

15-213 Recitation

Malloc Lab (Part II)

Your TAs

Friday, March 14th

Agenda

■ Review:

- Heap Layout + Quick Roadmap of Malloc Checkpoint

■ Debugging

- Finding errors with contracts and **`gdb`**
- Instrumentation

■ Malloc Final Overview

■ Style

Reminders

- `malloc` Deadlines:
 - Checkpoint: ***March 18th (Tuesday)***
 - Final: ***March 25th***
 - 7% of final grade (+4% for Checkpoint)
- Written 7 due ***March 19th***
- Watch your email for Checkpoint Code Review sign-ups!

Review: Malloc Checkpoint

Where are we? - Checkpoint

Quick Discussion: What should we implement for checkpoint and how do they aid in performance?

0. Started with a working (but slow) implicit list version
1. Implement **coalesce_block()** *first*.
2. Implement an *explicit free list*.
3. Implement *segregated lists*



This is a good
place to currently
be!

Debugging

What does “Garbled Bytes” mean?

1. Your `malloc` returns a block pointer to satisfy a request.
2. `mdriver` writes bytes into payload
3. Later, `mdriver` checks that those bytes are intact:
 - If bytes have been overwritten, your `malloc` is overwriting data in an allocated block!

Now what?

- Double check your heap invariants. Are they exhaustive?
- If that doesn't help, use `gdb` to watch writes to the address getting garbled.

Debugging: Overview

- Refer to last week's recitation for common errors, and what they mean.
- *Use tools: **gdb** breakpoints and watchpoints.*
 - Note: Valgrind will not work!
- *Write a heap checker! We'll be grading it in the next code review!*
 - Add new heap invariants as you add new features.
- ***Today: debugging walkthrough!***
 - Garbled bytes + Using contracts

Debugging Activity

- What happens if we run the program normally?

```
$ ./mdriver -c ./traces/syn-struct-short.rep
```

```
ERROR [trace ./traces/syn-struct-short.rep, line 16]: block 1 (at  
0x8000000a0) has 8 garbled bytes, starting at byte 16
```

```
ERROR [trace ./traces/syn-struct-short.rep, line 21]: block 4 (at  
0x800000180) has 8 garbled bytes, starting at byte 16
```

```
correctness check finished, by running tracefile  
"traces/syn-struct-short.rep".  
=> incorrect.
```

```
Terminated with 2 errors
```

Not very helpful...

Debugging: Using Watchpoints

- Now let's try again with watchpoints!

```
$ gdb --args ./mdriver-dbg1 -c ./traces/syn-struct-short.rep

(gdb) watch *0x8000000a0
(gdb) run

// Keep continuing through the breaks:
// write_block()
// 4 x memcpy
Hardware watchpoint 1: *0x8000000a0

Old value = 129
New value = 32
write_block() at mm.c:333
```

- Now we know to take a closer look at **write_block()**!

Debugging: Using Contracts

- Now let's run a version of the file that uses *contracts*:

```
$ ./mdriver-dbg2 -c ./traces/syn-struct-short.rep  
  
mdriver-dbg: mm.c:331: void write_block(block_t *, size_t, _Bool):  
Assertion `(unsigned long)footerp < ((long)block + size)' failed.  
Aborted (core dumped)
```

- This version had a contract in place to check that the footer is where we expect it to be.
- Writing effective contracts can save a lot of debugging time!

Debugging: Miscellaneous Tips

■ `mdriver`

- Use `-D` option to detect garbled bytes as soon as possible
- Use `-V` for verbose mode to find out which trace caused the error

- If the error happens in the first few allocations, can set breakpoints on `mm_malloc` and `mm_free` and step through line by line.

Instrumentation

Common Problems

- *Throughput is very low*
 - Which operation is likely the most costly? Where is the program likely to spend most of its time?
- *Utilization is very low / Out of Memory*
 - Which operation can cause you to allocate more memory than you may need?
- We can use *instrumentation* to investigate both problems!

Adding Instrumentation

- Instrumentation: add *temporary* code that collects measurements for metrics you're interested in.
 - **eg.** how often are certain functions called?
 - You can always remove the code afterwards.
 - Can temporarily go over 128 byte writable global limit!
- These measurements can guide your development process:
 - Develop insights into performance before you spend time on implementation.

