Malloc Bootcamp

Minji, Pallavi, Gauri

October 27, 2019

Agenda

- Conceptual Overview
  - Explicit List
  - Segregated list
  - Splitting, coalescing
  - Hints on hints
- Advanced debugging with GDB
  - Fun GDB tricks
- Writing a good heap checker
- Appendix

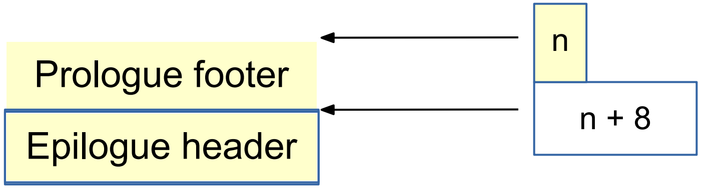
Conceptual Outline

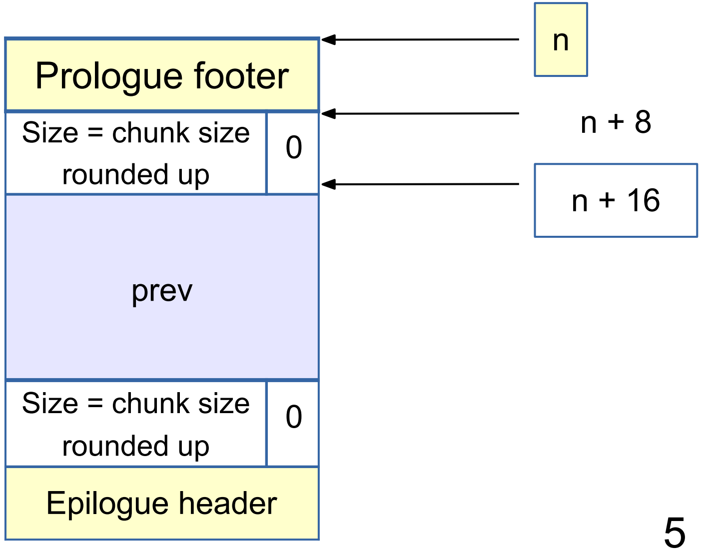
Dynamic Memory Allocation

- Used when
  - we don't know at compile-time how much memory we will need
  - when a particular chunk of memory is not needed for the entire run
    - lets us reuse that memory for storing other things
- Important terms:
  - malloc/calloc/realloc/free
  - mem_sbrk
  - payload
  - fragmentation (external vs internal)
  - Splitting / coalescing

mm_init

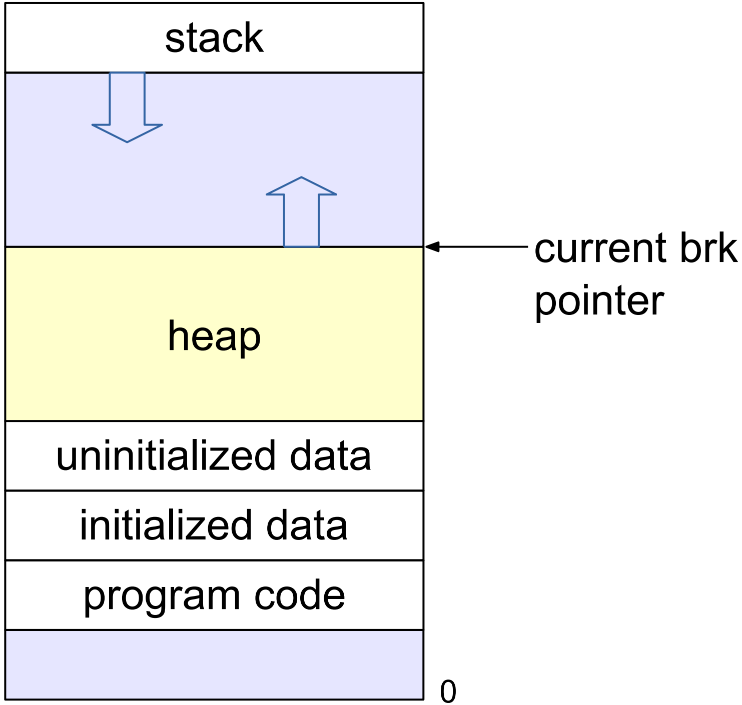
- Why prologue footer and epilogue header?
- Payload must be 16-byte aligned
- But, the size of payload doesn't have to be a multiple of 16 - just the block does!
- Things malloc'd must be within the prologue and epilogue





