CSE 351

Dynamic Memory Allocation
Dynamic Memory

• Dynamic memory is memory that is “requested” at runtime

• Solves two fundamental dilemmas:
  • How can we control the amount of memory used based on run time conditions?
  • How can we control the lifetime of memory?

• Important to understand how dynamic memory works:
  • We want to use allocators efficiently
  • Can result in many errors if used incorrectly
Dynamic Memory

• Example program:
  • Dynamically adds/removes/sorts nodes in a large linked list

• Without dynamically-allocated memory:
  • Use the `mmap()` or equivalent system call to map a virtual address to a page of physical memory
    • This essentially gives you a page of memory to use
  • Use pointer addition/subtraction to segment the page into linked list nodes
  • Manage which regions of the page have been used
  • Request a new page when that one fills up
  • Get fired from your job
  • MESSY! NOBODY DOES THIS!
Dynamic Memory

• With dynamically-allocated memory:
  • Use `malloc()` from the C standard library to request a node-sized chunk of memory for every node in the linked list
  • When removing a node, simply carry out the necessary pointer manipulation and use `free()` to allow that space to be used for something else
  • Keep your job!
• You will come to love `malloc()` because it does all the heap management for you…
• …But for the next week you will hate it, because you are in charge of implementing it
malloc()

• Provided to you by the C standard library using
  #include <stdlib.h>

• Programs allocate blocks from the heap by calling the malloc() function
  • The heap is the memory region dedicated to dynamic storage

• How to use malloc():
  • Takes a size_t representing the number of bytes requested
  • Returns a void* pointing to the start of the block or NULL if there was an error

  ```
  int * array = (int *)malloc(10 * sizeof(int));
  ```
free()

- Also part of the C standard library
- Programmers also need to be able to “free up” dynamically-allocated memory that they no longer need
- Simply pass free() a pointer to a block received from malloc()
  - Using free() allows for more efficient heap usage
  - Subsequent calls to malloc() will be able to re-use that block
- Double-free
  - This occurs when you free the same block twice
  - It usually results in a segmentation fault
  - We will see why that might occur when we look at how malloc() is implemented
The Heap

• What does the heap look like exactly?
  • Imagine a giant contiguous region of memory
• This region is segmented into free block and used blocks
  • The free blocks form an explicit, doubly-linked list
  • To allocate a block, we remove it from the list and return a pointer to it
  • To free a block, we insert it back into the list
Block header

- Every block has a 64-bit header
- Three of those bits are used for tags
  - LSB is set if the block is currently used (not in the free list)
  - Next bit (to the left) is set if the block preceding it in memory is used
  - The third bit is not used
- The upper 61 bits store the size of the block
- This 64-bit value is also referred to as the block’s “sizeAndTags”

```
sizeAndTags:
+-------------------------------------------+
| 63 | 62 | 61 | 60 | ... | 2 | 1 | 0 |
+-------------------------------------------+
  ^                                  ^
high bit                           low bit
```
The free block has a sizeAndTags value on either side of the free block.

It also has a pointer to the next and previous blocks in the list.
  - Remember, the blocks are not necessarily in address order, so the pointers can point to blocks anywhere in the heap.

Each free block is a BlockInfo struct followed by free space and the boundary tag (footer)

```c
struct BlockInfo {
    size_t sizeAndTags;
    struct BlockInfo* next;
    struct BlockInfo* prev;
};
```
Used blocks

- Used blocks only have a sizeAndTags, followed by the payload
- The payload is the actual block of memory returned to a user program that invokes `malloc()`
Putting it all together

- Initial 128-byte heap layout:
  - BlockInfo* FREE_LIST_HEAD that always points to the first block in the free list
  - The BlockInfo for this free block would look like this:
    - sizeAndTags: 130 (128 + 0x2)
    - next: null
    - prev: null
  - The PrecedingUsed tag is set because the previous block is not free (comes into play when we look at coalescing later)

Size: 128, Preceding Used: 1, Used: 0
Putting it all together

- `void* a = malloc(32):`
  - Searches the free list for a block big enough
  - The first (and only) block is 128 bytes, which will work
  - Bad implementation: return a 120-byte payload (8-byte header)
  - Good implementation: split off 40 bytes, return a 32-byte payload
Putting it all together

• `void* b = malloc(16);`:
  • Only needs a block of $16 + 8 = 24$ bytes, but if we were to free this block in the future, we would need at least $32$ bytes to create a free block. Thus, the minimum block size is $32$ bytes
Putting it all together

- `void* c = malloc(48):
  - `FREE_LIST_HEAD = null`
Putting it all together

- \texttt{free(b)}:
  - This will insert block b into the start of the list
Putting it all together

- \texttt{free(c)}:
  - Is this what the heap looks like at the end of \texttt{free(c)}?
Putting it all together

- When we have multiple free blocks adjacent to each other in memory, we should coalesce them
- Coalescing basically combines free blocks together
- Bigger blocks are always better; a large block can satisfy both large and small `malloc()` requests
Your assignment

- Implement `malloc()` and `free()`
- Before you start to feel overwhelmed...
  - We give you many functions already including:
    - `searchFreeList()`
    - `insertFreeBlock()`
    - `removeFreeBlock()`
    - `coalesceFreeBlock()`
    - `requestMoreSpace()`
Your assignment

- **Implementing `malloc()`**
  - Figure out how big a block you need
  - **Call `searchFreeList()`** to get a free block that is large enough
    - **NOTE:** If you request 16 bytes, it might give you a block that is 500 bytes
  - Remove that block from the list
  - Update size + tags appropriately
  - Return a pointer to the payload of that block
Your assignment

- **Implementing** `free()`
  - Convert the given used block into a free block
  - Insert it into the free list
  - Update size + tags appropriately
  - Coalesce if necessary by calling `coalesceFreeBlock()`
Starter code

- We’ll now go through some of the starter code included in the assignment
- If you are struggling to understand where to get started, read through `coalesceFreeBlock()`
  - If you can understand this function, you will understand everything
- Make sure you use the provided macros
  - They work, so it will help minimize bugs
  - More readable code