Fill in the check-in form on cs107a.stanford.edu!
Announcements

● **Schedule moving forward**
  ○ I’ll hold office hours tonight from 6:45pm-7:45pm in Huang
  ○ Friday OH moved TO 4:15-5:15pm
  ○ Tuesday section is cancelled, we’ll do OH in Thornton 110
  ○ I can probably do OH over the weekend, I’ll update later
  ○ Final review session is the Sunday after Week 10, details TBD

● **Clarification on Friday deadline on Ed**
  ○ Easiest thing to do is to plan to make a complete and final submission for implicit.c on Friday
  ○ Details in the Ed post, but it’s not worth working on implicit.c past Friday unless you will make significant progress on functionality afterwards
Agenda

- Debugging Heap Allocator
- Explicit List Overview
- Headers
- Free Payloads
- Blocks
- Explicit List
Debugging Heap Allocator
To recap: What should you do if you’re not passing a script?

- If you’re segfaulting, that means your heap got so corrupted that you got undefined behavior and segfaulted. The corruption should have been caught the moment it happened. To improve the chances of this, make your validate_heap stronger.
- Ensure your validate_heap calls breakpoint() upon catching an issue.
- Once validate_heap catches an issue, run test_implicit on the script (that fails earliest) in gdb.
- You should breakpoint in validate_heap. Go up stack frames (“up”) until you’re in the test harness. Check the value of “req”.
- Make a conditional breakpoint in the test harness in the loop that services each request, conditional on that value of “req”.
- Rerun, and step into the request that caused the heap to become invalid (either mymalloc, myfree, or myrealloc). This likely has a bug.
- Use dump_heap to understand the state of the heap, and step through your code checking for something not working the way it should.
Explicit List Overview
The heap is made up of an implicit list of blocks.
[Also unchanged from implicit list]  
Each [free/allocated] block consists of a header and a payload

<table>
<thead>
<tr>
<th>0x10</th>
<th>0x11</th>
<th>0x12</th>
<th>0x13</th>
<th>0x14</th>
<th>0x15</th>
<th>0x16</th>
<th>0x17</th>
<th>0x18</th>
<th>0x19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req. 1</td>
<td>Free</td>
<td>Req. 2</td>
<td>Free</td>
<td>Req. 3</td>
<td>Free</td>
<td>Req. 4</td>
<td>Free</td>
<td>Req. 5</td>
<td>Free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24 Used</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 Used</td>
<td>e</td>
</tr>
</tbody>
</table>
Each header is the bitwise combination of a size and an allocated bit.
[NEW for explicit list]
Payloads differ between free and allocated blocks

<table>
<thead>
<tr>
<th>Req. 1</th>
<th>Free</th>
<th>Req. 2</th>
<th>Free</th>
<th>Req. 3</th>
<th>Free</th>
<th>Req. 4</th>
<th>Free</th>
<th>Req. 5</th>
<th>Free</th>
</tr>
</thead>
</table>

Allocated block: payloads reserved for client, DO NOT TOUCH

![Diagram showing memory allocation]
[NEW for explicit list]
Payloads differ between free and allocated blocks

<table>
<thead>
<tr>
<th>Req. 1</th>
<th>Free</th>
<th>Req. 2</th>
<th>Free</th>
<th>Req. 3</th>
<th>Free</th>
<th>Req. 4</th>
<th>Free</th>
<th>Req. 5</th>
<th>Free</th>
</tr>
</thead>
</table>

```
list_start_ptr
```

```
NULL
```

```
NULL
```
Two ways to traverse the heap! Implicit (all blocks) and explicit (only free blocks)
Headers
Reuse from implicit list heap allocator!

- `bool is_free(header_t *header);`
- `void set_header(header_t *header, size_t size, int status);`
- `size_t get_size(header_t *header);`
Reuse from implicit list heap allocator!

- `void *header2payload(header_t *header);`
- `header_t *payload2header(void *payload);`
- `header_t *next_block(header_t *header);`

Header pointer
(same as block pointer)

Payload pointer

Next block pointer
Free Payloads
Pointers within a Block

```
struct node {
    struct node *prev;
    struct node *next;
};
```
Given a header pointer to a free block, how would you

- Set the `prev` and `next` pointers to NULL?
- Check if the next block's `prev` pointer points back to this block?
- Check both `prev` and `next` for a self-loop?

```
struct node {
    struct node *prev;
    struct node *next;
};
```
Doubly Linked Lists
This is a doubly linked list, a la CS 106B

```
0x10  0x11  0x12  0x13  0x14  0x15  0x16  0x17  0x18  0x19
Req. 1  Free  Req. 2  Free  Req. 3  Free  Req. 4  Free  Req. 5  Free
```

heap_start_ptr

heap_end_ptr

list_start_ptr

NULL
Initial state (after `my_init`) (length 1)

- `list_start_ptr`: NULL
- `heap_start_ptr`: NULL
- `heap_end_ptr`: NULL
- `header (free, 2^32)`: NULL
- `prev`: NULL
- `next`: NULL
Possible Intermediate Empty State (length 0)

- Empty state: list_start_ptr is NULL
- 1-element state: list_start_ptr’s next and prev are both NULL
- 2+ element state: list_start_ptr’s next is not NULL
Explicit list-level Operations

- Iterate through the explicit list
- void add_free_block(struct node *new_free_payload);
- void detach_free_block(struct node *free_payload);

(Why?) Malloc algorithm:

- Foreach free block in explicit list:
  - If big enough
    - Detach from free list
    - Split block or take entire block
    - Add split free block back in, if there is one
    - Return