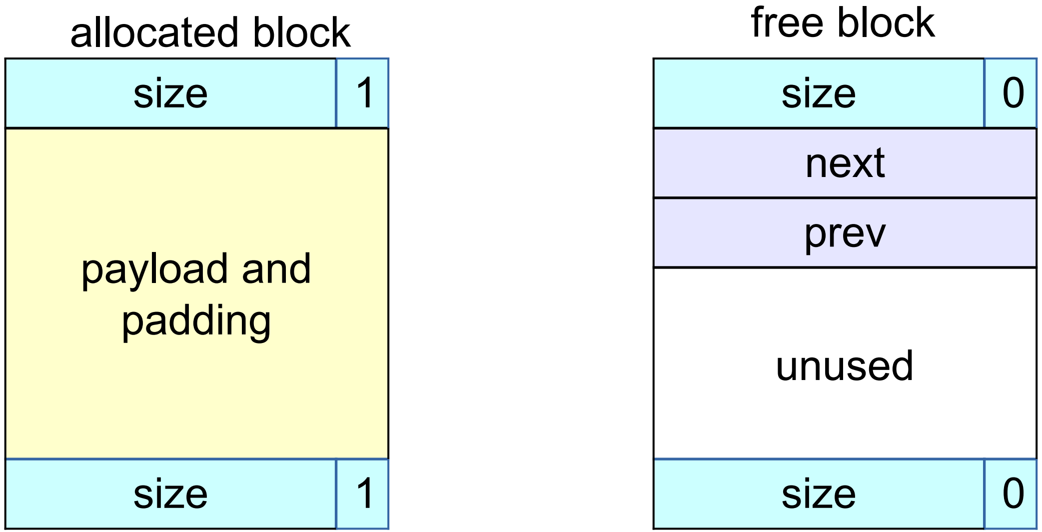
If We Can't Find a Usable Free Block

- Assume an implicit list implementation
- Need to extend the heap
  - mem_sbrk()
    - sbrk(num_bytes) allocates space and returns pointer to start
    - sbrk(0) returns a pointer to the end of the current heap
- For speed, extend the heap by a little more than you need immediately
  - use what you need out of the new space, add the rest as a free block
  - What are some tradeoffs you can



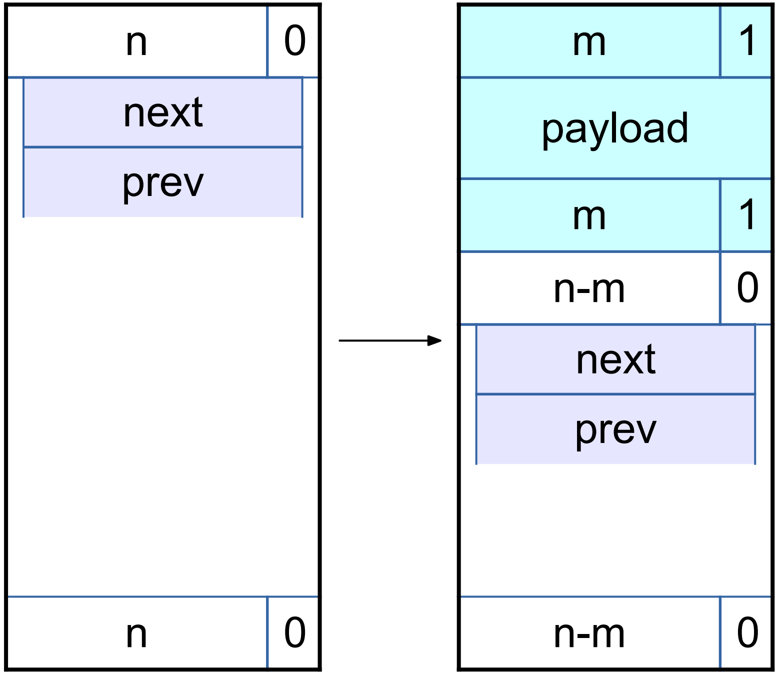
Tracking Blocks: Explicit List

- Maintain a list of *free* blocks instead of *all* blocks
  - means we need to store forward/backward pointers, not just sizes
  - we only track free blocks, so we can store the pointers in the payload area!
  - need to store size at end of block too, for coalescing



Splitting a Block

- If the block we find is larger than we need, split it and leave the remainder for a future allocation
  - explicit lists: correct previous and next pointers
  - Segregated lists: same as explicit
- When would we **not** split a block?

