Malloc: Gear Up

Overview

- Dynamic Storage Allocator
  - Create storage as needed
  - Use a doubly-linked explicit free list to manage free blocks
- Functions You Will Write
  - mm_init(), mm_malloc(), mm_free(), mm_realloc()
  - mm_check_heap()

Roadmap

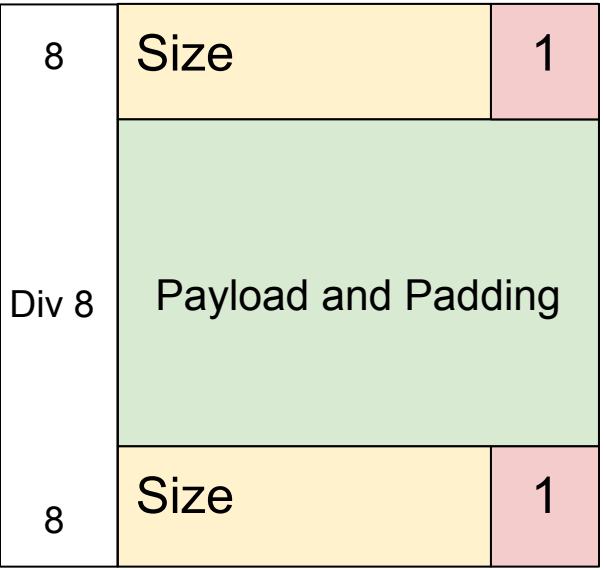
Install: **cs0330_install_malloc**

Coding

- mm_init()
- mm_check_heap()
- mm_malloc()
  - mm_extend_heap()
- mm_free()
  - coalesce()
- mm_realloc()

To Run: **./mdriver -V**

Blocks: Structure

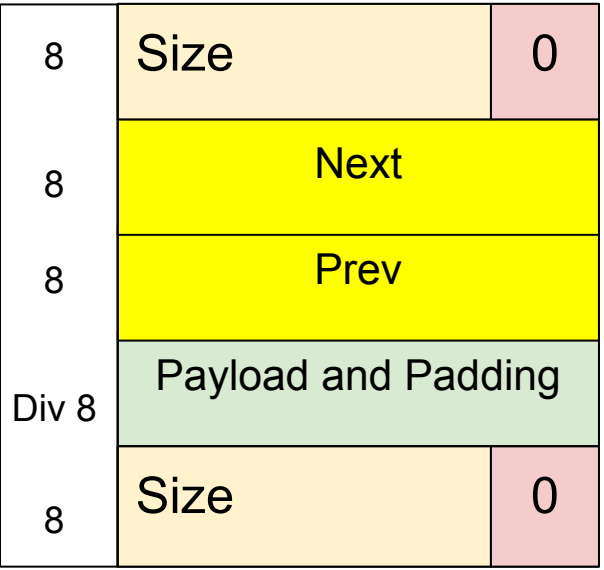


Header: Contains the **size** of the block (note: size is inclusive of the header and footer) and a **boolean** representing whether the block is allocated or free

Payload: For allocated blocks, this is where the data is stored

Footer: Same as the header. We use it to coalesce free blocks.

Free Blocks: Structure



Next and Prev: Free-list pointers

- **Next**: Points to the next free block in the free list
- **Prev**: Points to the previous free block in the free list

In our implementation, the free list is a **circularly, doubly linked list**. What does this mean for prev and next pointers in an empty list? single-item list? n-item list?

