

# Malloc Bootcamp

Shankara, Cindy, Jingwei

July 5th, 2022

# Agenda

- Recap on fragmentation, structs & unions
- Modularity and Design
- Increasing Utilization
  - Eliminating footers
  - Decreasing minimum block size
  - Other improvements

# Conceptual Outline

Me: \*recompiles code I  
know damn well I didn't change\*

\*code breaks\*

Also me:

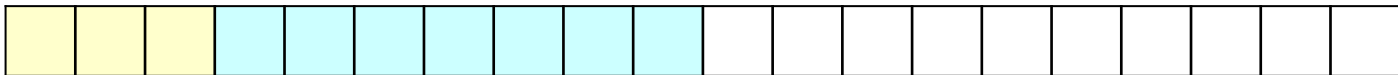


# Dynamic Memory Allocation: Example

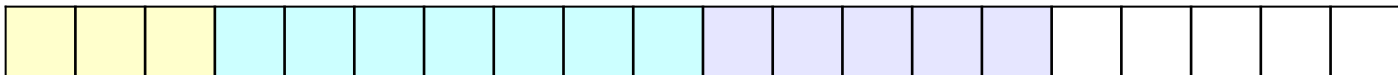
p1 = malloc(3)



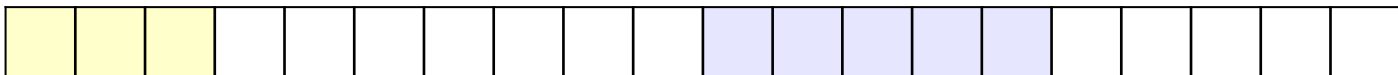
p2 = malloc(7)



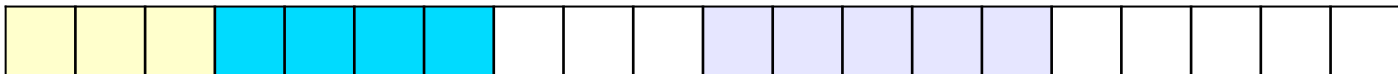
p3 = malloc(5)



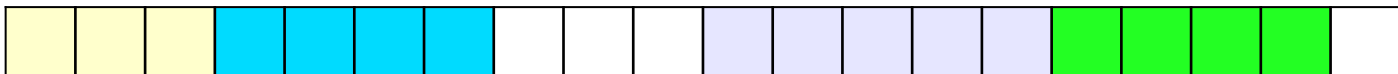
free(p2)



p4 = malloc(4)



p5 = malloc(4)



# Dynamic Memory Allocation

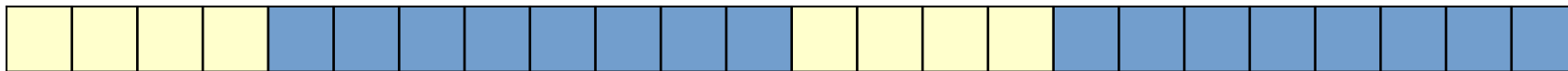
- Used when
  - need lot of memory
  - memory must live after the function return
  - we don't know at compile-time how much memory we will need
- Important terms:
  - malloc/calloc/realloc/free
  - mem\_sbrk
  - payload
  - fragmentation (external vs internal)
  - Splitting / coalescing

# 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 want to know how you'll be using the memory so it can stay efficient, so uses the strictest alignment:
  - `void *m1 = malloc(13); void *m2 = malloc(11);`
  - `m1` and `m2` both have to be aligned on 8-byte boundaries



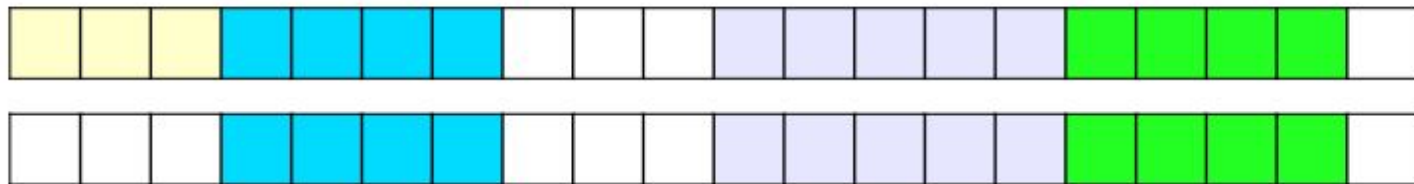
# External Fragmentation

- 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....**



# 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 {
        int y;
        float z;
    };
} my_a;
```

- What is the type of `x`?
- How do we access `x` or `my_a`?
- What is the type of `my_b`?
- How do we access `y` of `my_a`?

`int`

`my_a.x`

`struct B`

`my_a.my_b.y`

`my_a.y`

```
/** @brief Represents the header and payload of one block in the heap */
typedef struct block {
    /** @brief Header contains size + allocation flag */
    word_t header;

