Recitation 7: Malloc Lab (Checkpoint)

Your TAs

Monday, March 13th, 2023

Logistics

- **Malloc checkpoint due <u>March 21</u>**
- **Malloc final due <u>March 28</u>**
- **Malloc Final Bootcamp <u>March 19th 5-7 pm</u>** (recommended)
- **PLEASE START EARLY!**
- **WRITE CHECKHEAP OR NO OH HELP!**

Checkpoint Submission

- **Style Grading**
  - We will grade your checkheap with your checkpoint submission!

- **Things to Remember:**
  - Document checkheap
  - See writeup for what to include in checkheap

Git Reminders

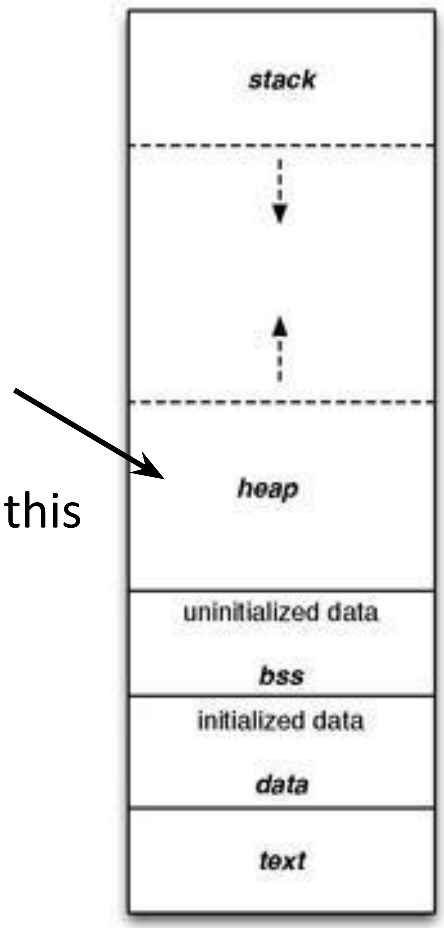
- **Code Review Grades for Cachelab will out by next week**
  - Please use detailed commit messages – things like “DONE” or “did a thing” aren’t enough
  - You should be committing often as you work on your code
    - Especially for malloc: `git diff` can show what you changed since your last working commit
  - Also allows you to restore your hard work in case your file gets deleted accidentally…
- **Commit early, commit often** 😤

Outline

- Concept
- How to choose blocks
- Metadata
- Debugging / GDB Exercises

What is malloc?

- **A function to allocate memory during runtime (dynamic memory allocation).**
  - More useful when the size or number of allocations is unknown until runtime (e.g., data structures)
- **The heap is a segment of memory addresses reserved almost exclusively for malloc to use.**
  - Your code directly manipulates the bytes of memory in this section.



Concept

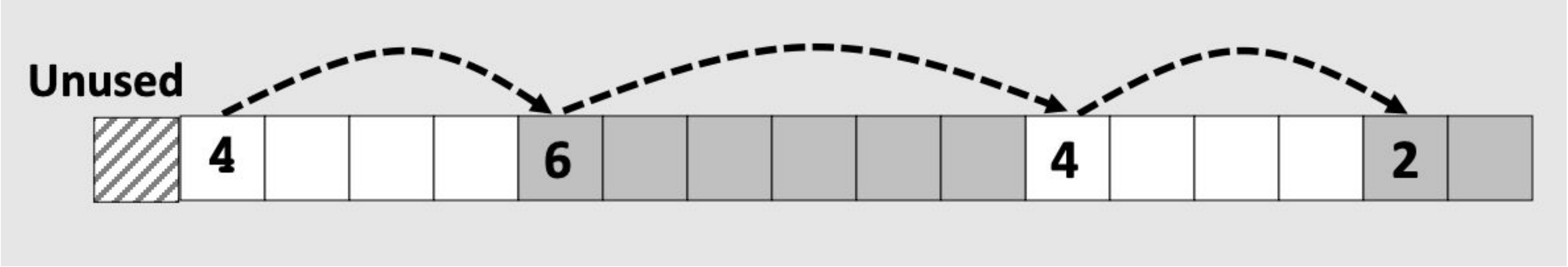
- Overall, malloc does three things:

1. Organizes all blocks and stores information about them in a structured way.
2. Uses the structure made to choose an appropriate location to allocate new memory.
3. Updates the structure when the user frees a block of memory.