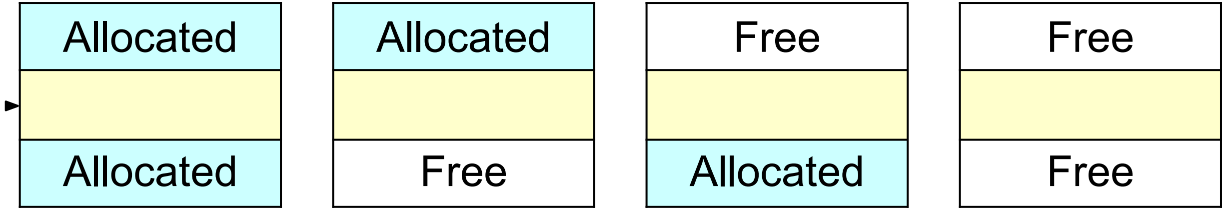


Coalescing Memory

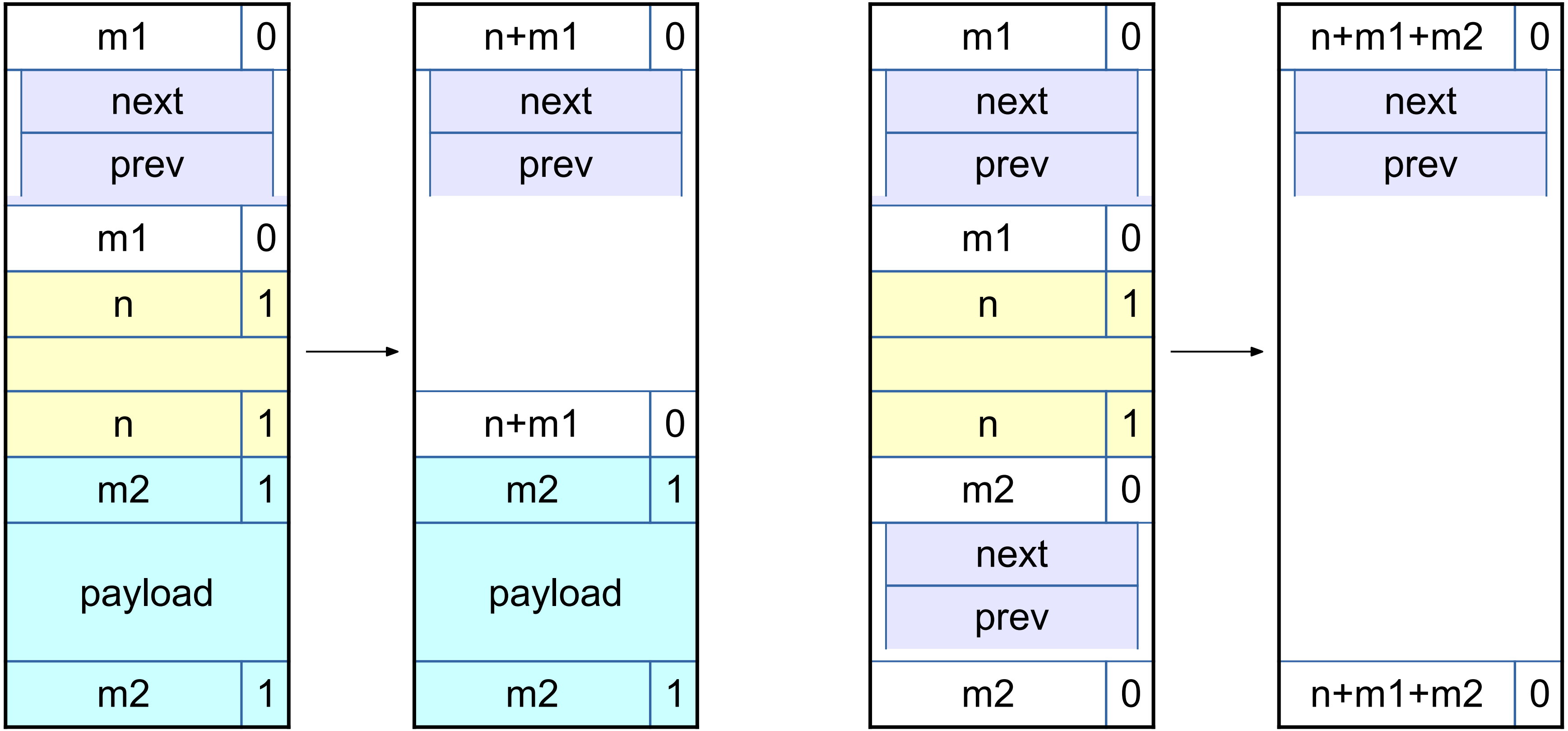
- Combine adjacent blocks if both are free
  - **explicit lists**: look forward and backward **in the heap**, using block sizes, **not** next/prev

- Four cases:

block to be freed



Coalescing Memory



Design Considerations

- Finding a matching free block
  - First fit vs. next fit vs. best fit vs. “good enough” fit
  - continue searching for a closer fit after finding a big-enough free block?
- Free block ordering
  - LIFO, FIFO, or address-ordered?
- When to coalesce
  - while freeing a block or while searching for free memory?
- How much memory to request with sbrk()
  - larger requests save time in system calls but increase maximum memory use

Hints on hints

For the final, you must greatly increase the utilization and keep a high throughput.

- Reducing external fragmentation requires achieving something closer to best-fit allocated
  - Using a better fit algorithm
  - Combine with a better data structure that lets you run more complex algorithms

- Reducing internal fragmentation requires reducing data structure overhead and using a ‘good’ free block

Segregated Lists

- Multiple explicit lists where the free blocks are of a certain size range
- Increases throughput and raises probability of choosing a better-sized block
- Need to decide what size classes (only 128 bytes of stack space)
    - Diminishing returns
    - What do you do if you can’t find something in the current size class?

- RootSizeClass1 -> free-block 1 -> free-block 2 -> free-block 3 ->
- RootSizeClass2 -> free-block 1 -> free-block 2 -> free-block 3 -> ...
- RootSizeClass3 -> free-block 1 -> free-block 2 -> free-block 3 -> ...
- ...

Modularity and Design

- Now you need to have more than one list
  - List operations are the same for all lists
    - Insert
    - Remove
  - Deciding which size class a block should go into
    - 14 if statements :(
    - A small **const** array of sizes + a loop :)
- It would be quite painful to maintain copy-pasted code
  - Abstractions are nice - it's what CS is all about!