    /**
     * @brief A pointer to the block payload.
     *
     * TODO: feel free to delete this comment once you've read it carefully.
     * We don't know what the size of the payload will be, so we will declare
     * it as a zero-length array, which is a GCC compiler extension. This will
     * allow us to obtain a pointer to the start of the payload.
     *
     * WARNING: A zero-length array must be the last element in a struct, so
     * there should not be any struct fields after it. For this lab, we will
     * allow you to include a zero-length array in a union, as long as the
     * union is the last field in its containing struct. However, this is
     * compiler-specific behavior and should be avoided in general.
     *
     * WARNING: DO NOT cast this pointer to/from other types! Instead, you
     * should use a union to alias this zero-length array with another struct,
     * in order to store additional types of data in the payload memory.
     */
    char payload[0];

    /**
     * TODO: delete or replace this comment once you've thought about it.
     * Why can't we declare the block footer here as part of the struct?
     * Why do we even have footers -- will the code work fine without them?
     * which functions actually use the data contained in footers?
     */
} block_t;
```

# 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
}
```

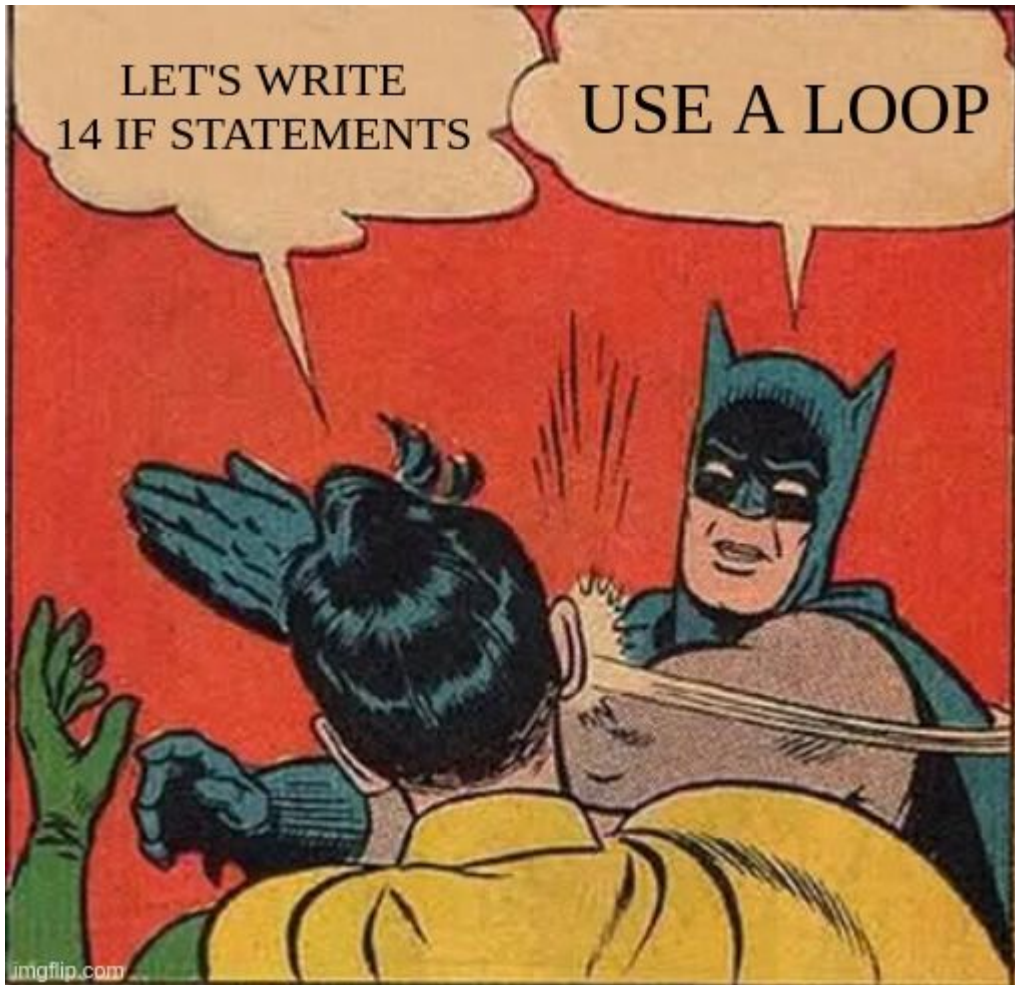
- It's a GCC extension - not part of the C specification!
- Must be at the end of a struct / union
- It simply allows us to represent variable-length structures
- `sizeof` on a zero-length array returns zero

# 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
  - 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

LET'S WRITE  
14 IF STATEMENTS

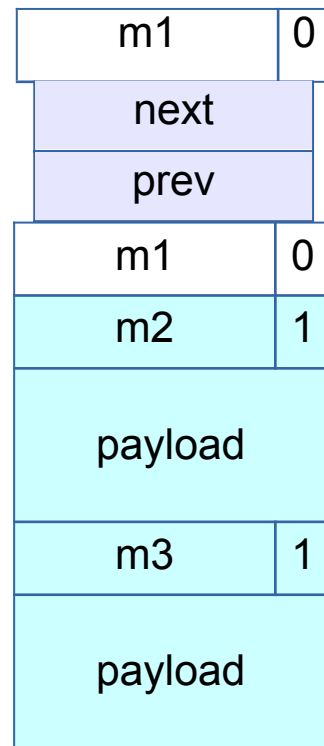
USE A LOOP



# Eliminate footers in allocated blocks

Reduces internal fragmentation (increases 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?

