CSE 410: Systems Programming

Dynamic Allocator Project

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

The Standard Allocator

The standard allocator provides a convenient method to:

- Allocate memory on demand
- Release memory when it is no longer needed

The UNIX system calls for memory management either:

- require the application to do extra bookkeeping work, or
- do not reliably allow for releasing memory.

In particular, the user need not track allocation sizes when using the standard allocator.

The Standard Allocator Interface

There are three allocation functions in the standard allocator:

- `void *malloc(size_t size);`
  Allocates `size` bytes of memory.
- `void *calloc(size_t nmemb, size_t size);`
  Allocates an array of `nmemb` elements of `size` bytes each, then zeroes the entire array.
- `void *realloc(void *ptr, size_t size);`
  Behaves like `malloc()` if `ptr` is `NULL`, otherwise adjusts the allocation of `ptr` to be `size` bytes if possible. If this is not possible, it creates a new allocation of `size` bytes and copies the contents of `ptr` into the new allocation.

Freeing Memory from the Standard Allocator

Any allocation made by the standard allocator can be freed with `free()`.

```
void free(void *ptr);
```

This will return the freed memory to the heap.

Freed memory may be used again for future allocations.

Allocation Sizes

Note that `free()` and `realloc()` must both know allocation sizes.

- `free()` must return memory to the heap
- `realloc()` might return memory to the heap, might copy memory, or might adjust an existing allocation size

Note also that the allocator-returned pointers:

- Allow the user to use memory starting immediately at the pointer
- Don't return any other user-visible metadata

This dictates that object size is stored somewhere else.

Space Between Allocations

The space between allocations can be used for metadata.

In particular, the space immediately before an allocation:

- Is unlikely to be accidentally accessed
- Is at a fixed offset from the allocation pointer

Contrast to the space after an allocation:

- Likely to be corrupted by array overruns
- At a variable offset from the allocation pointer

By making allocations somewhat larger and using the extra space to store metadata, an allocator can provide easy, simple interfaces.

Accessing Metadata Before the Allocation

Pointer math can be used to access allocation metadata.

For example, if the allocation size is the pointer word before an allocation:

```
#include <stdint.h>

void free(int *ptr) {
    uintptr_t *sizeptr = ptr - sizeof(uintptr_t);
    uintptr_t size = *sizeptr;
    ...
```

(Normally, of course, you wouldn't do that in two steps…)

Project Heap Structure

In your project, blocks (whether allocated or free) have a header and a footer.

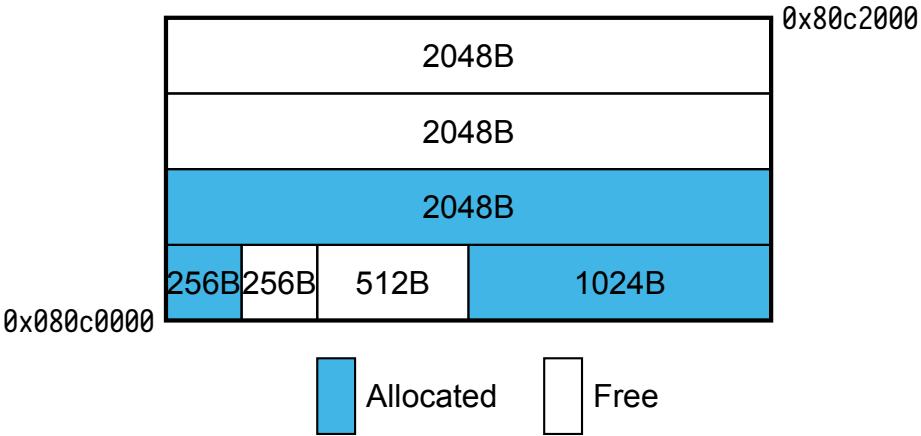
Both are 8 bytes (one pointer word) in size.

Blocks are side-by-side in address space on the heap.