Modularity and Design

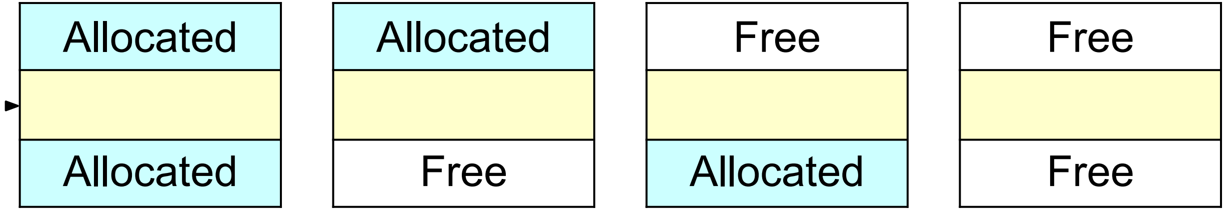
- Make sure you have modular, extensible code
  - It will save you a lot of time spent debugging and style points
  - It will make you happy when you come back to your code
    - In 6 days when you start the final submission
    - Or in 6 hours if you're missing sleep - please get some rest!
  - It will make it easier to explain to students when you become a TA later :)
- Labs in this course are NOT meant to be done in one sitting
- Labs in this course are NOT meant to be done in 2-3 nights
- Plan ahead, leave plenty of time for **design**
  - Measure twice, cut once
- Take a break between sittings
  - Your brain can keep working subconsciously
  - Leave time for “aha!” moments

Coalescing Memory

- Combine adjacent blocks if both are free
  - **segregated lists**: look forward and back using block sizes, then
    - Use the **size** of the **coalesced** block to determine the proper list
      - What else might you need to do to maintain your seglists?
    - Insert into list using the insertion policy (LIFO, address-ordered, etc.)

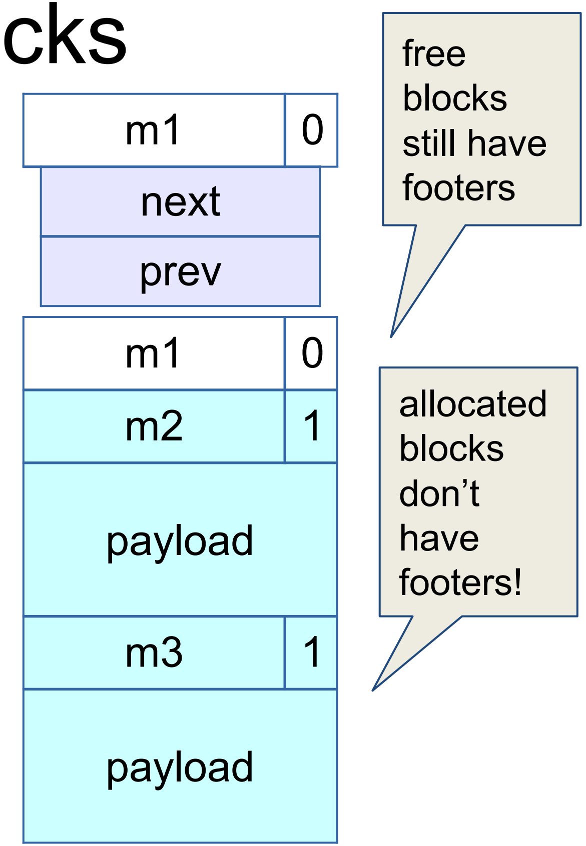
- Four cases:

block to be freed



Eliminate footers in allocated blocks

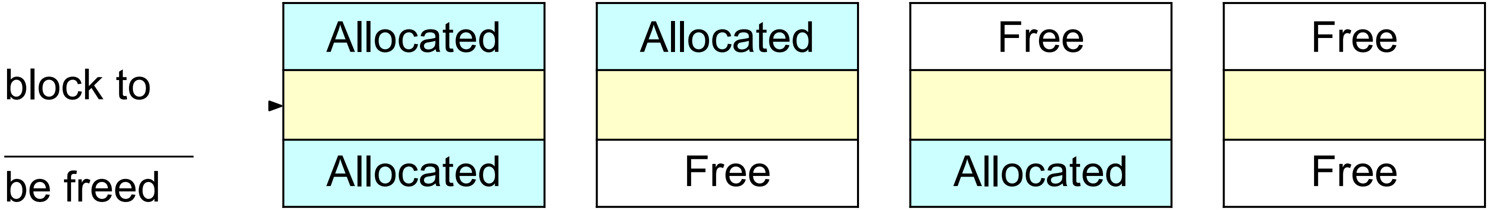
- Reduces internal fragmentation (increase utilization)
- Why do we need footers?
  - Coalescing blocks
  - What kind of blocks do we coalesce?
- Do we need to know the size of a block if we’re not going to coalesce it?
- Based on that idea, can you design a method that helps you determine when to coalesce?
  - Hint: where could you store a little **bit** of extra information for each block?



Coalescing Memory

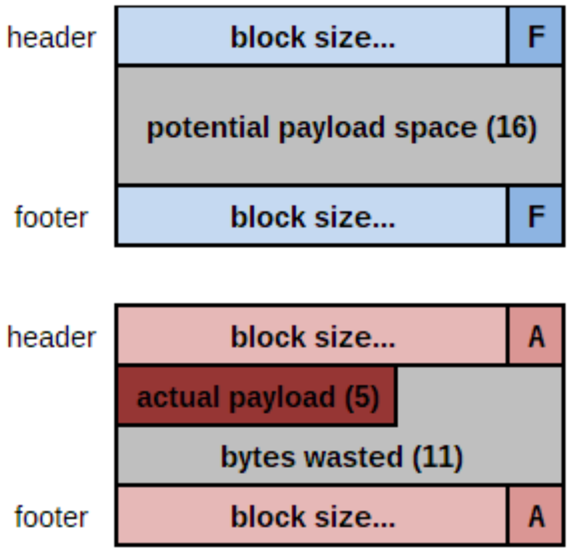
- Combine adjacent blocks if both are free
  - **footerless:** if free, obtain info from footer then use next/prev