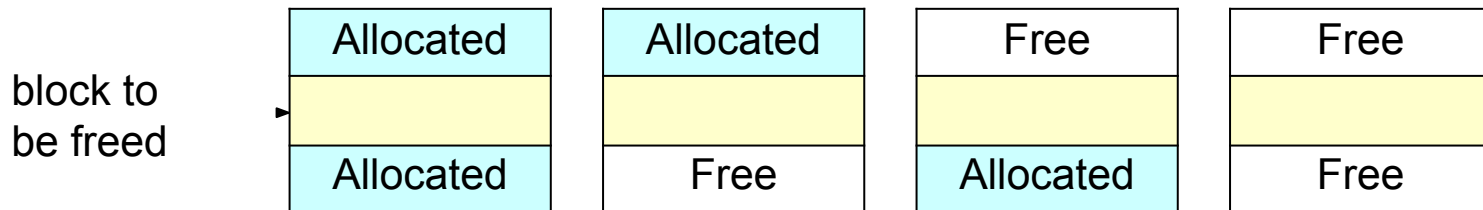


free  
blocks  
still have  
footers

allocated  
blocks  
don't  
have  
footers!

# Coalescing Memory

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



# Pitfall: Minimum Block Size

NOTE: The minimum block size does not become 16 or 24 when you remove footers- it remains at 32!

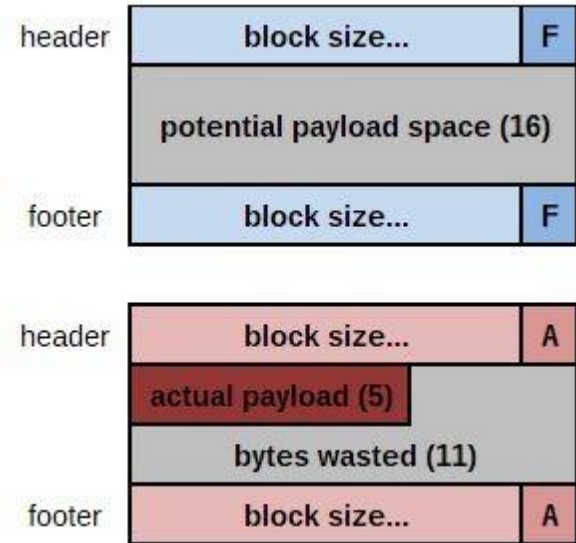
Please ask questions now if you are confused!



# 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

A lot of bytes are wasted!  
How can we prevent this?



# Decrease the minimum block size

HINT: Your minimum block size should be 16 in order to pass final, meaning you only keep 2 of the fields that we had before. Which two fields you keep though is for you to think about!

# Small utilization improvements

- Insertion Policy
  - LIFO (last-in-first-out) vs FIFO (first-in-first-out)
- Segregated List Buckets
  - Potentially reconsider size classes (only 128 bytes of stack space)
    - Diminishing returns
    - Adjust buckets based trace files (please don't hard code)
- Chunksize
  - Potentially reconsider smaller size
- Fit Algorithm
  - First-fit
  - Best-fit (which segregated list approximates)
  - Better Fit (ex. search for the next 20 blocks after finding a fit)

# 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

# Ways to Improve

<b>Optimization</b>	<b>Utilization</b>	<b>Throughput</b>
Implicit List (Starter Code)	59%	10-100
Explicit Free List	- <sup>3</sup>	2000-5000
Segregated Free Lists	-	11000
Better Fit Algorithm	59%	Variable
Eliminating Footers in Allocated Blocks	+9%	-
Decreasing Block Size/Mini Blocks	+6%	-20%
Compressing Headers	+2%	-

source: writeup

# 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

# 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 for 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 for every element of a free list?

