Recitation 10: Malloc Lab

Instructors

April 1, 2019

Administrivia

- **Please fill out mid-semester feedback!**
  - Course Feedback (https://bit.ly/2I1WliR)
  - Individual TA Feedback (https://bit.ly/2UiigZN)
- **Malloc checkpoint due Tuesday, April 9!** yeeT
- **Malloc final due the week, Thursday, April 18!** yooT
- **Malloc Bootcamp:**

  Sunday, April 7 at Rashid Auditorium, 7-9PM □
  - We will cover ✧fun and flirty✧ ways to succeed post-malloc checkpoint!
  - Tell your friends to come (if they're in 213 (if they want to come (don't force your friends to do things they don't want to do that's not what friends are for)))

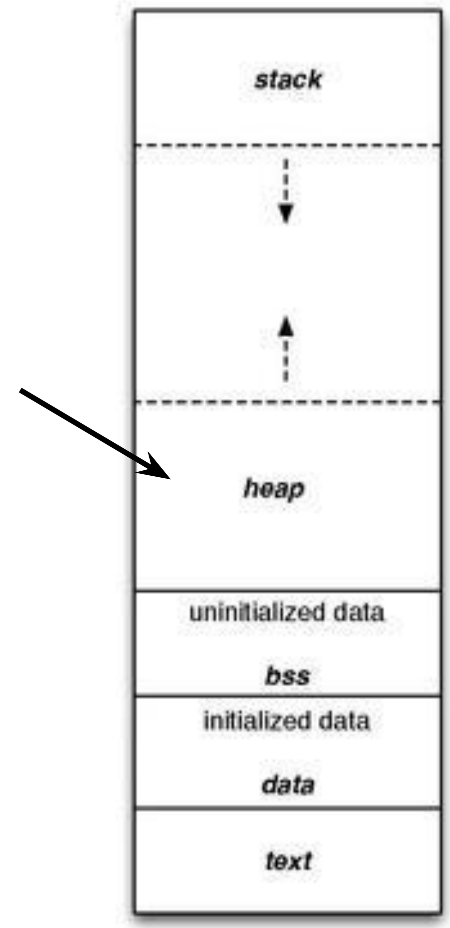
Outline

- 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

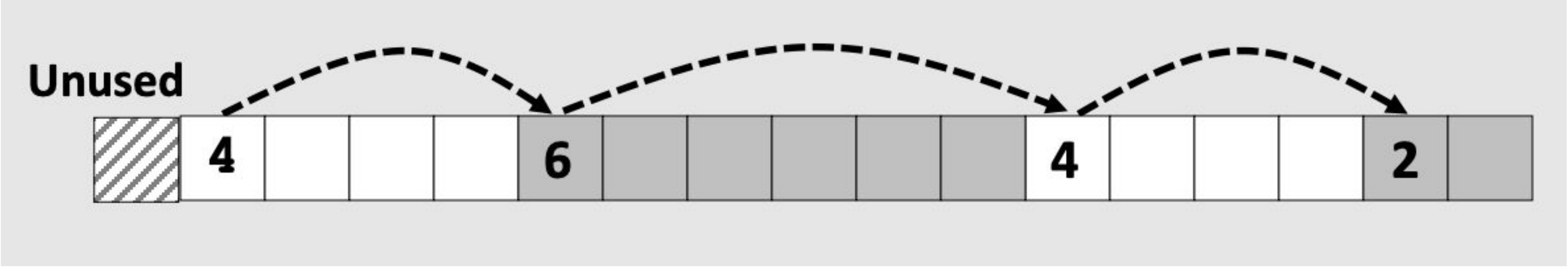
- Overall, malloc does three things:

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.

