CSE 220: 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.

©2019 Ethan Blanton / CSE 220: Systems Programming

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.

©2019 Ethan Blanton / CSE 220: Systems Programming

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.

©2019 Ethan Blanton / CSE 220: Systems Programming

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.

©2019 Ethan Blanton / CSE 220: Systems Programming

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.

©2019 Ethan Blanton / CSE 220: Systems Programming

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…)

©2019 Ethan Blanton / CSE 220: Systems Programming

Project Heap Structure

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

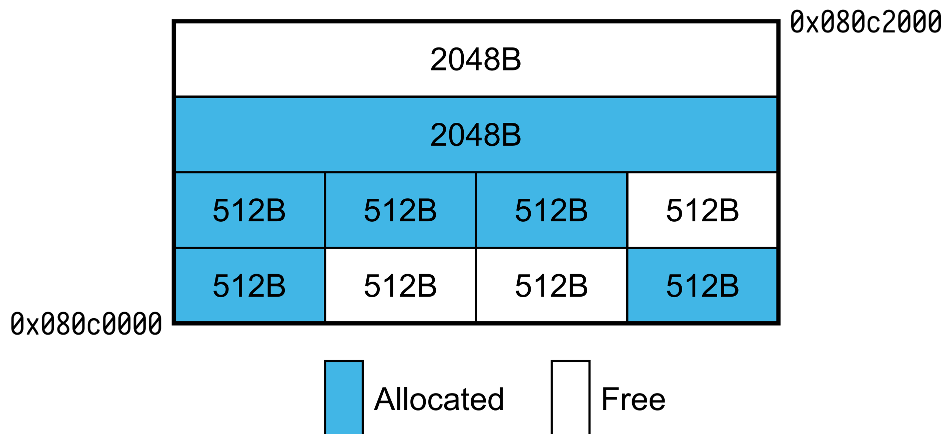
It is 8 bytes (one pointer word) in size.

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.

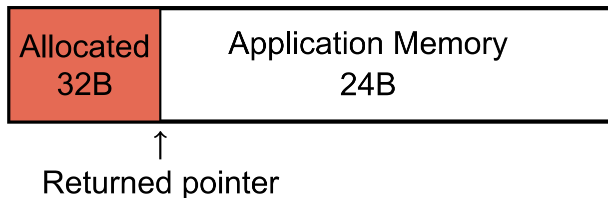
©2019 Ethan Blanton / CSE 220: Systems Programming

The Heap as an Array



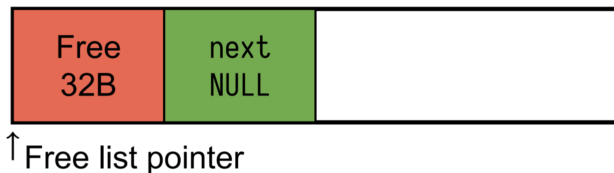
©2019 Ethan Blanton / CSE 220: Systems Programming

An Allocated Block



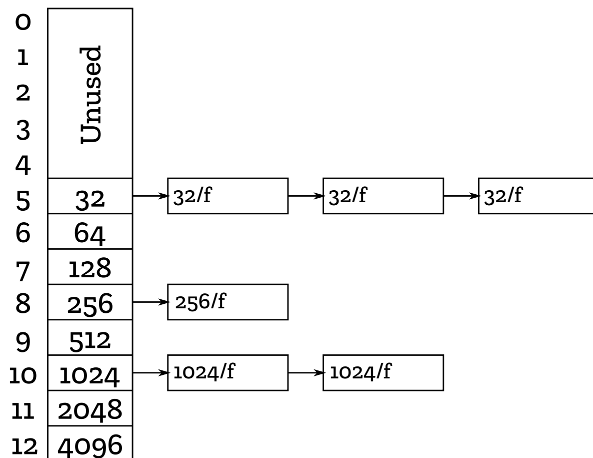
©2019 Ethan Blanton / CSE 220: Systems Programming

A Free Block



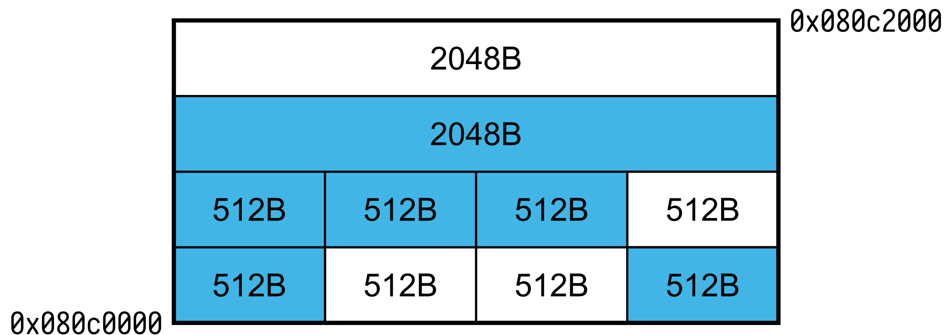
©2019 Ethan Blanton / CSE 220: Systems Programming