# Heap Invariants (**Non-Exhaustive**)

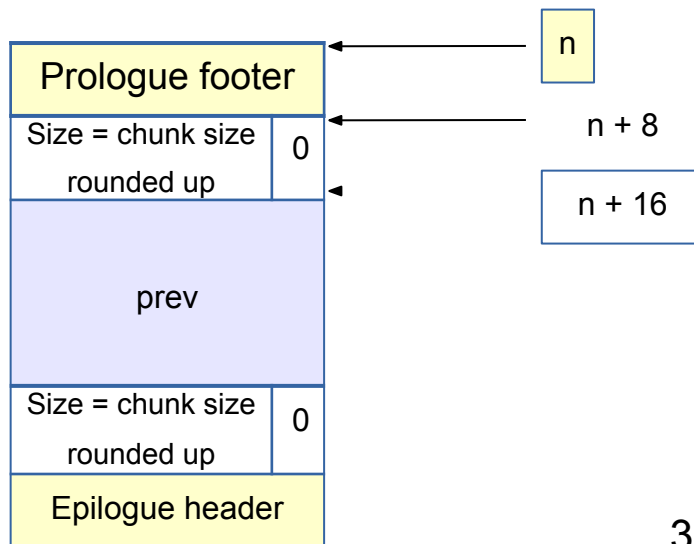
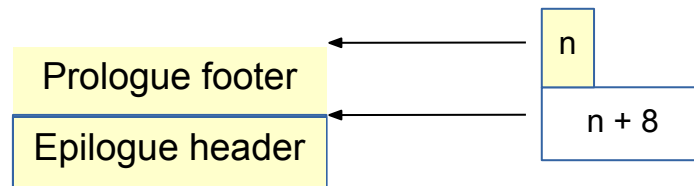
- Block level
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- List level
  - next/prev pointers in consecutive free blocks are consistent
  - no allocated blocks in free list, all free blocks are in the free list
  - no 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 for 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

# mm\_init

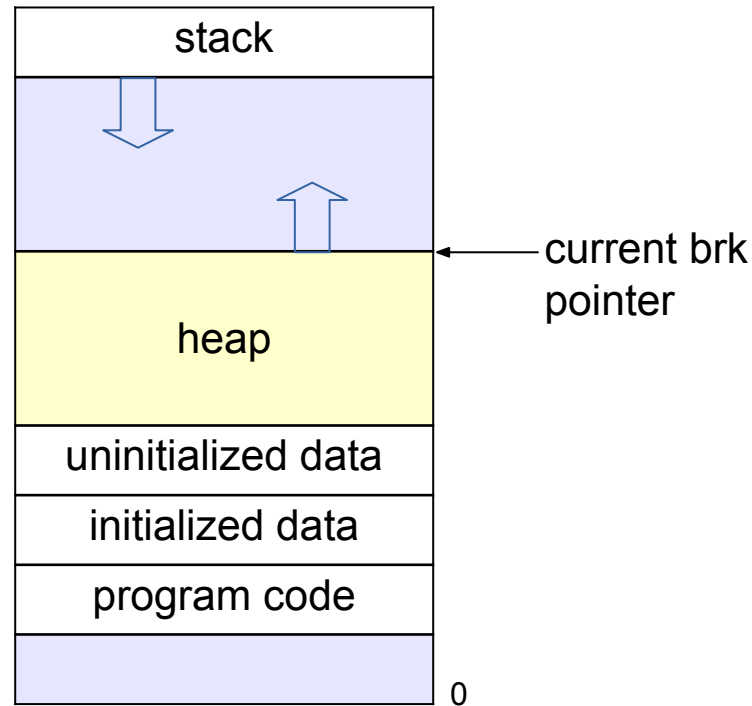
- Why prologue footer and epilogue header?
- Payload must be 16-byte aligned
- But, the size of payload doesn't have to be a multiple of 16 - just the block does!
- Things malloc'd must be within the prologue and epilogue





# 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 make?



# 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 - needs to be updated

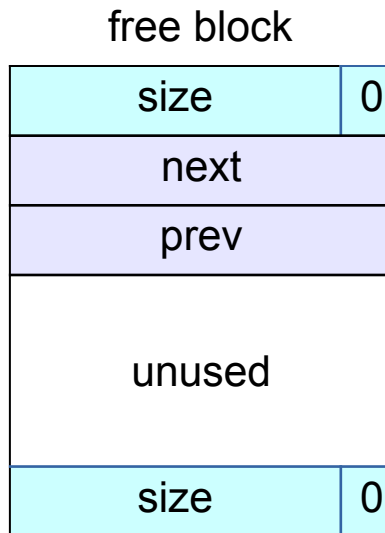
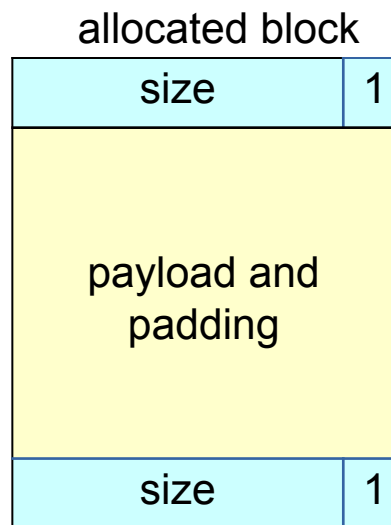
## ■ *Please read the writeup!*

- 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 over Zoom. See pinned Piazza post!
- Post a **private** question on Piazza
- Obtain a rubber duck at <https://tinyurl.com/malloc-f18>

