Lab 3: Malloc Lab
“What do we need to do?”
Logistics

* Due 11/26
* One more assignment after this one
* Partnering
  * Non-Honors students may work with one other person (if they so choose)
    * Must give credit where credit is due
  * Honors students may not work with a partner
* Only need to hand in one file (mm.c)
Overview of the Malloc Lab

* Implement a memory management library
  * Three main functions to be implemented:
    * `malloc()`: reserve a block of memory
    * `free()`: release a chunk of memory
    * `realloc()`: resize a reserved chunk
  * Two main goals (50/50 split):
    * Space efficiency (utilization)
    * Speed efficiency (throughput)
int mm_init(void)
* Will be called by the trace-driven program in order to perform necessary initializations
  * ie. allocating initial heap area
  * Return -1 if an error occurs and 0 if otherwise

void *mm_malloc(size_t size)
* Returns pointer to an allocated block payload of at least size bytes
  * Block must lie within heap area
  * Block cannot overlap with another allocated chunk
  * Must return 8-byte aligned pointer
void mm_free(void *ptr)
  * Frees the block pointed to by ptr
  * Returns nothing

void *mm_realloc(void *ptr, size_t size)
  * If ptr is NULL, call mm_malloc(size)
  * If size==0, call mm_free(ptr)
  * If otherwise, change the size of the block of memory pointed to by ptr to size bytes and returns the address of this new block
    * Possible for this address to remain the same as for the old block
      * ie: if a consecutive block of memory is free and satisfies the given size requirement
    * Contents of memory block remain the same (as much as possible if allocating to a smaller size block)
You will also need to implement a function called `mm_check()`

- Mostly for your own benefit
- Implementation will depend on your design
- Calls to `mm_check()` should be removed from the copy of your code that you turn in
  - We will check `mm_check()` manually for correctness and style
- Make sure to use `mm_check()` as you go along, preferably each time you add a new feature

Style pointzzz!!!!!!!!!!!!!!!@#$%^#^$%
What should you check the heap for?

- Is every chunk of memory in the free list actually free?
- Are there any consecutive free chunks of memory that escaped coalescing?
  - Coalescing means combining consecutive free chunks of memory into one big chunk of memory
- Is every chunk of memory that is free actually in the free list?
- Do any allocated chunks of memory overlap?
How will you keep track of used and freed memory chunks?
* You are not allowed to use the actual malloc() to make room for data structures

How will you track memory chunks while maintaining space efficiency?
* You must minimize storage overhead for metadata

How will you track memory chunks while maintaining speed efficiency?
* Topic of next recitation
What functions can you use?

* Since you cannot use the library malloc(), you will need to use:
  * mem_sbrk(n): grows heap by positive n bytes
  * mem_heap_lo(): pointer to the lowest byte in the heap
  * mem_heap_hi(): pointer to the first byte above the heap
  * mem_heapsize(): size of heap
    * size of heap = mem_heap_hi() – mem_heap_lo()
Functionally, arrays and pointers are the same

array[n] = *(array + n)

Arrays tend to be seen as more aesthetically pleasing while pointers tend to demonstrate a closer version to what the assembly instructions are doing to represent the higher level language code

Implicit multiplication

ptr + offset = (char*)ptr + (sizeof(*ptr))*(offset)

threeintegers[1] = *(threeintegers + 4 bytes)

You cannot offset void* ptr because the implicit coefficient is impossible to determine

Solution: type-casting (ie: *((char*)ptr + 1) = *(ptr + 1 byte))
Advanced Features and Performance Tuning
No explicit structure is used to track the location of free/allocated memory chunks

Instead, the overhead in each chunk containing the size of the chunk and the allocated bit of the chunk form a “block list”

Note that all chunks of memory will be in this list, not just the free chunks

Location of the next chunk determined based on the size of the current chunk
Finding a free block

First fit:
- Search list from beginning and choose first free block that fits; if no such block exists, must call mem_sbrk to extend the heap area
- Takes time linear in total number of blocks (free and allocated)

Next fit:
- Starts first fit search where the last search left off instead of at the beginning of the list

Best fit:
- Search entirety of list and choose the free block closest in size to the size desired
- Slower than first fit and next fit
Implicit Free List (Continued)

- **a** = 1: block is allocated
- **a** = 0: block is free

**size**: block size

**payload**: data

*Pictures adapted from Matt Welsh, Harvard University*
Implicit Free List (Continued)

Coalescing

```
addblock(p, 4)
```

```
free(p)
```
Implicit Free List (Continued)

Dangers of Not Coalescing:

There is enough free space, but the allocator won’t be able to find it!
Implicit Free List (Continued)

How would you coalesce with the previous block of memory?

<table>
<thead>
<tr>
<th>a</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>payload and padding</strong></td>
<td></td>
</tr>
</tbody>
</table>

a = 1: block is allocated  
a = 0: block is free

<table>
<thead>
<tr>
<th>a</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boundary tag</strong> (footer)</td>
<td></td>
</tr>
</tbody>
</table>

payload: application data

― "Pointers" in headers

4 4 4 4 4 6 6 4 4

― "Pointers" in footers
Explicit Free List

- Linked list among free blocks
  - Use space typically reserved for actual data for metadata, specifically the link pointers
  - Typically doubly linked list is used
  - Still need boundary tags for coalescing
  - Links don’t necessarily need to be in the same order as the order of the blocks in memory
Explicit Free List (Continued)

* Insertion Policy
  * Address-ordered policy
    * Insert freed blocks so that the free list is always in address order
    * Requires a search using boundary tags
  * LIFO (last-in-first-out)
    * Insert freed block at the beginning of the list
      * Constant time operation
    * Possibly more fragmentation than with aforementioned policy
Explicit Free List (Continued)
Utilize different free lists for different size classes

- Each size class has its own collection of blocks
- May enable quicker determination of appropriately-sized chunks of memory
- May need to be used in combination with other types of lists such as an explicit free list in order to enable coalescing
Segregated Free List (Continued)

Segregated lists

Multiple bins

free-list

Key question: How do we decide bin-sizes?
* Store free chunks of memory is binary tree format, determining each chunk’s position within the tree based on the size of the chunk
  * Optimal choice of structure since finding a free chunk of size n takes $O(\log(n))$ time as opposed to $O(n)$ time with a simple linked list
  * Certain test cases you are given are optimized to work more efficiently when a binary tree free list is being utilized
* Things to think about:
  * How to store chunks of memory of equal size
  * How to coalesce consecutive free chunks of memory
More Heap Consistency Check

For( mem_block : list){
  if(alloc(mem_blocks){
    //1, check if header and footer are consistent
    // check alloc bit and size
  }
  else{
    //1, check if the mem_block exists in the free list
    //2, does size match? Is it free?
  }
}
When a request of size (n) is served with a free block of size (m) such that m > n

Solution/ Design Choice:
- to split the remaining (m-n) memory block and make it a new free block
- Or just assume it is allocated as well.
Suggestions on where to begin:

* Start with the implicit list implementation provided in your textbook
  * Code it and understand it!
* Think about how to implement a heap checker for the implicit list implementation
* Start implementing an explicit list
* Add to your heap checker as you add placement policies to your explicit list implementation
* Move on to a segregated free list implementation
* If you have time to spare move on to a binary tree implementation
Important.