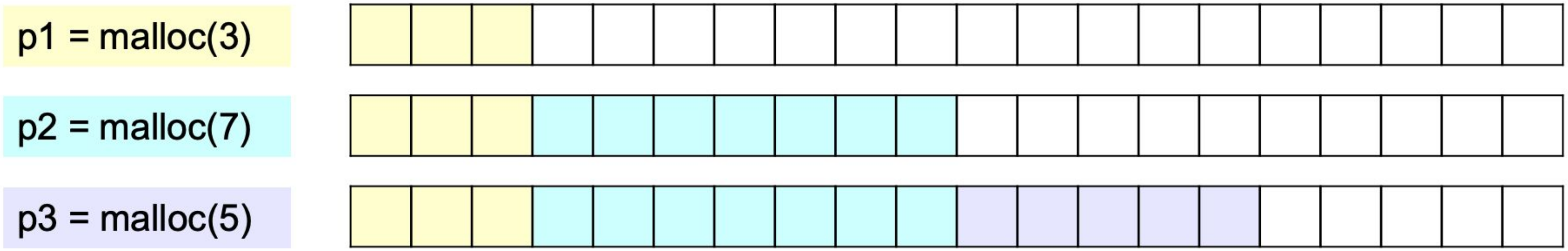
Concept (Implicit list)

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



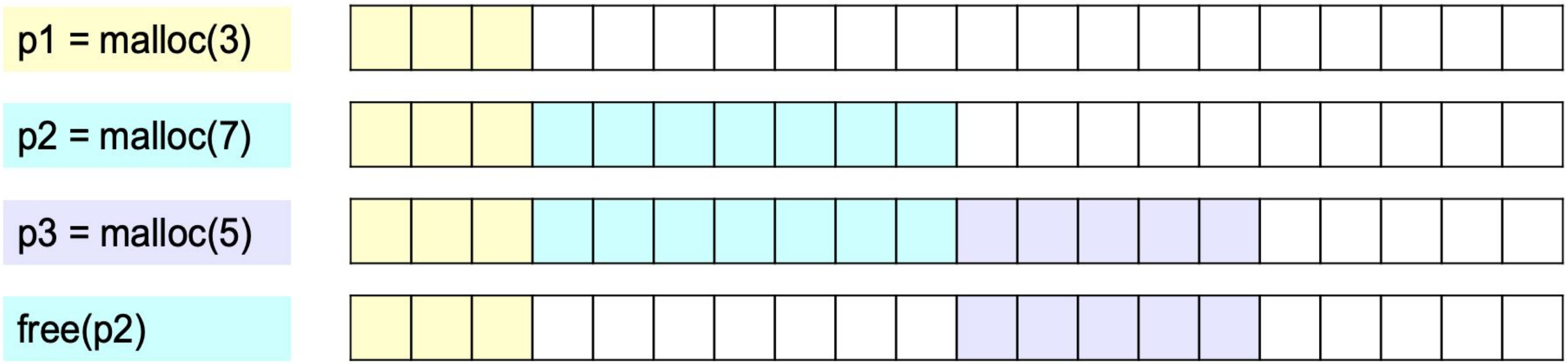
Concept (Implicit list)

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



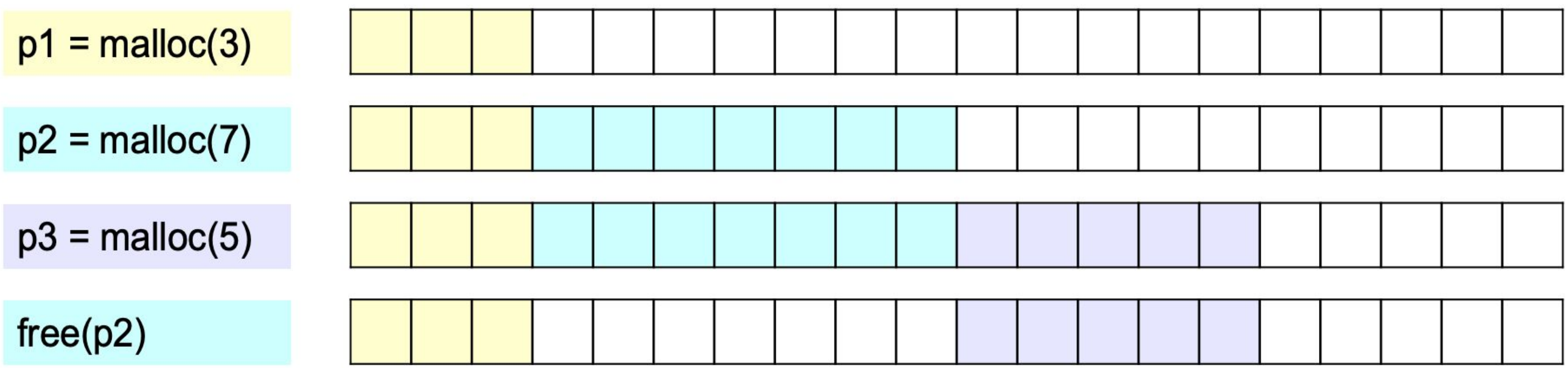
Concept (Implicit list)