This process occurs even for a complicated algorithm like segregated lists.

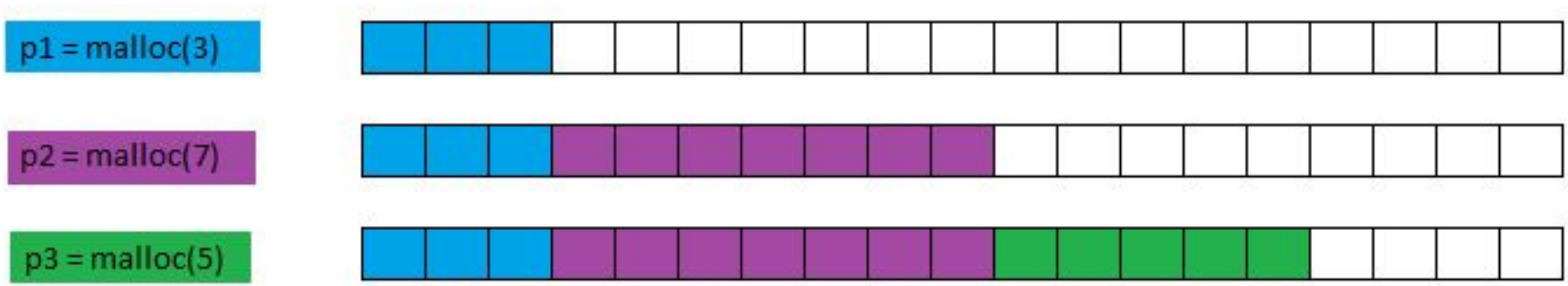
Concept (Implicit list)

1. Connects and organizes all blocks and stores information about them in a structured way, typically implemented as a singly linked list



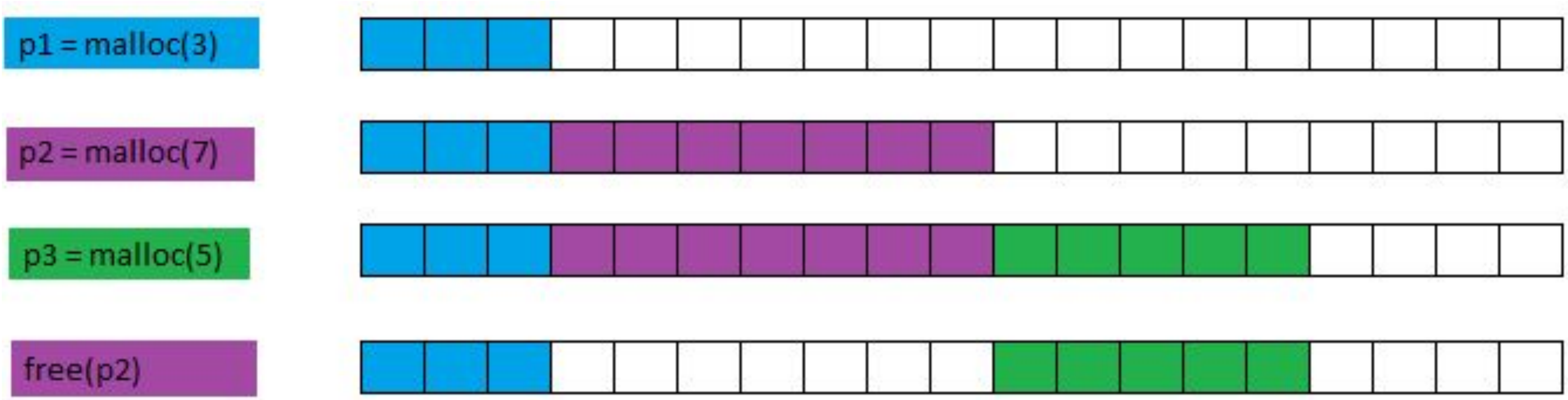
Concept (Implicit list)