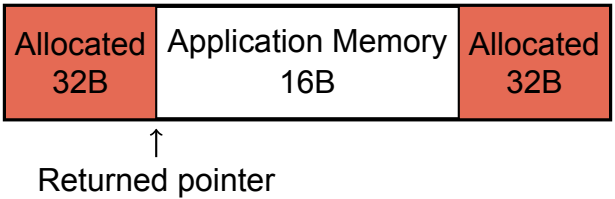
Each free block is tracked by the allocator.

Allocated blocks are the responsibility of the application.

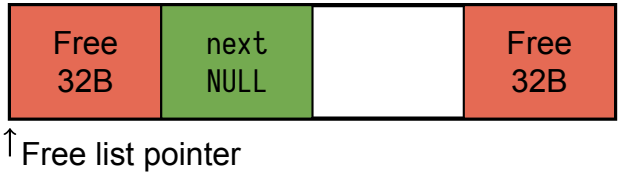
The Heap as an Array



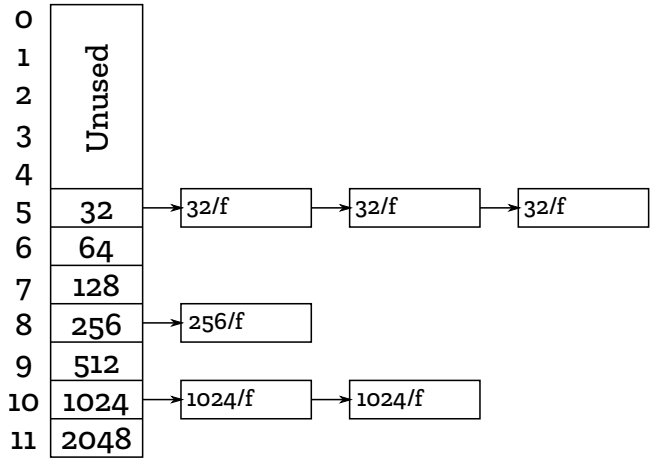
An Allocated Block