- Four cases:



Decrease the minimum block size

- Reduces internal fragmentation (increase utilization)
- Currently, min block size is 32.
  - 8 byte header
  - 16 byte payload (or 2 8 byte pointers for free)
  - 8 byte footer
- If you just need to malloc(5), and the payload size is 16 bytes, you waste 11 bytes.
- Must manage free blocks that are too small to hold the pointers for a **doubly** linked free list



9 bytes are wasted!
How can we prevent this?

Debugging: GDB & The Almighty Heap Checker

When your scattered print statements
don't reveal where the error is

What’s better than printf? Using GDB

- Use GDB to determine where segfaults happen!
- **gdb mdriver** will open the malloc driver in gdb
  - Type run and your program will run until it hits the segfault!
- **step/next** - (abbrev. **s/n**) step to the next line of code
  - **next** steps over function calls
- **finish** - continue execution until end of current function, then break
- **print <expr>** - (abbrev. **p**) Prints **any C-like expression** (including results of function calls!)
  - Consider writing a heap printing function to use in GDB!
- **x <expr>** - Evaluate <expr> to obtain address, then examine memory at that address
  - **x /a <expr>** - formats as address
  - See **help p** and **help x** for information about more formats

Using GDB - Fun with frames

- **backtrace** - (abbrev. **bt**) print call stack up until current function
  - **backtrace full** - (abbrev. **bt full**) print local variables in each frame

(gdb) backtrace
#0 find_fit
(...)
#1 mm_malloc (...)
#2 0x0000000000403352 in
eval_mm_valid (...) #3 run_tests (...)
#4 0x0000000000403c39 in main (...)

- **frame 1** - (abbrev. **f 1**) switch to mm_malloc's stack frame
  - Good for inspecting local variables of calling functions

Using GDB - Setting breakpoints/watchpoints

- **break mm_checkheap** - (abbrev. **b**) break on “mm_checkheap()”
  - **b mm.c:25** - break on line 25 of file “mm.c” - **very useful!**
- **b find_fit if size == 24** - break on function “find_fit()” if the local variable “size” is equal to 24 - “**conditional breakpoint**”

- **watch heap_listp** - (abbrev. **w**) break if value of “heap_listp” changes - “**watchpoint**”
- **w block == 0x80000010** - break if “block” is equal to this value
- **w *0x15213** - watch for changes at memory location 0x15213
  - Can be *very* slow

- **rwatch <thing>** - stop on reading a memory location
- **awatch <thing>** - stop on *any* memory access

Heap Checker

- int mm_checkheap(int verbose);
- critical for debugging
  - **write this function early!**
  - update it when you change your implementation
  - check all heap invariants, make sure you haven't lost track of any part of your heap
    - check should pass if and only if the heap is truly well-formed
  - should only generate output if a problem is found, to avoid cluttering up your program's output
- meant to be correct, **not** efficient
- call before/after major operations **when the heap should be well-formed**

Heap Invariants (**Non-Exhaustive**)

- Block level
  - What are some things which should always be true of every block in the heap?

Heap Invariants (**Non-Exhaustive**)

- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - What are some things which should always be true of every element of a free list?

Heap Invariants (**Non-Exhaustive**)

- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - next/prev pointers in consecutive free blocks are consistent
  - no allocated blocks in free list, all free blocks are in the free list
  - no cycles in free list unless you use a circular list
  - each segregated list contains only blocks in the appropriate size class
- Heap level
  - What are some things that should be true of the heap as a whole?

Heap Invariants (**Non-Exhaustive**)

- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - next/prev pointers in consecutive free blocks are consistent
  - no allocated blocks in free list, all free blocks are in the free list
  - no cycles in free list unless you use a circular list
  - each segregated list contains only blocks in the appropriate size class
- Heap level
  - all blocks between heap boundaries, correct sentinel blocks (if used)

How to Ask for Help

- Be specific about what the problem is, and how to cause it
  - **BAD:** “My program segfaults.”
  - **GOOD:** “I ran mdriver in gdb and it says that a segfault occurred due to an invalid next pointer, so I set a watchpoint on the segfaulting next pointer. How can I figure out what happened?”
  - **GOOD:** “My heap checker indicates that my segregated list has a block of the wrong size in it after performing a coalesce(). Why might that be the case?”
  - What sequence of events do you expect around the time of the error? What part of the sequence has already happened?
- Have you written your mm_checkheap function, and is it working?
  - We **WILL** ask to see it!
- Use a rubber duck!