2. **Uses the structure made to choose an appropriate location to allocate new memory.**



Concept (Implicit list)

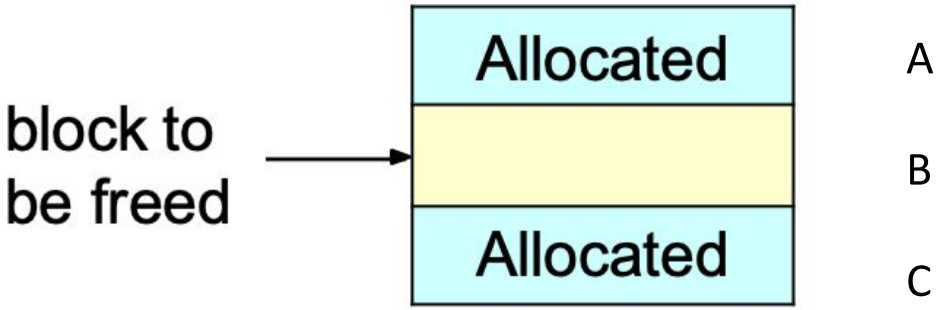
3. **Updates the structure when the user frees a block of memory.**



Concept (Implicit list)

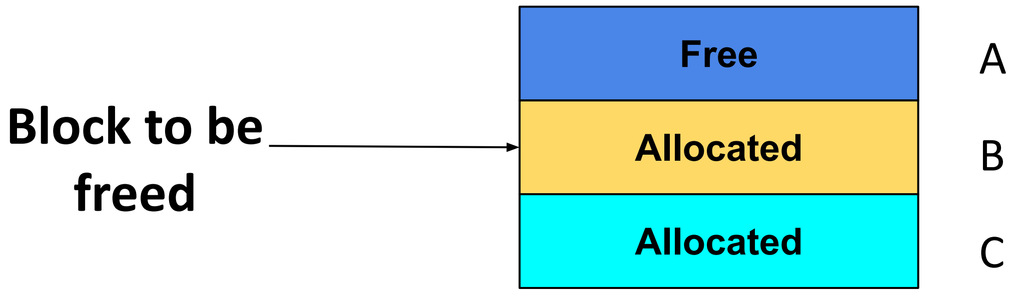
3. **Updates the structure when the user frees a block of memory.**

Coalesce: Case 1

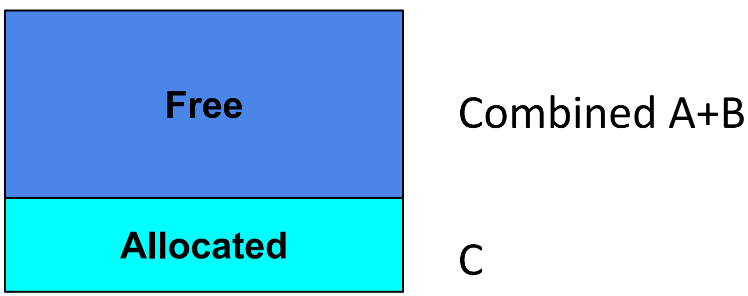


Result:

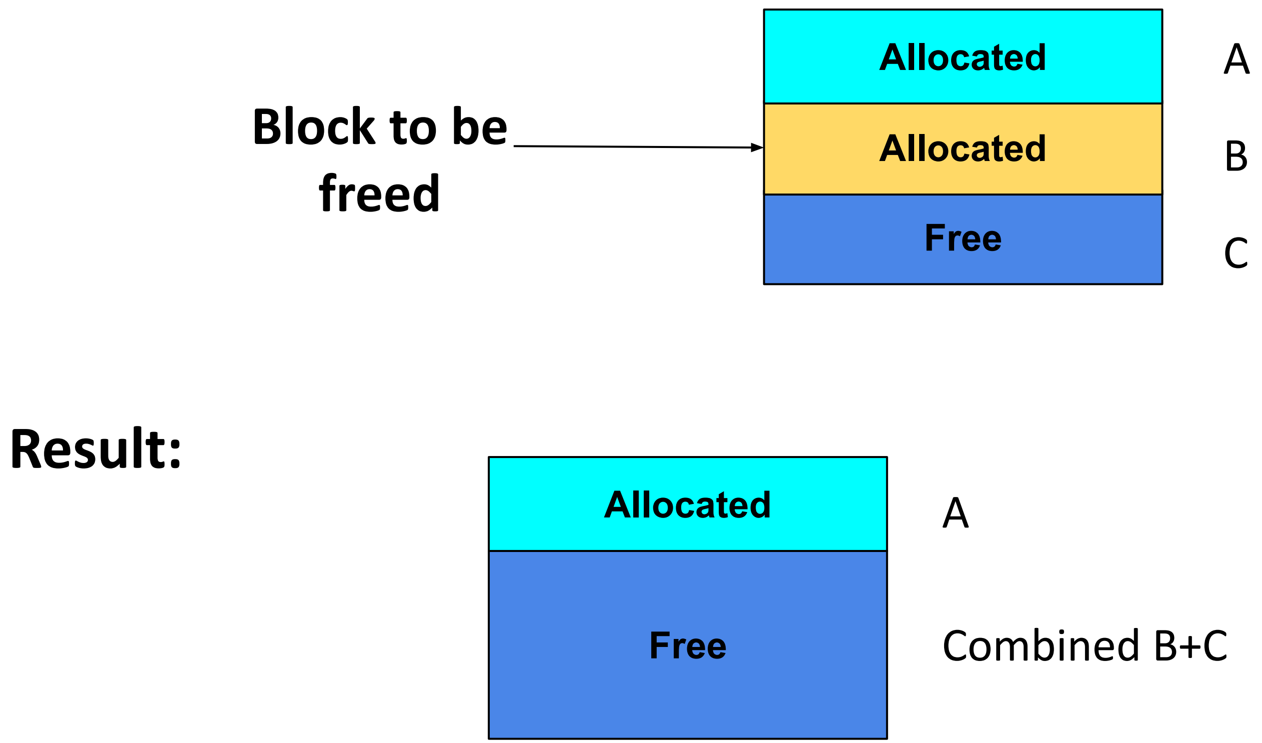
Coalesce: Case 2



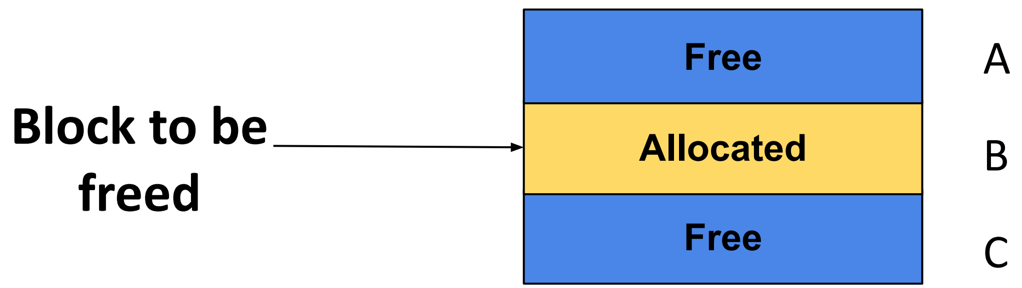
Result:



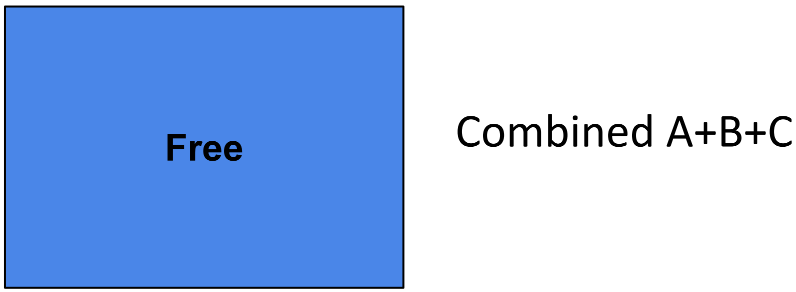
Coalesce: Case 3



Coalesce: Case 4



Result:



