CSE 351 Section 9

Dynamic Memory Allocation
Dynamic Memory

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

- Solves two fundamental dilemmas:
  - How can we control the amount 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
Example Program: why dynamic allocation?

Goal: Dynamically add/remove/sort nodes in a large linked list

Option 1: 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
- MESSY! NOBODY DOES THIS!
Example Program: why dynamic allocation?

Goal: Dynamically add/remove/sort nodes in a large linked list

Option 2: 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
- You will come to love `malloc()` because it does all the heap management for you...
- ...But for the next week it may be more annoying 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
- Run `man malloc` in a Linux terminal for more information!

- 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

```c
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
  - Later calls to malloc() will be able to re-use that block

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

Double-free
- This occurs when you free the same block twice
- It usually results in a segmentation fault
  - It will become more apparent why when you learn how malloc() is implemented

```c
int* array = (int*) malloc(10 * sizeof(int));
...
free(array);
free(array); // Double free...ouch...
```
The Heap

What does the heap look like exactly?

- Imagine a giant contiguous region of memory
- This region is segmented into free blocks and used blocks
- Consecutive free blocks form what we call a “free list”
- Two types of free lists:
  - Implicit: use block sizes to traverse the heap looking for a free block
  - Explicit: use doubly linked list of free blocks to find a free block
Implicit vs. Explicit Free List

**Implicit:** Using sizes to traverse blocks, checking to see if each block is allocated

**Explicit:** Using pointers to create linked list of free blocks
Block Header Format

- Each block needs to indicate its size, if it is used, and if the prev block is used
- Could use two 8-byte fields, but wastes a lot of space
- Standard trick:
  - Since size will always be aligned to a certain multiple of 2, some of the lower order bits will be 0 (for instance if all sizes are multiples of 8, the lowest 3 bits will always be 0)
  - Can store additional tag information in those lowest bits
  - Just need to remember to mask them away when reading the size (see the SIZE macro in the lab5 starter code!)
Block Header Format

- Every block has a 8-byte (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”
Free Blocks

A free block has:

- A sizeAndTags value on either side of the free space.
- Pointers 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).

<table>
<thead>
<tr>
<th>sizeAndTags</th>
<th>struct BlockInfo *next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free space</td>
<td>struct BlockInfo *prev</td>
</tr>
</tbody>
</table>

```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().

Example:

```c
int* a = (int*) malloc(10 * sizeof(int));
```

*a points to the payload (not the start of the block!)*
Walkthrough of Example Heap

Initial Heap

Note FREE_LIST_HEAD always points to the first block in the free list.
void *ptr1 = malloc(32);

- Need to search free list to find a block big enough for 40 (32 + header) bytes
Walkthrough of Example Heap

void *ptr1 = malloc(32);

- Note that ptr1 points to the start of the payload, NOT the start of the block
- The initially 256 byte free block is split to maximize memory usage!
Walkthrough of Example Heap

void *ptr2 = malloc(16);

- Only need a block of 24 (16 + header) bytes, but what if we needed to free it later… think about what the minimum block size needs to be
Walkthrough of Example Heap

void *ptr2 = malloc(16);

- Need at least 32 bytes to create a free block, meaning we must allocate at least this much for a used block!
Walkthrough of Example Heap

void *ptr3 = malloc(24);

- Same procedure as before
Walkthrough of Example Heap

void *ptr3 = malloc(24);

- Same procedure as before
Walkthrough of Example Heap

free(ptr2);

● Now we need to free a block!
Walkthrough of Example Heap

```c
free(ptr2);
```

- Need to insert block allocated for `ptr2` into the free list (and update tags!)
- Which tags get updated?
Walkthrough of Example Heap

code{free(ptr3);}

- Same thing as before, except now the pointers get really messy...

![Diagram of heap with pointers and nodes.](image-url)
Walkthrough of Example Heap

free(ptr3);

- Same thing as before, except now the pointers get really messy...
  - next pointers are the ones higher up in the diagram, prev lower down...
Walkthrough of Example Heap

```c
free(ptr3);

- Good enough? What happens if user calls malloc(200)?
```
Walkthrough of Example Heap

```
free(ptr3);

- Coalesce neighboring free blocks into one large free block!
- Allows for larger future mallocs, can still split later for smaller chunks
```
Lab 5

- You get to implement malloc() and free()!
- Less overwhelming than it may sound, we give you many functions already including:
  - `searchFreeList()`
  - `insertFreeBlock()`
  - `removeFreeBlock()`
  - `coalesceFreeBlock()`
  - `requestMoreSpace()`
  - see spec/starter code for full list!
Some notes about 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
  - Might have to splice into a smaller/bigger chunk (see NOTE above)

- Update size + tags appropriately (do neighbor blocks need updating?)

- Return a pointer to the payload of that block
Some notes about implementing `free()`

- Remember, the pointer you are passed is to the payload!
- Convert the given used block into a free block
- Insert it into the free list
- Update size + tags appropriately (do neighbor blocks need updating?)
- Coalesce if necessary by calling `coalesceFreeBlock()`
C Macros

Pre-compile time “find and replace” your code text

Defining constants:
- `#define NUM_ENTRIES 100`
  - OK

Defining simple operations:
- `#define twice(x) 2*x`
  - Not OK, `twice(x+1)` becomes `2*x+1` because preprocessor uses naive find and replace
- `#define twice(x) (2*(x))`
  - OK, now `twice(x+1)` becomes `2*(x+1)`
  - Always wrap in parentheses!
Why even use Macros?

- Why macros?
  - Create more readable/reusable code for constants
  - “Faster” than function calls
  - In malloc: Quick access to header information (payload size, valid)

- Drawbacks
  - Less expressive than functions
  - Arguments are not typechecked, local variables
  - They can easily lead to errors that are more difficult to find (see prev slide)
Some Lab 5 Provided Macros

- **UNSCALED_POINTER_ADD**(\(p, x\)) Add without using “pointer arithmetic”
- **UNSCALED_POINTER_SUB**(\(p, x\)) Subtract without using “pointer arithmetic”
- **MIN_BLOCK_SIZE** The size of the smallest block that is safe to allocate
- **SIZE**(\(x\)) Gets the size from ‘sizeAndTags’
- **TAG_USED** Mask for the used tag
- **TAG_PRECEDING_USED** Mask for the preceding used tag
- ...

There are lots more, don’t forget to use them!
- They will absolutely make your life easier
- Part of good C style (which will be part of this assignment’s grade)
Getting Started Lab5:

- If you are struggling to understand where to get started, read through `coalesceFreeBlock()`
  - Understanding the details of this function will provide clarity on the general structure you are manipulating
- Make sure you use the provided macros!
  - They work, so it will help minimize bugs
  - More readable code