If You Get Stuck

- ***Please read the writeup!***
- CS:APP Chapter 9
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Friday 5:30-9:30pm in GHC 5207
- Post a **private** question on Piazza
- Obtain a rubber duck at https://tinyurl.com/malloc-f18

APPENDIX

Anonymous Structs/Unions

Same idea with unions. For the difference between unions and structs, refer to the C bootcamp slides.

```
struct A {
    int x;
    struct B {
        int y;
        float z;
    } my_b;
} my_a;
```

struct name

member name

```
struct A {
    int x;
    struct B {
        int y;
        float z;
    };
} my_a;
```

- What is the type of `x`? **`int`**
- How do we access `x` or `my_a`? **`my_a.x`**
- What is the type of `my_b`? **`struct B`**
  - **struct B**
- How do we access `y` of `my_a`? **`my_a.my_b.y`**
  - **`my_a.y`**

Zero-Length Arrays

```
struct line {
    int length;
    char contents[0];
};

int main() {
    struct line my_line;
    printf("sizeof(contents) = %zu\n",     sizeof(L.contents)); // 0
    printf("sizeof(struct line) = %zu\n", sizeof(struct line)); // 4
}
```

- It's a GCC extension - not part of the C specification!
- Must be at the end of a struct / union
- It simply allows us to represent variable-length structures
- **sizeof** on a zero-length array returns zero

Internal Fragmentation

- Occurs when the *payload* is smaller than the block size
  - due to alignment requirements
  - due to management overhead
  - as the result of a decision to use a larger-than-necessary block
- Depends on the current allocations, i.e. the pattern of *previous* requests

Internal Fragmentation

- Due to alignment requirements – the allocator doesn't know how you'll be using the memory, so it has to use the strictest alignment:
  - void *m1 = malloc(13); void *m2 = malloc(11);
  - m1 and m2 both have to be aligned on 8-byte boundaries

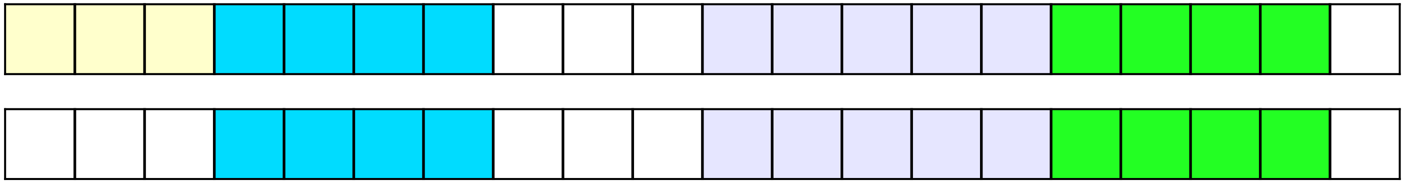
l	e	n	1	p	a	y	l	o	a	d	1	l	e	n	2	p	a	y	l	o	a	d	2

External Fragmentation

- Occurs when the total free space is sufficient, but no single free block is large enough to satisfy the request
- Depends on the pattern of *future* requests
  - thus difficult to predict, and any measurement is at best an estimate
- Less critical to malloc traces than internal fragmentation

p5 = malloc(4)

free(p1)



p6 = malloc(5)

***Oops!* Seven bytes available, but not in one chunk....**

C: Pointer Arithmetic

- Adding an integer to a pointer is different from adding two integers
- The value of the integer is always multiplied by the size of the
type that the pointer points at
- Example:
  - type_a *ptr = ...;
  - type_a *ptr2 = ptr + a;
- is really computing
  - ptr2 = ptr + (a * sizeof(type_a));
  - i.e. lea (ptr, a, sizeof(type_a)), ptr2

C: Pointer Arithmetic

- ```
  int *ptr = (int*)0x152130;
  int *ptr2 = ptr + 1;
  ```
- ```
  char *ptr = (char*)0x152130;
  char *ptr2 = ptr + 1;
  ```
- ```
  char *ptr = (char*)0x152130;
  void *ptr2 = ptr + 1;
  ```
- ```
  char *ptr = (char*)0x152130;
  char *p2 = ((char*)(((int*)ptr)+1));
  ```

C: Pointer Arithmetic

- ```
  int *ptr = (int*)0x152130;
  int *ptr2 = ptr + 1; // ptr2 is 0x152134
  ```
- ```
  char *ptr = (char*)0x152130;
  char *ptr2 = ptr + 1; // ptr2 is 0x152131
  ```
- ```
  char *ptr = (char*)0x152130;
  void *ptr2 = ptr + 1; // ptr2 is still 0x152131
  ```
- ```
  char *ptr = (char*)0x152130;
  char *p2 = ((char*)(((int*)ptr)+1));// p2 is 0x152134
  ```

Dynamic Memory Allocation: Example

p1 = malloc(3)