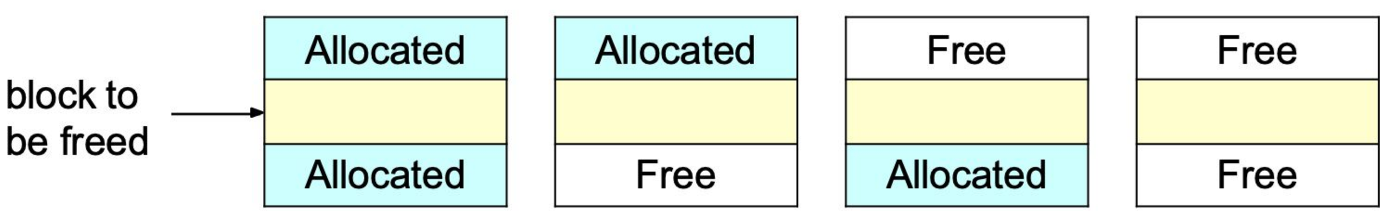
3. **Updates the structure when the user frees a block of memory.**



Concept (Implicit list)

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





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!**

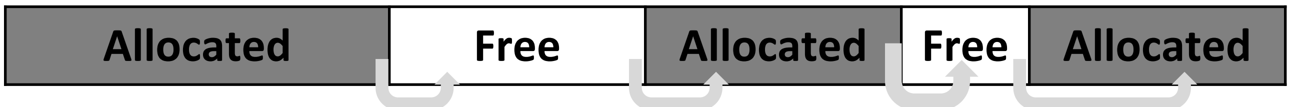
- **The simplest implementation is the implicit list. mm.c uses this method.**
  - Unfortunately…

```
[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 10000 Kops/sec

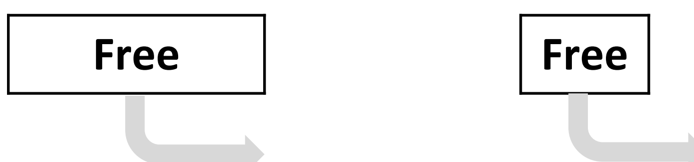
Allocation methods in a nutshell

- **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 fits.**
  - 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 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@2.27G

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…

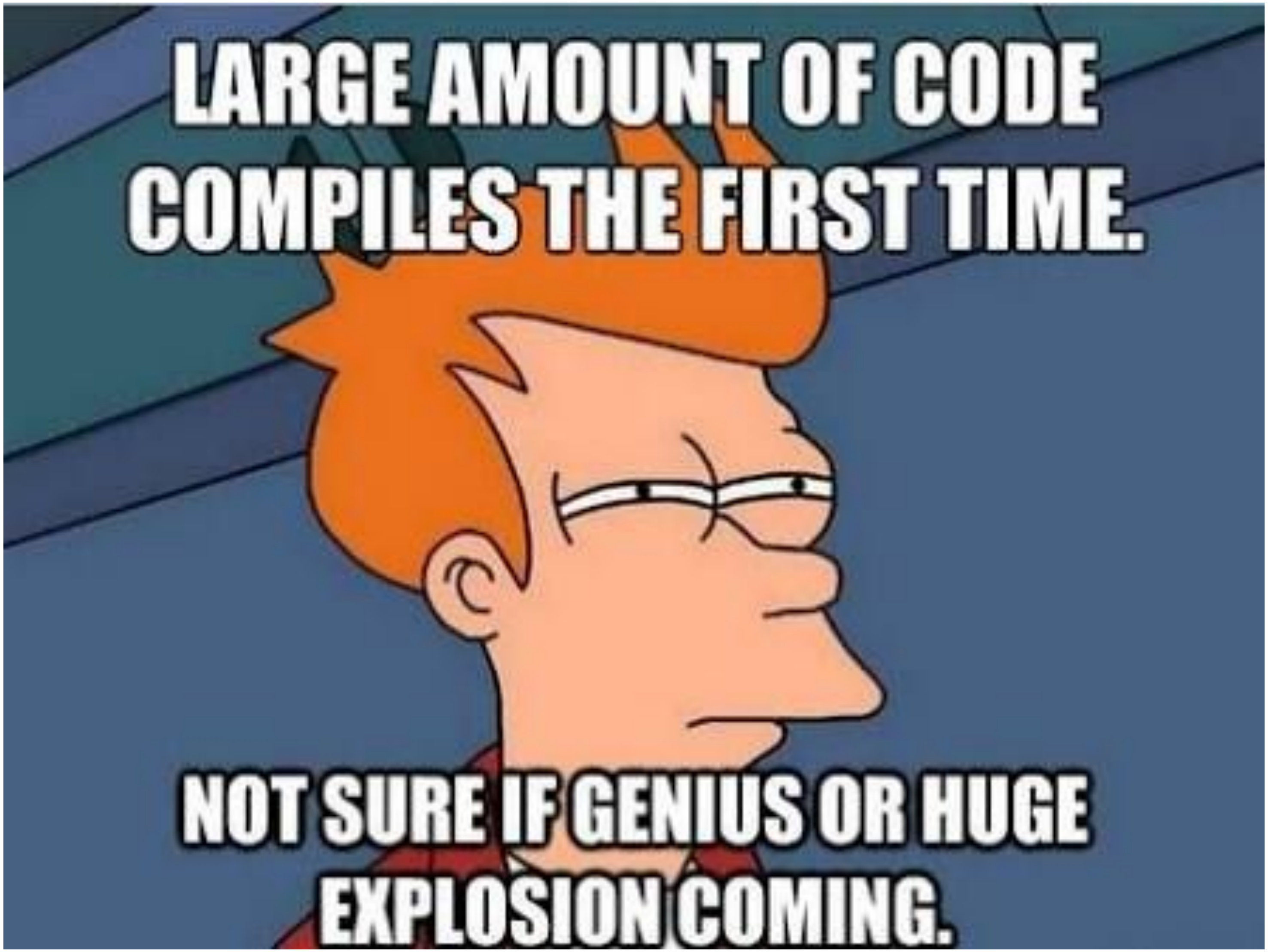
Setting up the blocks, metadata, lists… etc (500 LoC)

\+ 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

Activity: Coalescing

- **What's wrong with this coalesce_block()?**

