
</tbody>
</table>
Allocator Basics

**Workload:**
- operations
- addresses
- sizes

**Allocators:**
- Best fit
- Worst fit
- First fit
- Next fit
- Slab
- Buddy

**Metrics:**
- internal fragmentation
- external fragmentation
- search time

read more in OSTEP
malloc provides a convenient library service to programs, abstracting the raw heap

Allocation is challenging because
- there is no right answer
- we can’t use malloc ourselves
- expensive searching for ways to coalesce
P2: Visit 537 site