Instrumentation Example: Low Throughput

- Program is likely to spend most of its time in `find_fit()`'s loops.
- How efficient is your fit algorithm? How might you find out?

```
static block_t *find_fit(size_t asize)
{
    block_t *block; call_count++
    for (block = heap_listp; get_size(block) > 0;
         block = find_next(block))
    { block_count++
        if (!(get_alloc(block)) && (asize <= get_size(block)))
        {
            return block;
        }
    }
    return NULL; // no fit found
}
```

Instrumentation: Other Metrics

- What are the most common request sizes?
 - How many are 8 bytes or less?
 - How many are 16 bytes or less?
 - How might this inform your design?
- What other things might we want to measure?

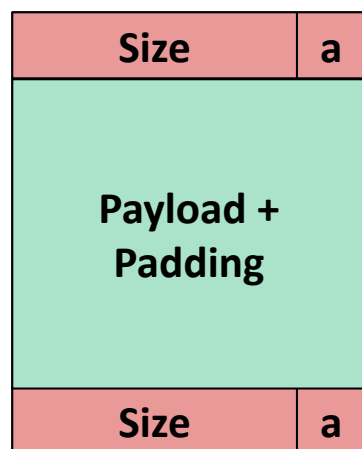
Malloc Final

What are we trying to do?

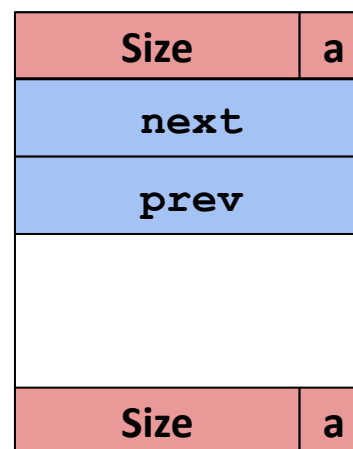
- In Checkpoint, you dramatically improved the *throughput* of your allocator.
- For Final, you will need to greatly improve *utilization* while maintaining a high throughput.
- We will cover:
 1. Footer Removal in Allocated Blocks
 2. Decreasing Minimum Block Size

Current Block Structure

- Here is the current structure of our block (post-checkpoint)
- When is each component utilized in our implementation?
- Do we **need** each component at all time / in all cases?



Allocated (as before)



Free

Zooming In: Footer

- When is the footer used?
 - To access the size/allocation status of previous block during coalescing
- When do we care about the size?
 - When the block is free! If the previous block is allocated, we no longer need to know the size.

Footer Removal: Implementation

- What do we need footers for?
 - Coalescing
 - **Key observation:** do we need to know the size or position of the previous block if we're not going to coalesce with it?
- We just need some way to determine whether the block before us is allocated...

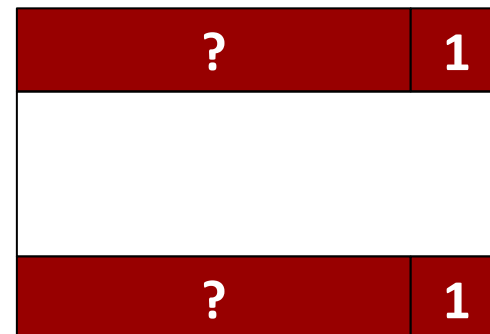
0	1
32	1
64	0
64	0
0	1

*No footers in
allocated blocks!*

*Free blocks still
need footers.*

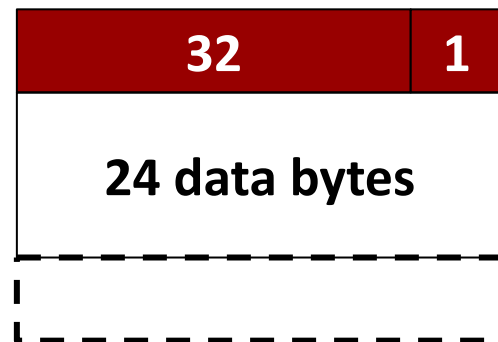
Footer Removal: Example 1

- Let's say we call `malloc(24)`. Can our block of size 32 satisfy the request?
- Add on overhead:
 - Header: +8 bytes = 32 bytes
 - Footer: +8 bytes = 40 bytes
- Round to multiple of 16 => 48 bytes
- Doesn't fit!



Footer Removal: Example 1

- What if we had no footer?
- Add on overhead:
 - Header: +8 bytes = 32 bytes
 - ~~○ Footer: +8 bytes = 40 bytes~~
- Round to multiple of 16 => 32 bytes
- Now it fits!
 - We have reduced *internal fragmentation*.