# APPENDIX

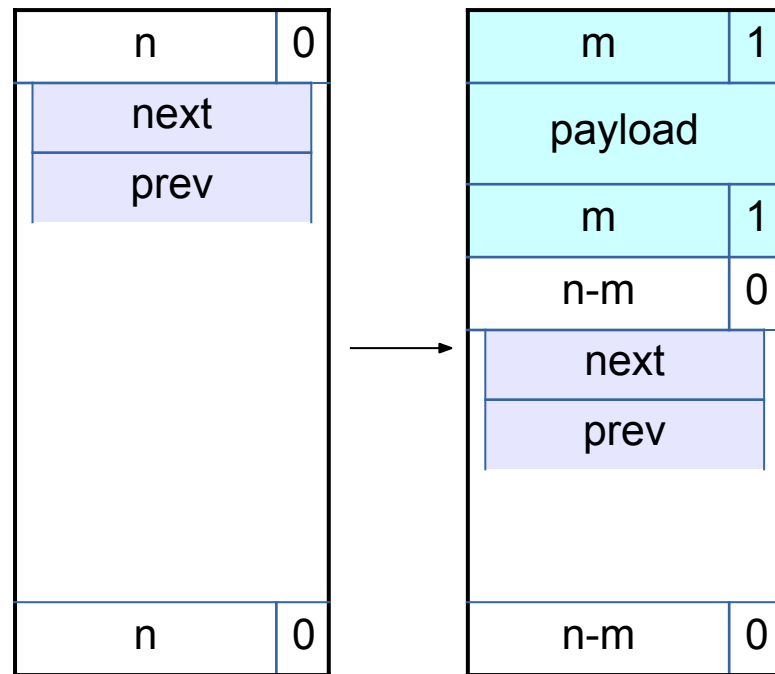
# Tracking Blocks: Explicit List

- 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

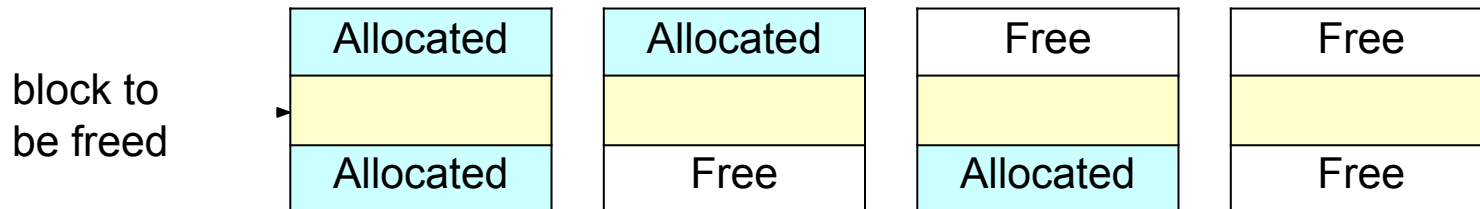
- If the block we find is larger than we need, split it and leave the remainder for a future allocation
  - explicit lists: correct previous and next pointers
  - Segregated lists: same as explicit
- When would we **not** split a block?