Free Lists



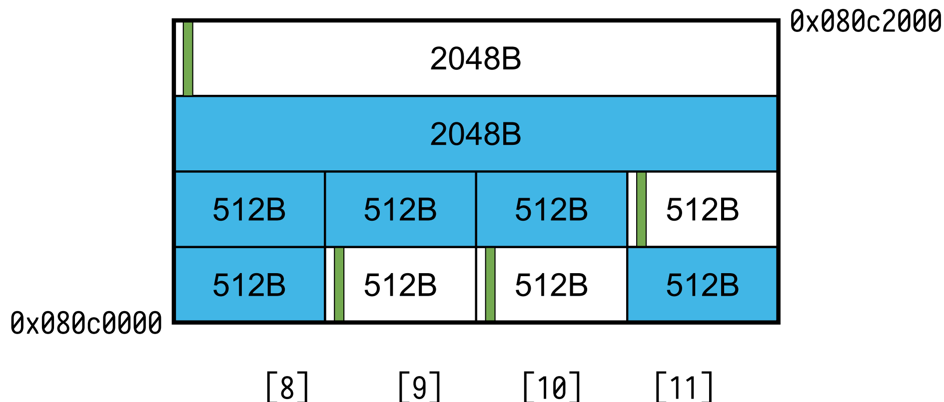
©2019 Ethan Blanton / CSE 220: Systems Programming

Threaded Heap



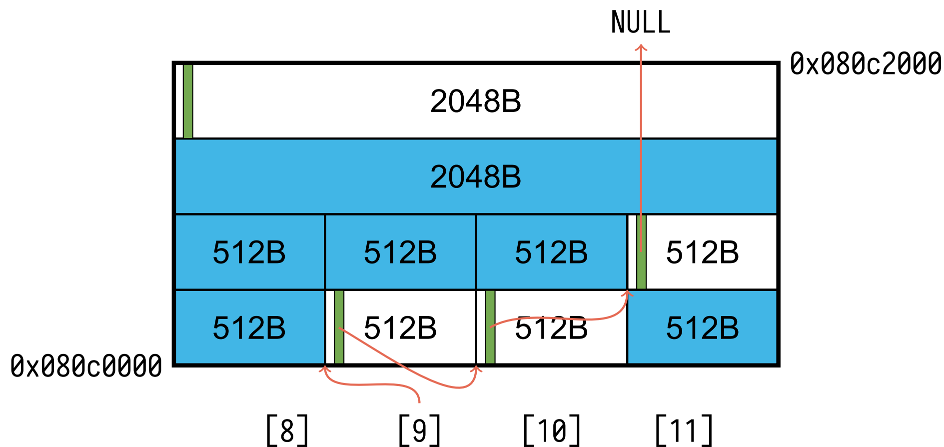
©2019 Ethan Blanton / CSE 220: Systems Programming

Threaded Heap



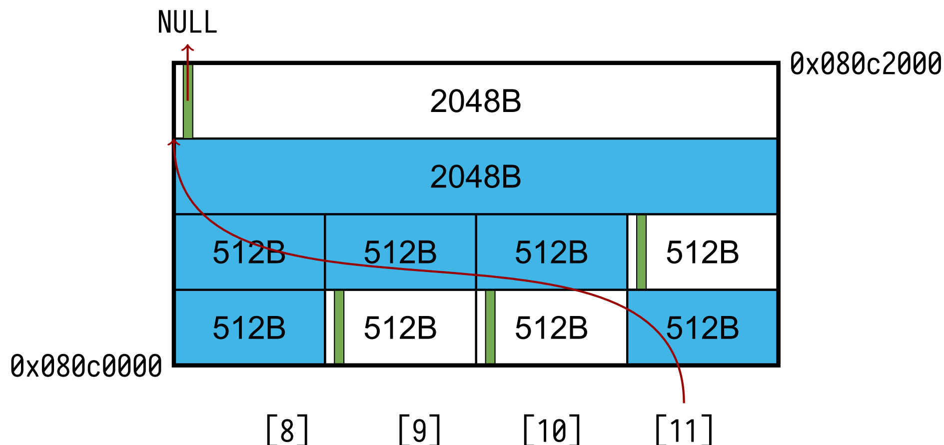
©2019 Ethan Blanton / CSE 220: Systems Programming

Threaded Heap



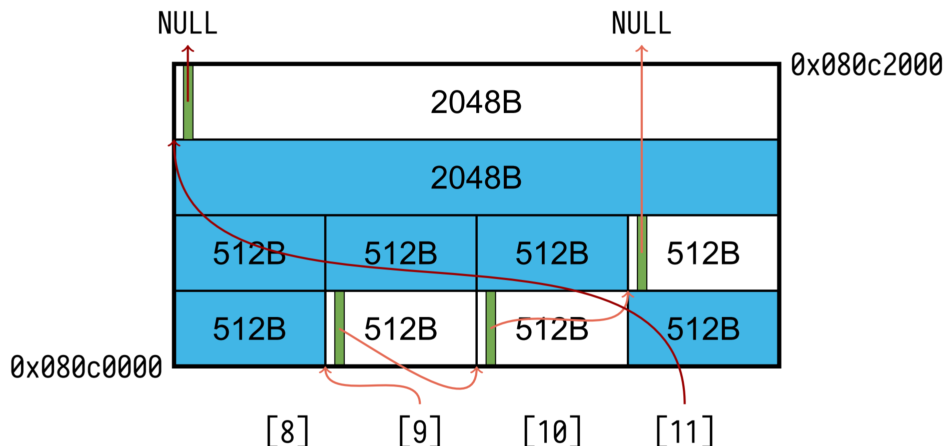
©2019 Ethan Blanton / CSE 220: Systems Programming

Threaded Heap



© 2019 Ethan Blanton / CSE 220: Systems Programming

Threaded Heap



Summary

- The standard allocator must keep track of information in the heap.
- We're keeping metadata between user-allocated objects.
- A header on each object indicates its size.
- The heap data structure has a dual nature:
  - a continuous stream of objects in address space
  - multiple lists threading through the free objects

©2019 Ethan Blanton / CSE 220: Systems Programming

References I

©2019 Ethan Blanton / CSE 220: Systems Programming

License

Copyright 2018, 2019 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

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