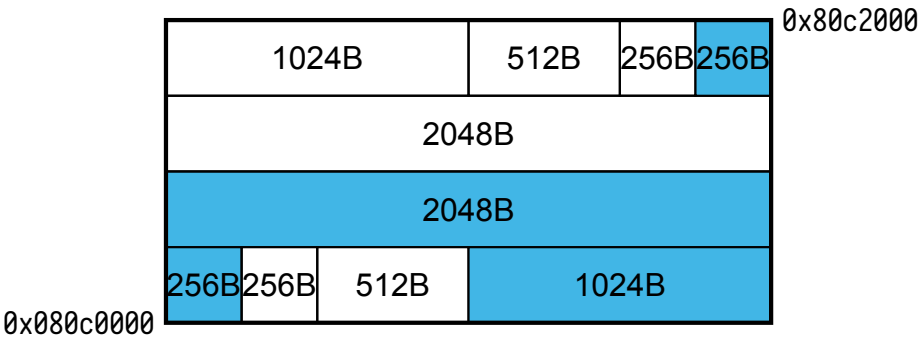
A Free Block



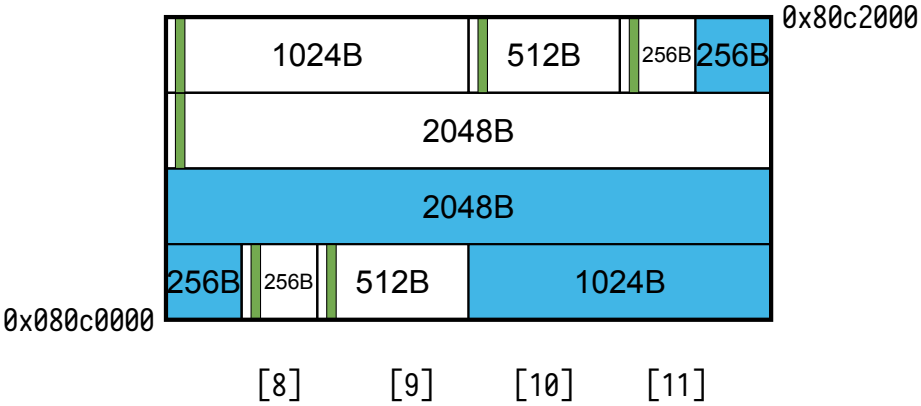
Free Lists



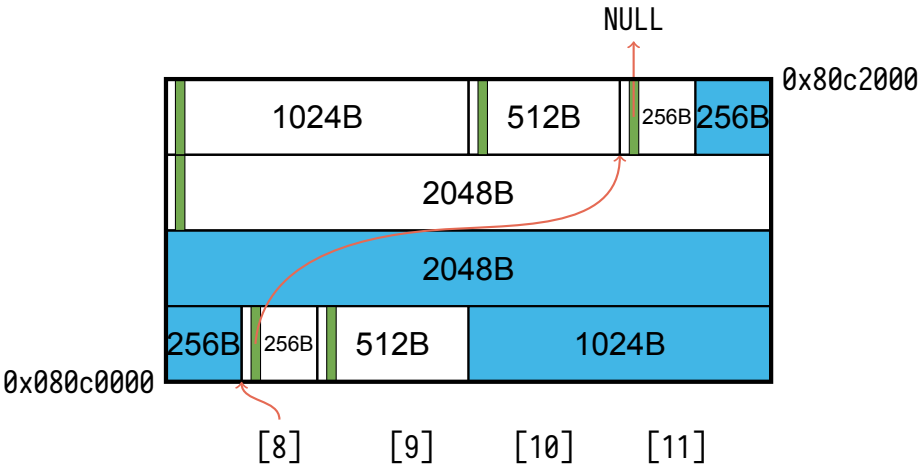
Threaded Heap



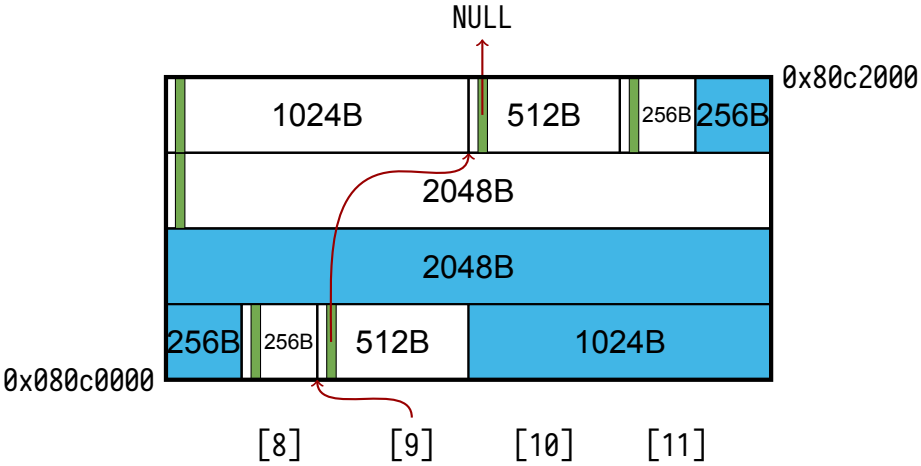
Threaded Heap



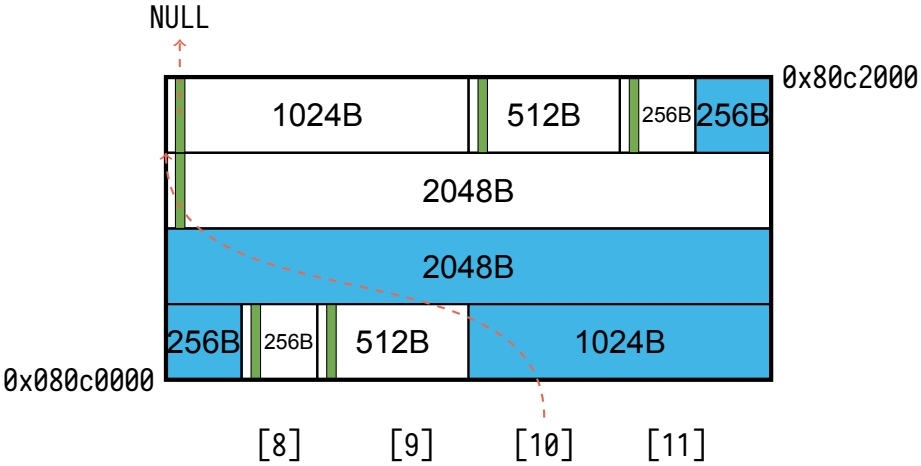