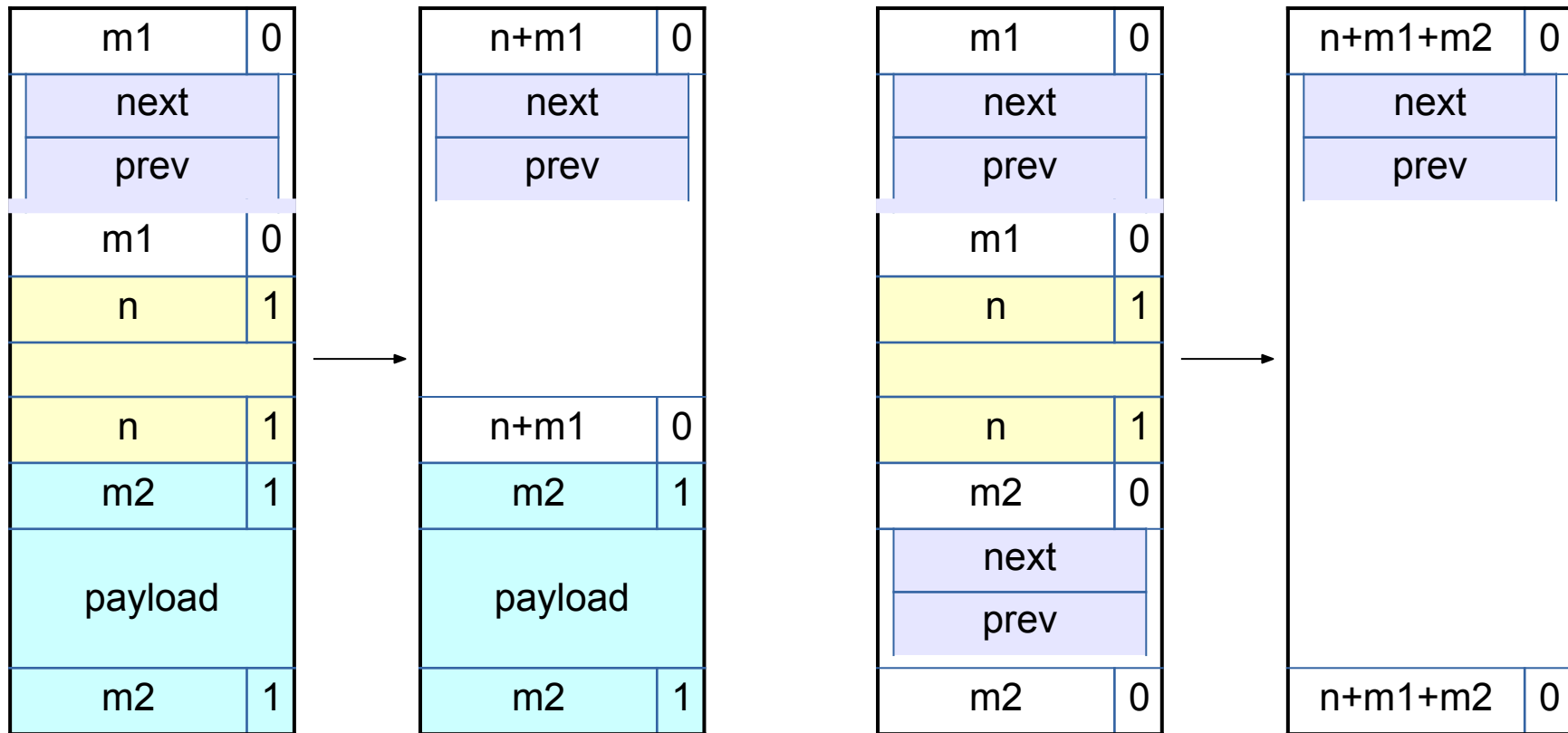
# Coalescing Memory

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

- Four cases:



# Coalescing Memory



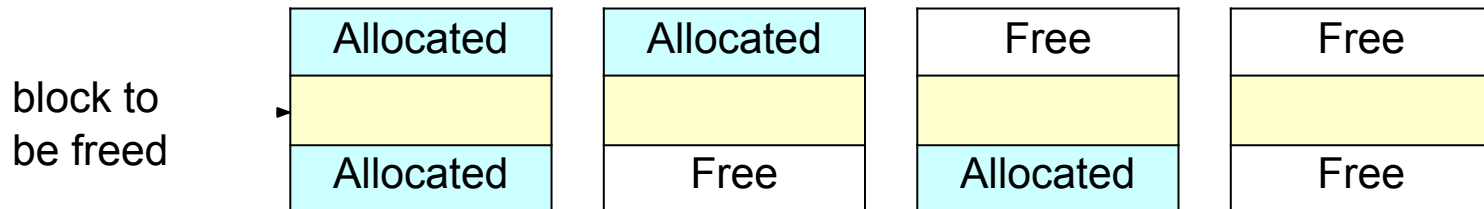


# Segregated Lists

- 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 -> ...
- ...

# 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:

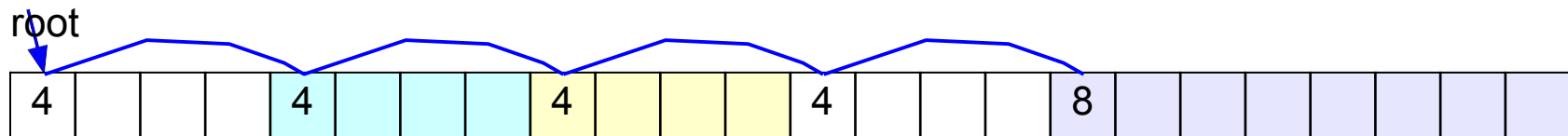


# Finding a Free Block

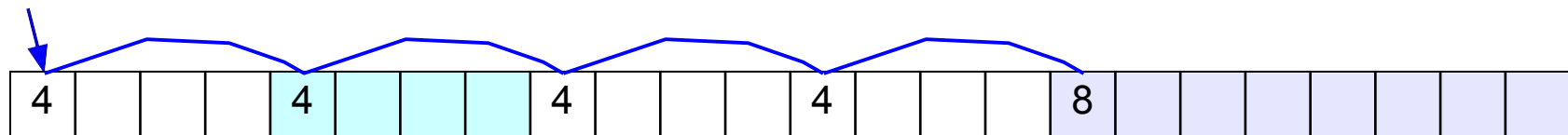
- First Fit
  - search from beginning, use first block that's big enough
  - linear time in total number of blocks
  - can cause small “splinters” at beginning of list
- Next Fit
  - start search from where previous search finished
  - often faster than first fit, but some research suggests worse fragmentation
- 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 if not coalesced



free(p)



malloc(8)

*Oops!*

# Memory-Block Information

- 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.