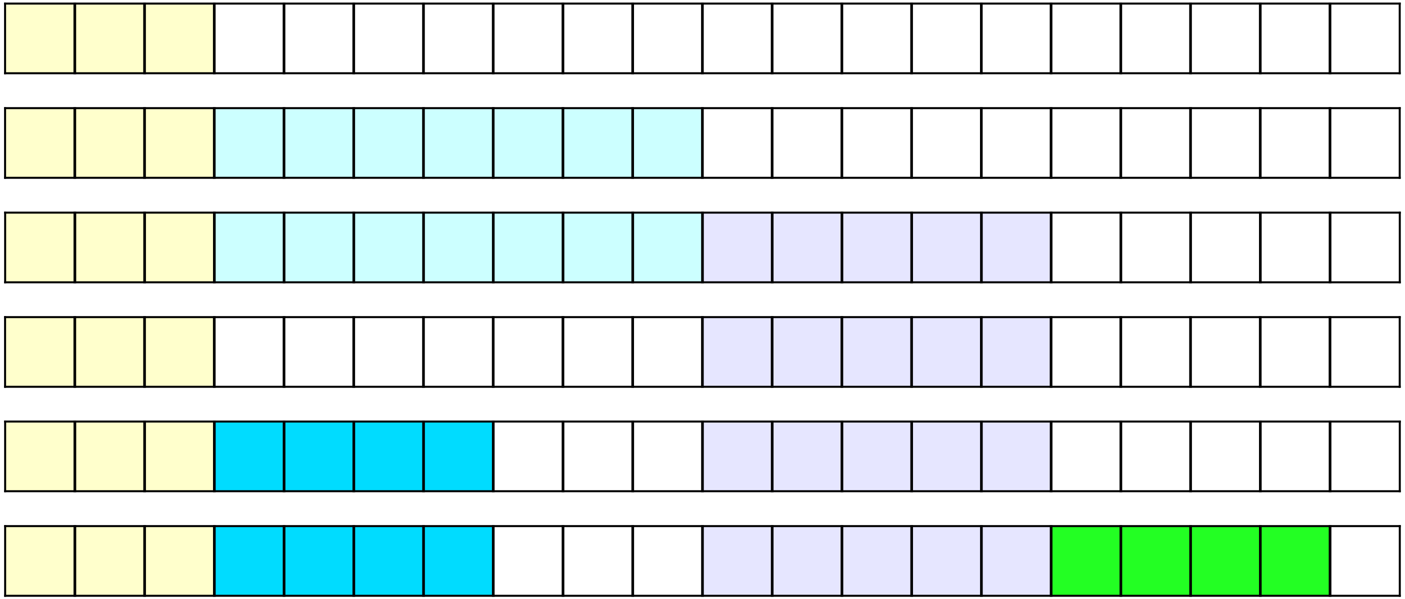
p2 = malloc(7)

p3 = malloc(5)

free(p2)

p4 = malloc(4)

p5 = malloc(4)



The Memory-Block Information Data Structure

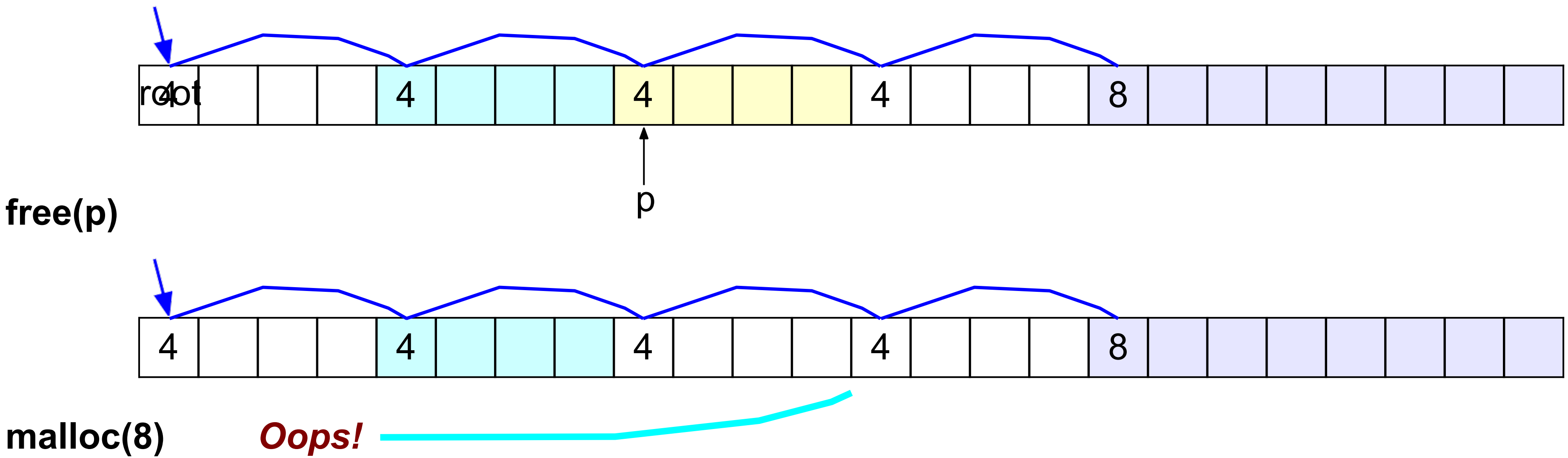
- Requirements:
  - tells us where the blocks are, how big they are, and whether they are free
  - must be able to update the data during calls to malloc and free
  - need to be able to find the **next free block** which is a “good enough fit” for a given payload
  - need to be able to quickly mark a block as free or allocated
  - need to be able to detect when we run out of blocks
    - what do we do in that case?
- The only memory we have is what we're handing out
  - ...but not all of it needs to be payload! We can use part of it to store the block information.