Threaded Heap



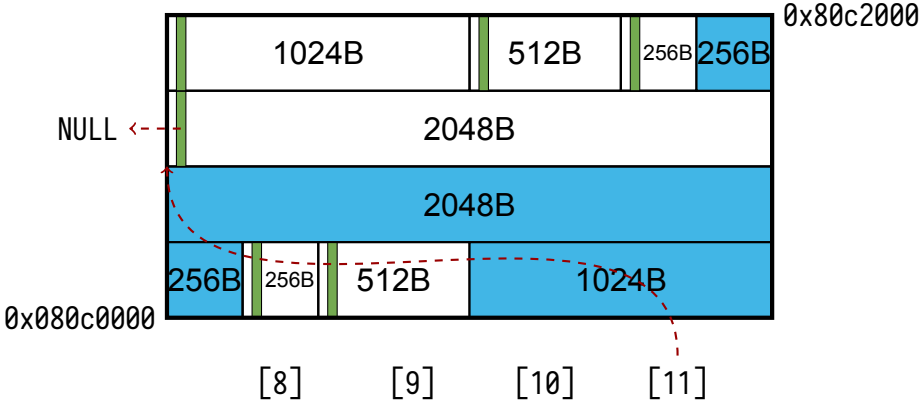
Threaded Heap



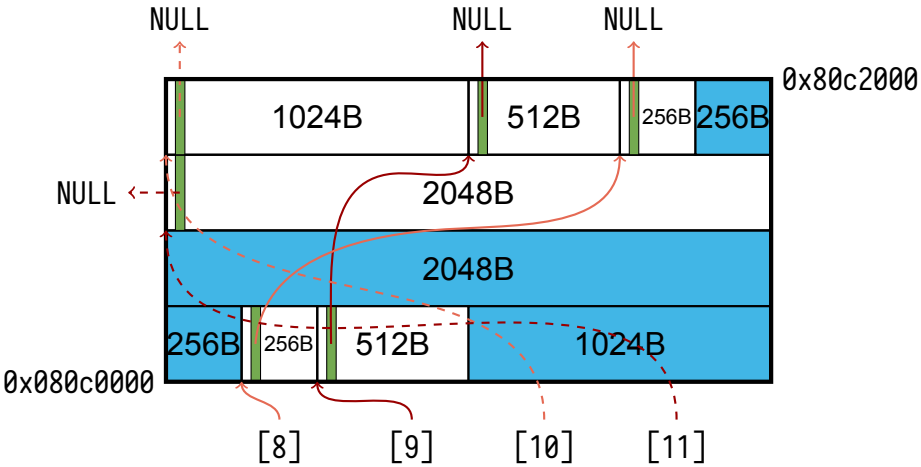
Threaded Heap



Threaded Heap



Threaded Heap



Joining Buddy Blocks

In order to efficiently use space, you must coalesce free blocks.