Goals

- **Run as fast as possible**
- **Waste as little memory as possible**

- **Seemingly conflicting goals, but with ~~the library malloc call~~ cleverness you can do very well in both areas!**

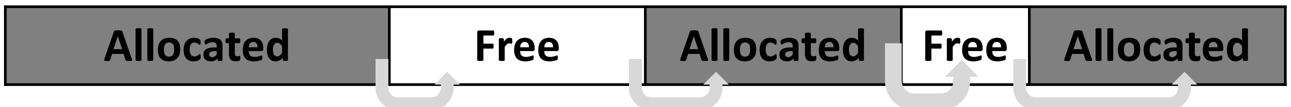
- **The simplest implementation is the implicit list. mm.c uses this method.**
  - Unfortunately…

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver -p
Found benchmark throughput 13090 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark checkpoint

Throughput targets: min=2618, max=11781, benchmark=13090
......................
Results for mm malloc:
  valid    util      ops    msecs    Kops  trace
   yes    78.4%       20    0.002    9632 ./traces/syn-array-short.rep
   yes    13.4%       20    0.001   25777 ./traces/syn-struct-short.rep
   yes    15.2%       20    0.001   24783 ./traces/syn-string-short.rep
   yes    73.1%       20    0.001   19277 ./traces/syn-mix-short.rep
   yes    16.0%       36    0.001   31192 ./traces/ngram-fox1.rep
   yes    73.6%      757    0.145    5237 ./traces/syn-mix-realloc.rep
 * yes    62.0%     5748    3.925    1464 ./traces/bdd-aa4.rep
 * yes    58.3%    87830 1682.766      52 ./traces/bdd-aa32.rep
 * yes    58.0%    41080  410.385     100 ./traces/bdd-ma4.rep
 * yes    58.1%   115380 4636.711      25 ./traces/bdd-nq7.rep
 * yes    56.6%    20547   26.677     770 ./traces/cbit-abs.rep
 * yes    55.8%    95276  675.303     141 ./traces/cbit-parity.rep
 * yes    58.0%    89623  611.511     147 ./traces/cbit-satadd.rep
 * yes    49.6%    50583  185.382     273 ./traces/cbit-xyz.rep
 * yes    40.6%    32540   76.919     423 ./traces/ngram-gulliver1.rep
 * yes    42.4%   127912 1284.959     100 ./traces/ngram-gulliver2.rep
 * yes    39.4%    67012  338.591     198 ./traces/ngram-moby1.rep
 * yes    38.6%    94828  701.305     135 ./traces/ngram-shake1.rep
 * yes    90.9%    80000 1455.891      55 ./traces/syn-array.rep
 * yes    88.0%    80000  915.167      87 ./traces/syn-mix.rep
 * yes    74.3%    80000  914.366      87 ./traces/syn-string.rep
 * yes    75.2%    80000  812.748      98 ./traces/syn-struct.rep
16 16     59.1%  1148359 14732.604     78