Finding a Free Block

- First Fit
  - search from beginning, use first block that's big enough
  - linear time in total number of blocks
  - can cause small “splinters” at beginning of list
- Next Fit
  - start search from where previous search finished
  - often faster than first fit, but some research suggests worse fragmentation
- Best Fit
  - search entire list, use smallest block that's big enough
  - keeps fragments small (less wasted memory), but slower than first fit

Freeing Blocks

- Simplest implementation is just clearing the "allocated" flag
  - but leads to external fragmentation



Insertion Policy

- Where do you put a newly-freed block in the free list?
  - LIFO (last-in-first-out) policy
    - add to the beginning of the free list
    - pro: simple and constant time (very fast)

    block->next = freelist; freelist = block;
    - con: studies suggest fragmentation is worse
  - Address-ordered policy
    - insert blocks so that free list blocks are always sorted by address

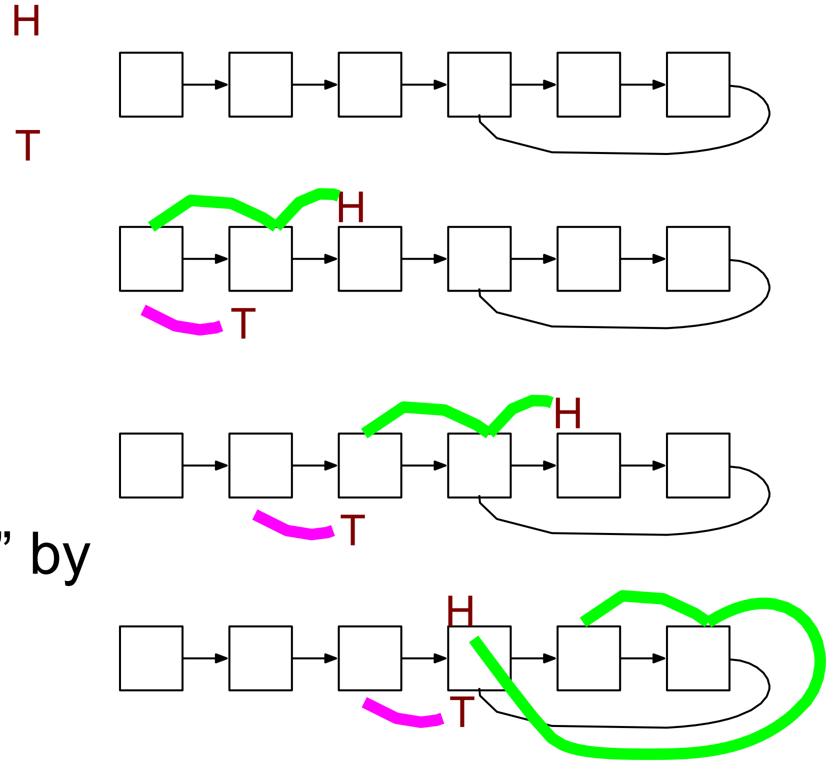
    addr(prev) < addr(curr) < addr(next)
    - pro: lower fragmentation than LIFO
    - con: requires search

C: Pointer Casting

- Notation: `(b*)a` "casts" `a` to be of type `b*`
- Casting a pointer doesn't change the bits!
  - `type_a *ptr_a=...; type_b *ptr_b=(type_b*)ptr_a;` makes ptr_a and ptr_b contain identical bits
- But it does change the behavior when dereferencing
  - because we *interpret* the bits differently
- Can cast `type_a*` to long/unsigned long and back
  - pointers are really just 64-bit numbers
  - such casts are important for malloclab
  - but be careful – this can easily lead to hard-to-find errors

Cycle Checking: Hare and Tortoise Algorithm

- This algorithm detects cycles in linked lists
- Set two pointers, called “hare” and “tortoise”, to the beginning of the list
- During each iteration, move “hare” forward by two nodes, “tortoise” by one node
  - if “tortoise” reaches the end of the list, there is no cycle
  - if “tortoise” equals “hare”, the list has a cycle



Debugging Tip: Using the Preprocessor

- Use conditional compilation with #if or #ifdef to easily turn debugging code on or off

```
#ifdef DEBUG
#define DBG_PRINTF(...)    fprintf(stderr, __VA_ARGS__)
#define CHECKHEAP(verbose) mm_checkheap(verbose)
#else
#define DBG_PRINTF(...)
#define CHECKHEAP(verbose)
#endif /* DEBUG */
```

```
//  comment line below to disable debug code!
#define DEBUG

void free(void *p) {
   DBG_PRINTF("freeing %p\n", p);
   CHECKHEAP(1);
   ...
}
```

Debugging Tip: Using the Preprocessor (contd)

```
#define DEBUG

void free(void *p) {
    DBG_PRINTF("freeing %p\n", p);
    CHECKHEAP(1);
    ...
}
```

preprocessor magic

```
void free(void *p) {
    fprintf(stderr, "freeing %p\n", p);
    mm_checkheap(1);
    ...
}
```

Replaced with debug code!

```
//  #define DEBUG

void free(void *p) {
    DBG_PRINTF("freeing %p\n", p);
    CHECKHEAP(1);
    ...
}
```

preprocessor magic

```
void free(void *p) {
    ...
}
```

Debug code gone!

Header Reduction

- **Note:** this is completely optional and generally **discouraged** due to its relative difficulty
    - Do **NOT** attempt unless you are satisfied with your implementation as-is

- When to use 8 or 4 byte header? (must support all possible block sizes)
- If 4 byte, how to ensure that payload is aligned?
- Arrange accordingly
- How to coalesce if 4 byte header block is followed by 8 byte header block?
- Store extra information in headers