```
exception AssertFail

fun coalesce_block (block : ref block_t) -> ref block_t =
  if not (get_alloc block) then raise AssertFail else
  let
    val size = get_size block

    val block_prev : ref block_t = find_prev block
    val block_next : ref block_t = find_next block

    val prev_alloc : bool = get_alloc block_next
    val next_alloc : bool = extract_alloc (!(find_prev_footer block))

    val new_size = (size +
      (if not prev_alloc then get_size block_prev else 0) +
      (if not next_alloc then get_size block_next else 0))

    val new_block : ref block_t =
      (if not prev_alloc then block_prev else block)
  in
    write_header (new_block, new_size, false);
    write_footer (new_block, new_size, false);

    if not (get_alloc new_block) then raise AssertFail else ();
    new_block
  end
```

Activity: Coalescing

happy april fool’s
C >>> SML
#functionsarepointers
#functionsarewelldocumented

- **What’s wrong with this coalesce_block()?**

```
exception AssertFail

fun coalesce_block (block : ref block_t) -> ref block_t =
  if not (get_alloc block) then raise AssertFail else
  let
    val size = get_size block

    val block_prev : ref block_t = find_prev_block
    val block_next : ref block_t = find_next_block

    val prev_alloc : bool = get_alloc block_next
    val next_alloc : bool = extract_alloc (!(find_prev_footer block))

    val new_size = size +
      (if not prev_alloc then get_size block_prev else 0) +
      (if not next_alloc then get_size block_next else 0))

    val new_block : ref block_t =
      (if not prev_alloc then block_prev else block)
  in
    write_header (new_block, new_size, false);
    write_footer (new_block, new_size, false);

    if not (get_alloc new_block) then raise AssertFail else ();
    new_block
  end
```

PRANKED

The *Real* Activity: GDB Practice

- **Using GDB well in malloclab can save you *HOURS*[1,2] of debugging time**
  - Average 20 hours using GDB for "B" on malloclab
  - Average 23 hours not using GDB for "B" on malloclab

*** Average time is based on Summer 2016 survey results**

- **Form pairs**