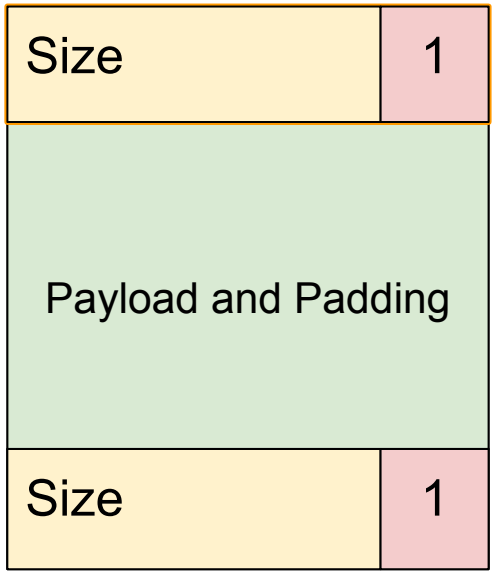
Blocks: Struct

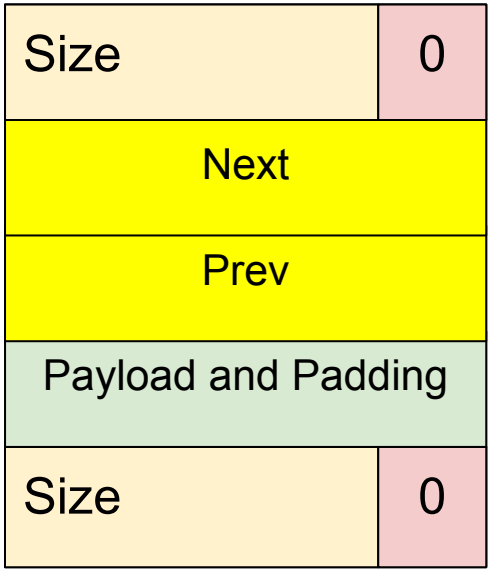
```
typedef struct block {
    size_t size;
    size_t payload[0];
} block_t;
```

- The `payload[0]` means that the array can be of any length
- Use the inline functions defined for you in mminline.h to manipulate blocks.

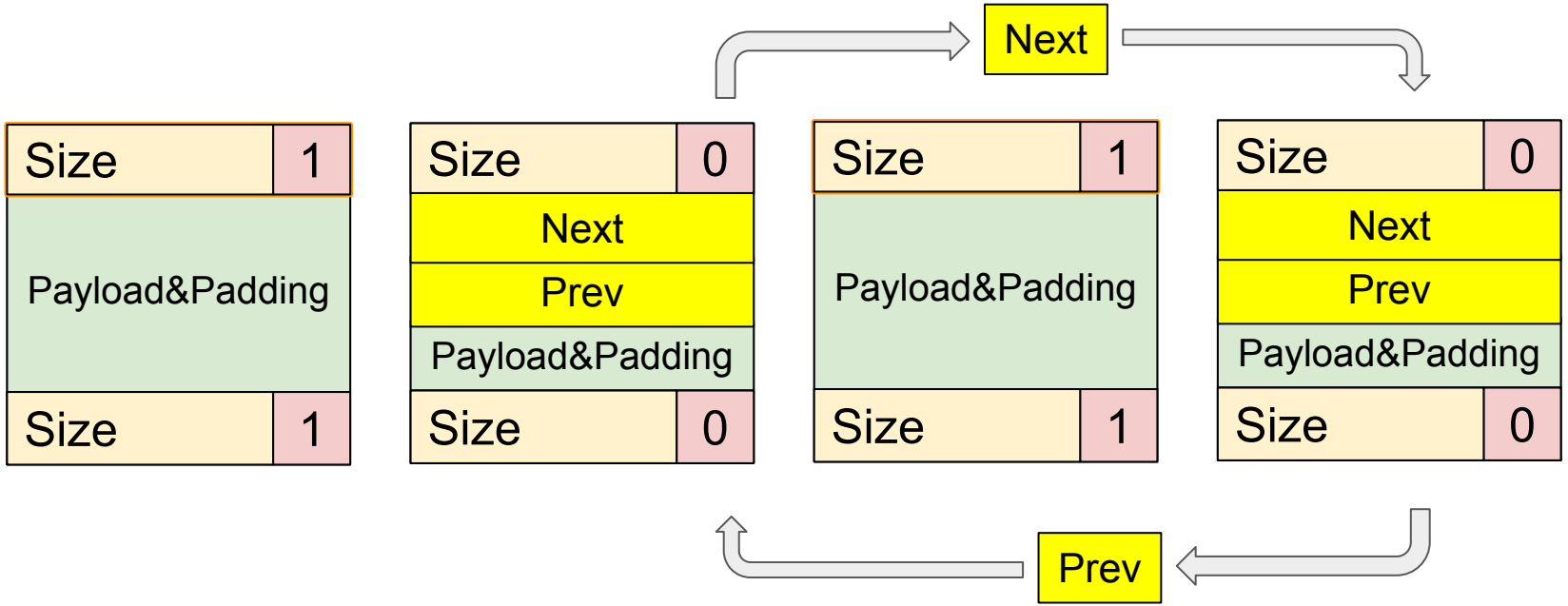
Discussion

What do you think the minimum block size is?