Average utilization = 59.1%. Average throughput = 78 Kops/sec
Checkpoint Perf index = 20.0 (util) + 0.0 (thru) = 20.0/100
```

This is pretty slow… most explicit list implementations get above 2000 Kops/sec

Allocation methods in a nutshell

- **Implicit list: a list is implicitly formed by jumping between blocks, using knowledge about their sizes.**



- **Explicit list: Free blocks explicitly point to other blocks, like in a linked list.**
  - Understanding explicit lists requires understanding implicit lists

- **Segregated list: Multiple linked lists, each containing blocks in a certain range of sizes.**
  - Understanding segregated lists requires understanding explicit lists

Choices

- **What kind of implementation to use?**
  - Implicit list, explicit list, segregated lists, binary tree methods, etc.
  - You can use specialized strategies depending on the size of allocations
  - Adaptive algorithms are fine, though not necessary to get 100%.
    - Don’t hard-code for individual trace files - you’ll get no credit/code deductions!
- **What fit algorithm to use?**
  - Best fit: choose the smallest block that is big enough to fit the requested allocation size
  - First fit / next fit: search linearly starting from some location, and pick the first block that fits.
  - Which is faster? Which uses less memory?
  - “Good enough” fit: a blend between the two
- **This lab has many more ways to get an A+ than, say, Cache Lab Part 2**

Finding a Best Block

- **Suppose you have implemented the explicit list approach**
  - You were using best fit with explicit lists

- **You experiment with using segregated lists instead. Still using best fit.**
  - Will your memory utilization score improve?

    Note: you don't have to implement seglists and run mdriver to answer this. That's, uh, hard to do within one recitation session.

  - What other advantages does segregated lists provide?
- **Losing memory because of the way you choose your free blocks is called <u>external fragmentation</u>.**

Metadata

- **All blocks need to store some data about themselves in order for `malloc` to keep track of them (e.g. headers)**
  - This takes memory too…
  - Losing memory for this reason is part of what is called **internal fragmentation**.
- **What data might a block need?**
  - Does it depend on the malloc implementation you use?
  - Is it different between free and allocated blocks?
- **Can we use the extra space in free blocks?**
  - Or do we have to leave the space alone?
- **How can we overlap two different types of data at the same location?**

In a perfect world…

Setting up the blocks, metadata, lists… etc (500 LoC)

\+ Finding and allocating the right blocks (500 LoC)

\+ Updating your heap structure when you free (500 LoC) =

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver
Found benchmark throughput 13056 for cpu type Intel(R)Xeon(R)CPUE5520@2.27

Throughput targets: min=6528, max=11750, benchmark=13056
.......................
Results for mm malloc:
  valid    util      ops    msecs    Kops  trace
   yes    78.1%       20    0.004    5595 ./traces/syn-array-short.rep
   yes     3.2%       20    0.004    5273 ./traces/syn-struct-short.rep
 * yes    96.0%    80000   17.176    4658 ./traces/syn-array.rep
 * yes    93.2%    80000    6.154   12999 ./traces/syn-mix.rep
 * yes    86.4%    80000    3.717   21521 ./traces/syn-string.rep
 * yes    85.6%    80000    3.649   21924 ./traces/syn-struct.rep
16 16     74.2% 1148359   55.949   20525

Average utilization = 74.2%. Average throughput = 20525 Kops/sec
Perf index = 60.0 (util) + 40.0 (thru) = 100.0/100
```