# 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  
 $\text{addr}(\text{prev}) < \text{addr}(\text{curr}) < \text{addr}(\text{next})$
    - pro: lower fragmentation than LIFO
    - con: requires search

# 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`
- Pointer arithmetic on `void*` is undefined (what's the size of a void?) 42

# 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

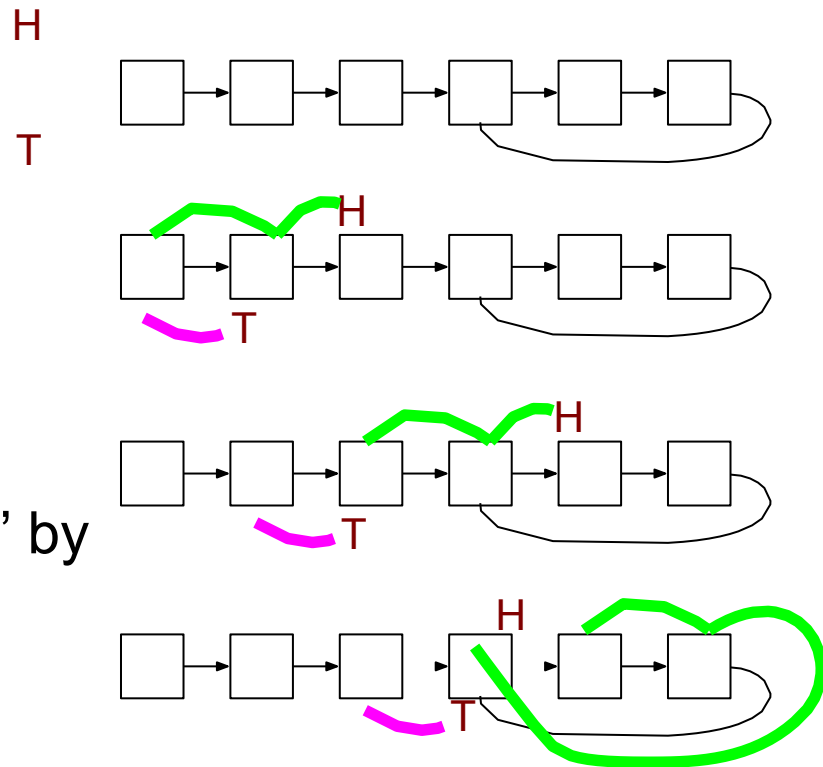
- `int *ptr = (int*)0x152130;`  
`int *ptr2 = ptr + 1; // ptr2 is 0x152134`
- `char *ptr = (char*)0x152130;`  
`char *ptr2 = ptr + 1; // ptr2 is 0x152131`
- `char *ptr = (char*)0x152130;`  
`void *ptr2 = ptr + 1; // ptr2 is still 0x152131`
- `char *ptr = (char*)0x152130;`  
`char *p2 = ((char*)((int*)ptr)+1);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

# 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 mallocclab
  - 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 debugging code!
#define DEBUG

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

# Debugging Tip: Using the Preprocessor

```
#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);  
    ...  
}
```

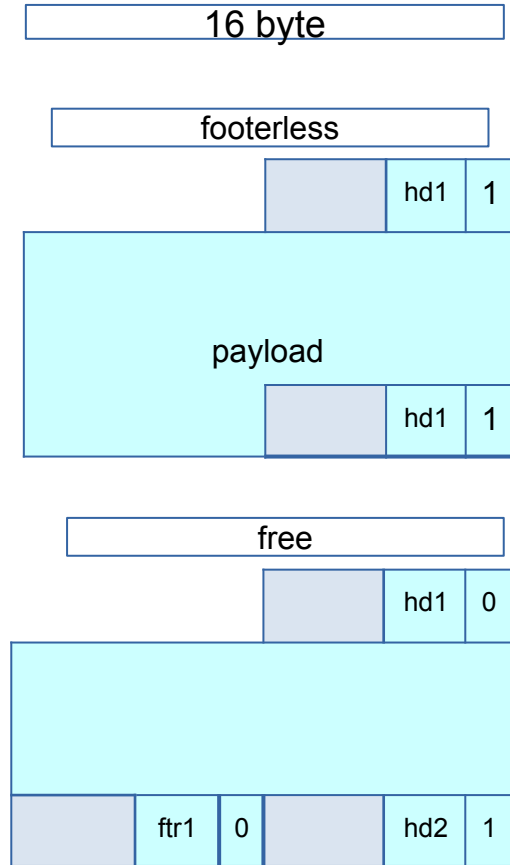
*preprocessor magic* →

```
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



# What's better than GDB? Using CGDB!

- CGDB is just like GDB
  - but with **COLOR** and **SOURCE\_CODE**
- Breaking at `mm_malloc` in GDB vs CGDB

```

gdb mdriver-dbg
5% 21 GB 3/19, 12:07 PM
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to copy and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/andrew.cmu.edu/usr10/jmcamero/private/213_testing/malloblab-s20/mdriver-dbg...done.
(gdb) b mm_malloc
Breakpoint 1 at 0x40441c: file mm.c, line 580.
(gdb) -f traces/syn-array-short.rep
Undefined command: "-f". Try "help".
(gdb) r -f traces/syn-array-short.rep
Starting program: /afs/andrew.cmu.edu/usr10/jmcamero/private/213_testing/malloblab-s20/mdriver-dbg -f traces/syn-array-short.rep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Found benchmark throughput 8009 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark regular
Throughput targets: min=4004, max=7208, benchmark=8009

Breakpoint 1, mm_malloc (size=9904) at mm.c:580
580     dbg_requires(mm_checkheap(__LINE__));
(gdb)
    
```

Colors!

C Code!

GDB terminal!