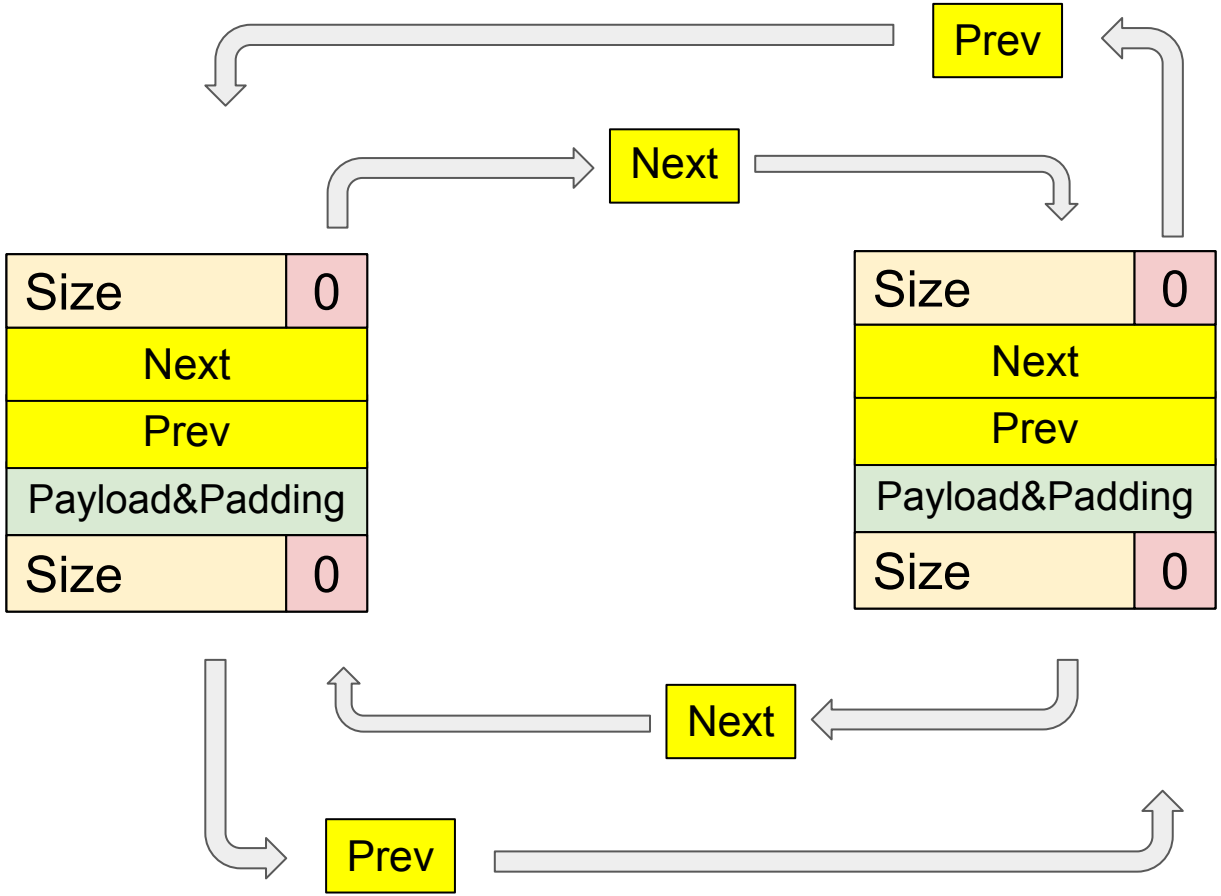




The Free List, a visual overview

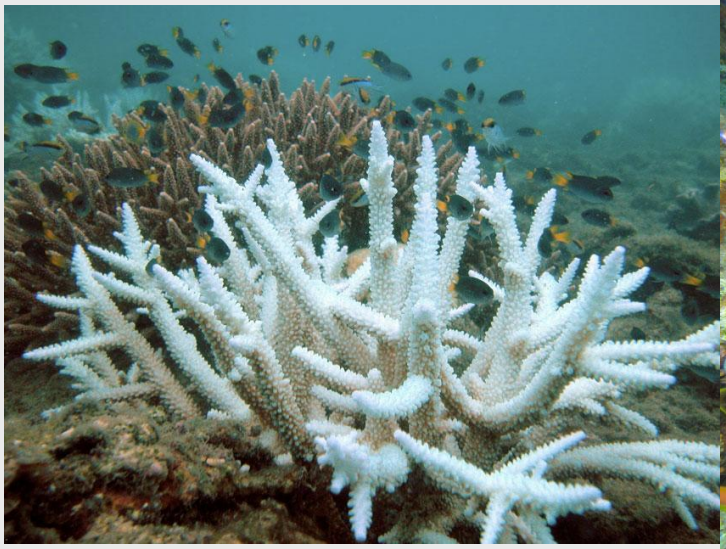


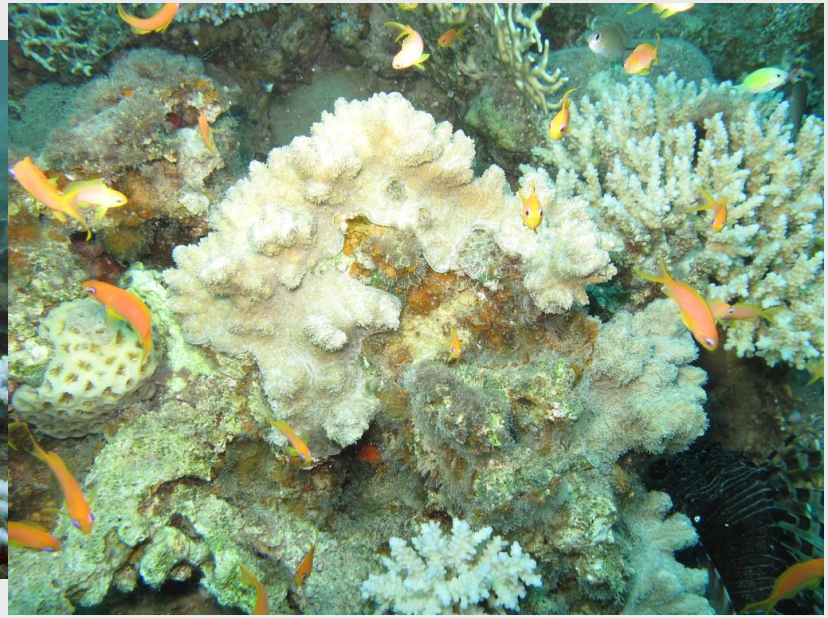
The Free List, a visual overview



Malloc

Implementing a successful malloc!





What functions do you need to implement?

`mm_init``(void)`: initializes the dynamic storage allocator

`mm_malloc``(size_t size)`: allocates a block of memory, returning a pointer to that block's payload

`mm_free``(void *ptr)`: frees a block of memory. This block can now be reused!

`mm_realloc``(void *ptr, size_t size)`: reallocates a memory block to update it with a new size

`mm_check_heap``(void)`: checks the heap for consistency

`mm_init(void)`

- Start by building prologue and epilogue blocks
- What type should these be?
- Should these be allocated or free?
- What other qualities about these blocks should you set?
- Remember to error check!
- Are there other objects that need to be set?

`mm_extend_heap(size_t size)`

What does extending the heap mean?

- Create a new free block
  - Break more memory in the heap
  - Error check
  - Set free block's attributes
  - Check: Is this free block next to other free blocks? If so, what should we do?
- Overwrite the epilogue
  - Reset the epilogues attributes
- Examine the input
  - What happens if the size input isn't a multiple of 8?
  - What can we do to ensure that it will be?

`mm_malloc(size_t size)`

- Find the first free block in the list
- What do you do if you can’t find a fit?
- Once you find a fit, decide whether or not you have to split the free block you want to allocate
- Check for edge cases (very important!!!!)
  - What if the caller tries to allocate memory of size 0?
  - What if the size isn’t aligned to 8 bytes?
  - What if you have to split the block?

`mm_free(void *ptr)`

- Retrieve the pointer to the header of the block
- Mark the header and footer as free (using the last bit in each)
- Remember to coalesce if necessary!

mm_realloc(void *ptr, size_t size)

What could you consider in implementing realloc?