In reality…

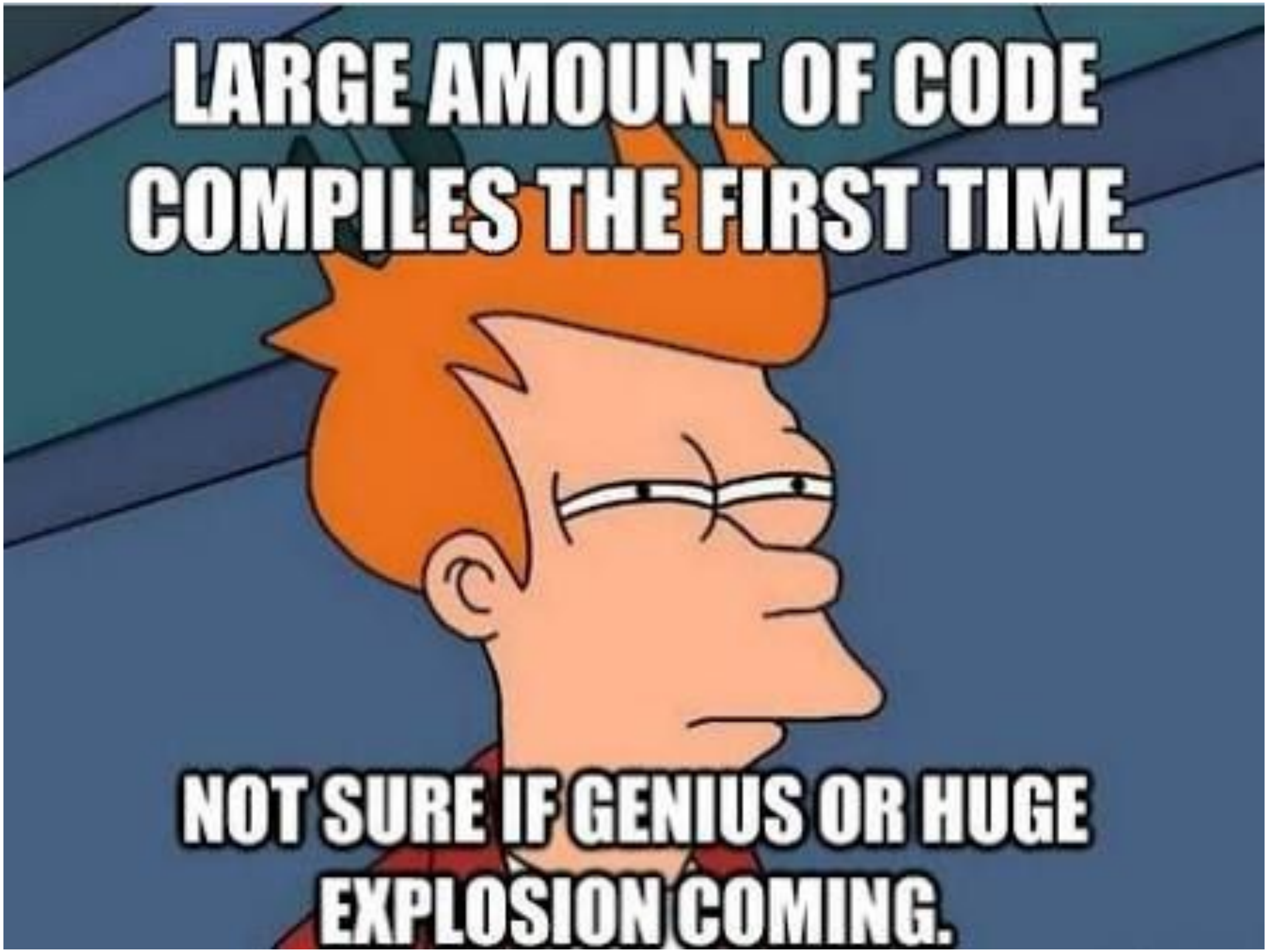
Setting up the blocks, metadata, lists… etc (500 LoC)

\+ Finding and allocating the right blocks (500 LoC)

\+ Updating your heap structure when you free (500 LoC)

+ One bug, somewhere lost in those 1500 LoC =

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver
Found benchmark throughput 13056 for cpu type Intel(R)Xeon(R)CPUE5520@2.27

Throughput targets: min=6528, max=11750, benchmark=13056
.....Segmentation fault
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $
```


LARGE AMOUNT OF CODE
COMPILES THE FIRST TIME.
NOT SURE IF GENIUS OR HUGE
EXPLOSION COMING.

Common errors you might see

- **Garbled bytes**
  - Problem: overwriting data in an allocated block
  - Solution: ~~remembering data lab and the good ol' days~~ finding where you're overwriting by stepping through with gdb
- **Overlapping payloads**
  - Problem: having unique blocks whose payloads overlap in memory
  - Solution: ~~literally print debugging everywhere~~ finding where you're overlapping by stepping through with gdb
- **Segmentation fault**
  - Problem: accessing invalid memory
  - Solution: ~~crying a little~~ finding where you're accessing invalid memory by stepping through with gdb

- **Try running $ `make`**
  - If you look closely, our code compiles your `malloc` implementation with the `-O3` flag.
  - This is an optimization flag. `-O3` makes your code run as efficiently as the compiler can manage, but also makes it horrible for debugging (almost everything is "optimized out").