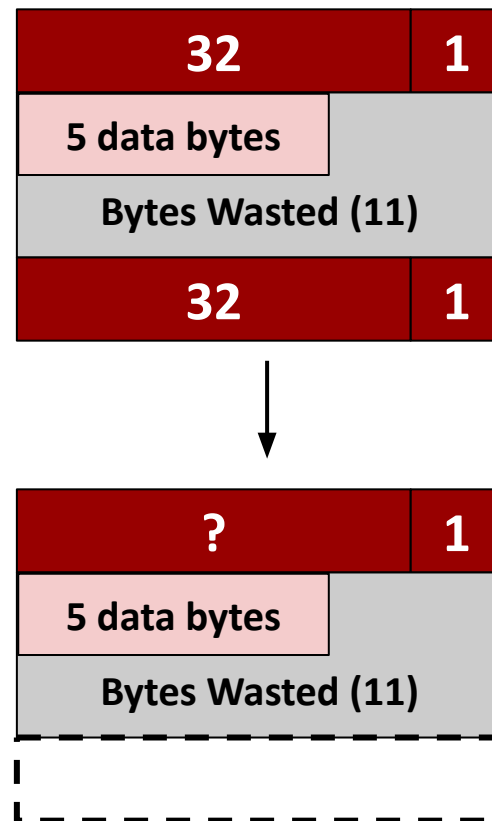


Footer Removal: Example 2

- Now suppose we call `malloc(5)`.

Does removing footers help?

- What is our minimum block size?
 - Still 32 bytes! (header, next, prev, footer for free blocks)
- Header + 16 byte minimum payload uses 24 bytes => round up to minimum block size
- **No benefits in this case!**



Recap: Removing Footers

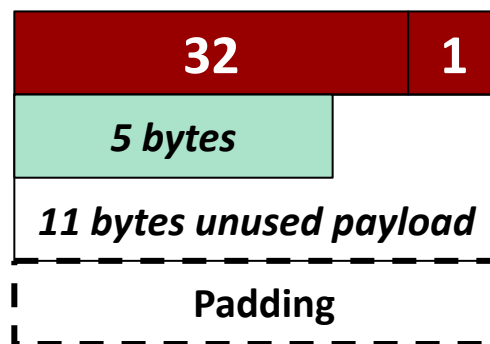
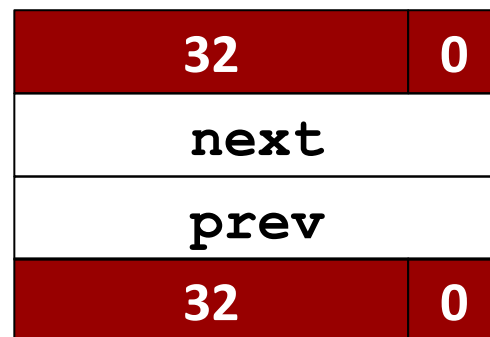
- For a large enough allocation request we can:
 - Include all the information we need for free blocks
 - Also reduce the total block size by cutting the overhead from footers!
- Remember, this does **not** reduce the minimum block size!
 - Though it can help us build towards it...

Why Mini Blocks?

- Let's go back to the example where we called `malloc(5)`
- What was the reason that removing footers yielded no benefits?
- How can we circumvent this barrier to reduce internal fragmentation?

Decreasing Minimum Block Size

- Currently, minimum block size is 32:
 - 8 byte header
 - 16 byte payload (min.)
 - 8 byte footer (for free blocks)
- If we do `malloc(5)`, there's a lot of wasted space due to the min size constraint
- Can we create a design with a smaller minimum block size?



Final Tips

- The shift from checkpoint to final requires us to think more about utilization rather than throughput!
- We talked about several features we can add to improve utilization in certain cases
- What are other features of malloc we can modify to further improve utilization? How do they help?

Style

Style

- Checkpoint Code Review: Heap Checker Quality
- Final Code Review: Code Style
- Remember the style guidelines!
 - Modularity: use helper functions (e.g., for linked lists)!
 - Documentation
 - *File header*: have you described all your design decisions (block structure, fit algorithm, etc.)?

Warning!

- Note that there are implementations that may achieve better performance vs be less complex to design!
- Compressed headers is a technique that reduces the size of the header, reducing internal fragmentation
- Another possible design is to represent your explicit list as a tree!
- Proceed with caution in implementing these two features as they have a higher complexity!

Wrapping Up

- `malloc` Deadlines:

- *Checkpoint: **March 18th (Tuesday)***
- *Final: **March 25th***

- Written 7 due ***March 19th***

The End