```

cgdb mdriver-dbg
6% 21 GB 3/19, 12:09 PM
574 * <Are there any preconditions or postconditions?>
575 *
576 * @param[in] size
577 * @return
578 */
579 void *malloc(size_t size) {
580     dbg_requires(mm_checkheap(__LINE__));
581
582     size_t asize; // Adjusted block size
583     size_t extendsize; // Amount to extend heap if no fit is found
584     block_t *block;
585     void *bp = NULL;
/afs/andrew.cmu.edu/usr10/jmcamero/private/213_testing/malloblab-s20/mm.c
r -f traces/syn-struct.rep
(gdb) r -f traces/syn-mix-short.rep
Starting program: /afs/andrew.cmu.edu/usr10/jmcamero/private/213_testing/malloblab-s20/mdriver-dbg -f traces/syn-mix-short.rep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Found benchmark throughput 8009 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark regular
Throughput targets: min=4004, max=7208, benchmark=8009

Breakpoint 1, mm_malloc (size=9904) at mm.c:580
(gdb)
    
```

# Using CGDB

- Initializes the same as GDB
  - Just write `cgdb mdriver-dbg` instead of `gdb mdriver-dbg`
- Source and gdb windows
  - To go from gdb (default) to source press `esc` (just like normal mode in vim!!!!!!)
  - to go from source to gdb press `i` (just like insert mode in vim!!!!!!)

Source

GDB

```
cgdb mdriver-dbg
7% 21 GB 3/19, 12:16 PM
591
592 // Ignore spurious request
593 if (size == 0) {
594     dbg_ensures(mm_checkheap(__LINE__));
595     return bp;
596 }
597
598 // Adjust block size to include overhead and to meet alignment requirem
599 asize = round_up(size + dsize, dsize);
600
601 // Search the free list for a fit
602 -> block = find_fit(asize);
603
604 // If no fit is found, request more memory, and then and place the bloc
605 if (block == NULL) {
606     // Always request at least chunksize
607     extendsize = max(asize, chunksize);
608     block = extend_heap(extendsize);
609     // extend_heap returns an error
610     if (block == NULL) {
611         return bp;
612     }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }
```



# Source Mode

## • Benefits

- see breakpoints in the file
- reread code while debugging
- review what is coming


## • Usage

- Very similar to vim!!
- j - move down a line
- k - move up a line
- :### jump to line ###
- /### search for ###
- Most other vim commands!

## • Notes

- Green line number is current line
- Red line number is breakpoint


Current Line  
to execute




Viewed line



Next  
Breakpoint



Breakpoint at  
mm.c 610



```
cgdb mdriver-dbg
6% 22 GB 3/19, 12:31 PM
595     return bp;
596     }
597
598     // Adjust block size to include overhead and to meet alignment requirements
599     asize = round_up(size + dsize, dsize);
600
601     // Search the free list for a fit
602 -> block = find_fit(asize);
603
604     // If no fit is found, request more memory, and then and place the block
605     if (block == NULL) {
606         // Always request at least chunksize
607         extendsize = max(asize, chunksize);
608         block = extend_heap(extendsize);
609         // extend_heap returns an error
610         if (block == NULL) {
611             return bp;
612         }
613     }
614
615     // The block should be marked as free
616     dbg_assert(!get_alloc(block));
617
/afs/andrew.cmu.edu/usr10/jmcamera/private/213_testing/malloclab-s20/mm.c
r -f traces/syn-string.rep
r -f traces/syn-struct-scaled.rep
r -f traces/syn-struct-short.rep
r -f traces/syn-struct.rep
(gdb) r -f traces/syn-mix-short.rep
Starting program: /afs/andrew.cmu.edu/usr10/jmcamera/private/213_testing/malloclab-s20/mdriver-dbg -f traces/syn-mix-short.rep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Found benchmark throughput 8009 for cpu type Intel(R)Xeon(R)CPU E5520@2.27GHz, benchmark regular
Throughput targets: min=4004, max=7208, benchmark=8009

Breakpoint 1, mm_malloc (size=9904) at mm.c:580
(gdb) n
(gdb)
(gdb)
(gdb)
(gdb)
(gdb) b mm.c:610
Breakpoint 2 at 0x40452f: file mm.c, line 610.
(gdb)
```

# CGDB Misc

- GDB mode functions exactly like normal GDB!
  - All the commands you know and love work the same!
- Shark machines use version 0.6.8
  - Means unfortunately no assembly viewer :( though that is not often needed.
- Website
  - <https://cgdb.github.io/>
- Documentation
  - <https://cgdb.github.io/docs/cgdb.pdf>
  - Version 0.7.1