- Cover the edges
  - What if the pointer is null?
  - What does it mean if the size is 0?
- Create a block for the pointer
  - What would it mean if this block was free? Allocated?
- Sizing: the original block had a size, the user has requested a new size
  - How should we keep track of these?
  - Is the requested size appropriately formatted? How do we know? What can we do if it isn't?
- Neighboring blocks
  - Are these free or allocated? What does that mean for our new block?

mm_realloc(void *ptr, size_t size)

What could you consider in implementing realloc?

- Backing up our data to the stack
  - Max memory realloc.rep asks for is 615000
  - Using memcpy: What is the size we should copy over? How could we save space?
- Set our block free!
- Maintaining the free list
  - How can we avoid undefined behavior when splitting or coalescing?

mm_realloc(void *ptr, size_t size)

Fit

- How do we determine if we fit?
- If we do fit, what do we do?
  - Get the new address
  - What should we consider about the size that's left over?
  - What if we're at the same address as before? What should we copy?
- If we don't fit, what do we do?
  - Take from the free list
    - There are many different ways to do this!

mm_check_heap(void)

- What would be good to check in this function?
  - Is everything aligned properly?
  - How can we keep track of the free blocks?
    - Is every free block actually in the free list?
    - Check free list has valid free blocks: is every block in the free list marked as free?
    - Are there any contiguous free blocks that somehow escaped coalescing?
    - Do the pointers in the free list point to valid free blocks?
  - Can you access the prologue and epilogue blocks properly?
- Use assert statements
- **Must have a viable mm_check_heap() before a TA can help you debug heap corruption bugs**

Where should I start?

Understand what happens to the blocks using the inline functions

Answer the questions posed in these slides

Coding

- Start with mm_init, mm_extend_heap
- Move on to malloc, free, realloc, and coalesce
- Implement mm_check_heap as you go along!!

Helpful GDB Tips!

Look at what is pointed to by the pointer blk by typing print *blk or p *blk in GDB

- This dereferences the pointer and lets you see the size and allocation status
- If it's allocated, the size will show up as a multiple of 4 plus 1, otherwise just a multiple of 4

REPL

- `mdriver` comes with a REPL you can use to test your malloc
- See section 4 of the handout!
- Run with `./mdriver -r`

```
Welcome to the Malloc REPL. (Enter 'help' to see available commands.)
> help
commands:
malloc <index> <size>    mallocs the block at <index> to a size <amount>
realloc <index> <size>   reallocs the block at <index> to <amount>
free <index>             frees block at <index>
check_heap               calls mm_check_heap() to test validity
print                    prints the heap
print -b <index>         prints the status of the block at <index>
quit                     quits repl
> malloc 1 99
> malloc 2 100
> print
heap size: 544
prologue                block at 0x7f7934a91010         size 16
free block              block at 0x7f7934a91020         size 272        Next: 0x7f7934a91020
block[2] allocated      block at 0x7f7934a91130         size 120
block[1] allocated      block at 0x7f7934a911a8         size 120
epilogue                block at 0x7f7934a91220         size 16


> print -b 2
block[2] allocated      block at 0x7f7934a91130         size 120
> free 2
```

Traces

A trace file is an ASCII file. It begins with a 4-line header:

```
<sugg_heapsize> /*suggested heap size (unused)*/
<num_ids>       /*number of request id's*/
<num_ops>       /*number of requests
<operations>
```

```
2000
6
12
1
```

```
a 0 2040
a 1 2040
f 1
a 2 48
a 3 4072
f 3
a 4 4072
f 0
f 2
a 5 4072
f 4
f 5
```

Traces, continued

The header is followed by num_ops text lines. Each line denotes either an allocate[a], reallocate[r], or free[f] request. The <alloc_id> is an integer that uniquely identifies an allocate or reallocate request.

```
a <id> <bytes>   /*ptr_<id> = malloc(<bytes>)*/
r <id> <bytes>   /*realloc(ptr_<id>, <bytes>)*/
f <id>           /*free(ptr_<id>)*/
```

```
2000
6
12
1
```

```
a 0 2040
a 1 2040
f 1
a 2 48
a 3 4072
f 3
a 4 4072
f 0
f 2
a 5 4072
f 4
f 5
```

Malloc

You can do this, and we're here to help!