```
[dalud@angelshark:~/.../15213/s17/rec11] $ make
gcc -Wall -Wextra -Werror -O3 -g -DDRIVER -Wno-unused-function -Wno-u
./macro-check.pl -f mm.c
clang -Wall -Wextra -Werror -O3 -g -DDRIVER -Wno-unused-function -Wno
gcc -Wall -Wextra -Werror -O3 -g -DDRIVER -Wno-unused-function -Wno-u
```

```
(gdb) print block
$3 = <optimized out>
(gdb) print asize
$4 = <optimized out>
```

  - For malloclab, we've provide you a driver, `mdriver-dbg`, that not only enables debugging macros, but compiles your code with `-O0`. This allows more useful information to be displayed in GDB

Debugging Strategies

- **Write a heap checker!**
  - Checks the invariants of your heap to make sure everything is well-formed
  - If you write detailed error messages, you can see exactly why your heap is incorrectly formed
- **Use assertions in your functions!**
  - 122 style contracts can also help you catch where things go amiss
  - Gives more information than a segfault
  - Import
- **Use a debugger!**

Debugging Guidelines

If you have this problem... → ***You might want to...***

Ran into segfault → Locate a segfault
- `run`
- `<>`
- `backtrace`
- `list`

Trace results don't match yours → Reproduce results of a trace

Don't know what trace output should be → Reproduce results of a trace

Reproduce results of a trace
- Run with gdb
  - `gdb args`

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

Debugging mdriver

- (gdb) **`x /gx block`**
  - Shows the memory contents within the block
  - In particular, look for the header.
- (gdb) **`print *block`**
  - Alternative: (gdb) **`print *(block_t *) <address>`**
  - Shows struct contents

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 consistency checker

- **mm-2.c activates debug mode, and so mm_checkheap runs at the beginning and end of many of its functions.**

```c
/*
 * If DEBUG is defined, enable printing on dbg_printf and contracts.
 * Debugging macros, with names beginning "dbg_" are allowed.
 * You may not define any other macros having arguments.
 */
#define DEBUG // uncomment this line to enable debugging

#ifdef DEBUG
/* When debugging is enabled, these form aliases to useful functions */
#define dbg_printf(...) printf(__VA_ARGS__)
```

***Even though the checker in mm-2.c is short and buggy**

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)

Strategy - Suggested Plan for Completing Malloc

0. *Start writing your checkheap!*

1. Get an explicit list implementation to work with proper coalescing and splitting

2. Get to a segregated list implementation to improve utilization

3. Work on optimizations (each has its own challenges!)
   - Remove footers
   - Decrease minimum block size
   - Reduce header sizes

Strategy - Suggested Plan for Completing Malloc

0. *Start writing your checkheap!* *Keep writing your checkheap!*

1. Get an explicit list implementation to work with proper coalescing and splitting *Keep writing your checkheap!*

2. Get to a segregated list implementation to improve utilization

Keep writing your checkheap!

3. Work on optimizations (each has its own challenges!)
   - Remove footers
   - Decrease minimum block size
   - Reduce header sizes

Keep writing your checkheap!

MallocLab Checkpoint

- **Checkpoint should take a bit less than half of the time you spend overall on the lab.**

please write checkheap
or we will scream

- **Read the write-up. Slowly. Carefully.**
- **Use GDB - watch, backtrace**
- **Ask us for debugging help**
  - Only after you implement mm_checkheap though! You gotta learn how to understand your own code - help us help you!

Appendix: Advanced GDB Usage

- `backtrace`: Shows the call stack
- `up/down`: Lets you go up/down one level in the call stack
- `frame`: Lets you go to one of the levels in the call stack
- `list`: Shows source code
- `print <expression>`:
  - Runs any valid C command, even something with side effects like mm_malloc(10) or mm_checkheap(1337)
- `watch <expression>`:
  - Breaks when the value of the expression changes
- `break <function / line> if <expression>`:
  - Only stops execution when the expression holds true
- Ctrl-X Ctrl-A or cgdb for visualization