CSE 351
Dynamic Memory Allocation
Dynamic Memory

Dynamic memory is memory that is “requested” at run-time

Solves two fundamental dilemmas:

- Persistent storage
- Program size can scale to meet input demands at run-time
Dynamic Memory

Example program A:
• 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 dynamic memory allocation

• 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
The function `malloc()` is not a system call provided by the OS

- It is a function exposed by the C standard library

A program compiled with `<stdlib.h>` will initialize the data structures necessary to manage the heap before running user code

- The heap is a region of the address space 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
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?

The heap can be thought of as 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
Every block has a 64-bit header

Three of those bits are used for tags
  • One is set if the block is currently used (not in the free list)
  • One 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”
Free blocks

The free block has a sizeAndTags value at the top and bottom
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:
• There is a void* HEAD that always points to the first block in the list

Size: 128, Used: 0, Preceding Used: 0
next = null;
prev = HEAD;
Putting it all together

```c
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

```c
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

```c
void* c = malloc(48);
```
Putting it all together

```c
free(b):
  • This will insert block b into the start of the list
```
Putting it all together

```c
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 all the other code
- We set up the free list
- We give you a function that finds a free block of a given size
- We give you a coalesce function
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