```
wget https://www.cs.cmu.edu/~213/activities/s19-rec-malloc.tar
tar xvf s19-rec-malloc.tar
cd s19-rec-malloc
make
```

- **Two buggy mdrivers**

Debugging Guidelines

If you have this problem…	***You might want to…***
Ran into segfault	Locate a segfault – `run` – `<>` – `backtrace` – `disas`
Trace results don't match yours Don't know what trace output should be	Reproduce results of a trace - Run with gdb – `gdb args`

Debugging mdriver

$ **gdb --args ./mdriver -c traces/syn-mix-short.rep**

(gdb) **run**

(gdb) **backtrace**

(gdb) **list**

Optional: Type Ctrl-X Ctrl-A to see the source code. Don’t linger there for long, since this visual mode is buggy. Type that key combination again to go back to console mode.

1) What function is listed on the top of backtrace?

2) What line of code crashed?

3) How did that line cause the crash?

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

Remember the output from `(gdb) bt`?

- `(gdb) frame 1`
  - Jumps to the function one level down the call stack (aka the function that called `write_footer`)
  - Ctrl-X, Ctrl-A again if you want to see visuals
- **What was the caller function? What is its purpose?**
  - Was it writing to `block` or `block_next` when it crashed?

Thought process while debugging

- **`write_footer` crashed because it got the wrong address for the footer…**
- **The address was wrong because the header of the block was some garbage value**
  - Since `write_footer` uses `get_size(block)` after all
- **But why in the world does the header contain garbage??**
  - The crash happened in `place`, which basically splits a free block into two and uses the first one to store things.
  - Hm, `block_next` would be the new block created after the split? The one on the right?
  - The header would be in the middle of the original free block actually. Wait, but I wrote a new header before I wrote the footer!
    - Right? …Oh, I didn't. Darn.

Heap consistency checker

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

```
/*
 * 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__)
```

- **The next bug will be a total nightmare to find without this heap consistency checker*.**

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

Now you try debugging this - second example!

```
$ gdb --args ./mdriver-2 -c
                 traces/syn-array-short.rep
(gdb) run
```

Yikes… what error are we getting?

Now you try debugging this - second example!

$ gdb --args ./mdriver-2 -c traces/syn-array-short.rep
(gdb) run

Yikes… what error are we getting?

~ g a r b l e d b y t e s ~

* an accurate representation of what's actually going on in your blocks

Now you try debugging this - second example!

```
(gdb) watch *0x8000026d0   /* Track from first garbled payload */

(gdb) run
(gdb) continue
(gdb) continue    /* Keep going until coalesce_block */


(gdb) backtrace
(gdb) list
```

Ah, it seems like nothing's amiss…

Running with mdriver-2-dbg...

- **Let's run it with mdriver-2-dbg, which has a lower optimization - will give us more insight, like the stack trace below**

```
(gdb) file mdriver-2-dbg
(gdb) run
(gdb) continue
…
(gdb) list
```

Running with mdriver-2-dbg...

- Now try printing out the values of prev_alloc / next_alloc...

```
(gdb) print prev_alloc
(gdb) $1 = <optimized out>
```

%rip, they're optimized out! We have to change the optimization level to get what we truly want.

Running with mdriver-2-dbg...

- Go into your Makefile (vim Makefile) => change “COPT_DBG = -O0” so that all local variables are preserved

```
$ make clean
$ make
$ gdb --args ./mdriver-2-dbg -d 2 -c
            traces/syn-mix-short.rep
(gdb) b mm-2.c:450     /* Cut to the chase… */
(gdb) run
(gdb) continue
…
(gdb) print next_alloc
(gdb) $1 = true      /* SUCCESS! */
```

Strategy - Suggested Plan for Completing Malloc

0. *Start writing your checkheap!*

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

3. Get to a segregated list implementation to improve utilization

4. 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!*

3. Get to a segregated list implementation to improve utilization

Keep writing your checkheap!

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

Keep writing your checkheap!

MallocLab Checkpoint

- Due ***next Tuesday!***
- **Checkpoint should take a bit less than half of the time you spend overall on the lab.**
- **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**
- `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 for visualization**