This means adjacent blocks of the same size should be joined

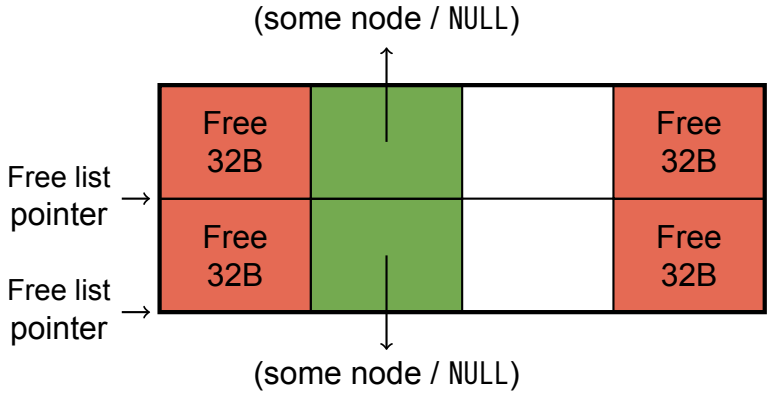
The text description joins all adjacent blocks.

However, this is not alloawable in our buddy allocator!

Adjacent blocks can be coalesced if:

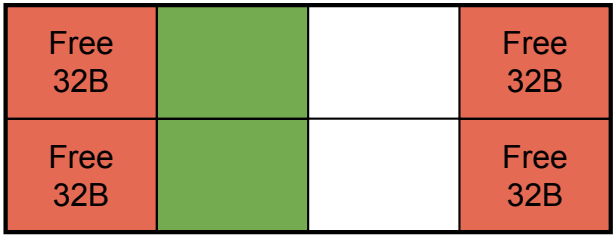
- they are the same size
- they are smaller than the max block size

Coalescing



Two adjacent free nodes
(May or may not be list-adjacent!)

Coalescing

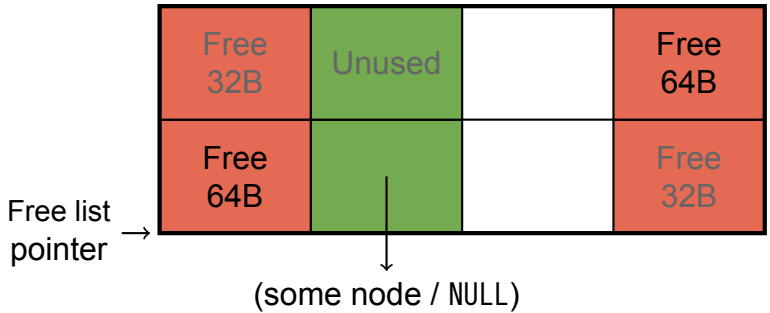


Remove both blocks from the free list.
You may find a doubly linked list helpful for this.

Coalescing

Join the two blocks

Coalescing



Put the new block on the appropriate free list

Finding Blocks to Coalesce

How do you find out if a block's neighbors are free?

Finding Blocks to Coalesce

How do you find out if a block's neighbors are free?

Pointer math again!

The footer of the previous block
…is the pointer word before the header of this block.

The header of the next block
…is the pointer word after the footer of this block.

Sentinels

The first block and the last block on the heap are a problem.

…they don't have neighbor words of predictable content!

We can fix this with sentinels.

A sentinel is data placed in memory to delineate a boundary.

An artifically "allocated" block can be used as a sentinel.

Note that an extra sentinel is required for each gap in the heap.
(*E.g.*, due to some other code using `sbrk()`.)

The final sentinel from one call to `sbrk()` may be freeable after the next call to `sbrk()`.

Summary

- The standard allocator must keep track of information in the heap.
- We're keeping metadata between user-allocated objects.
- A header and a footer make object freeing and coalescing fast and precise.
- The heap data structure has a dual nature:
  - a continuous stream of objects in address space
  - multiple lists threading through the free objects
- Sentinels mark the boundaries of heap-managed space.

References I

Optional Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 9: 9.9. Pearson, 2016.

License

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.