

15-213 Recitation

Malloc Part II

Your TAs

Monday, October 25th, 2021

Mid-semester Feedback

- Take 5 minutes to fill out the anonymous feedback form on Piazza
- Be as honest as you can since we will try and make any reasonable adjustments for the next half of the semester

Logistics

- Malloc Lab Checkpoint is due on **TOMORROW** at 11:59 pm
- Malloc Bootcamp on **Friday, October 29th**, 7-9 pm (see piazza)
- Malloc Lab Final is due **Tue, November 2nd** at 11:59 pm
- 7% of final grade (+4% for checkpoint)
- Style matters! Don't let all of your hard work get wasted.
 - There are many different implementations and TAs will need to know the details behind your implementation.

Agenda

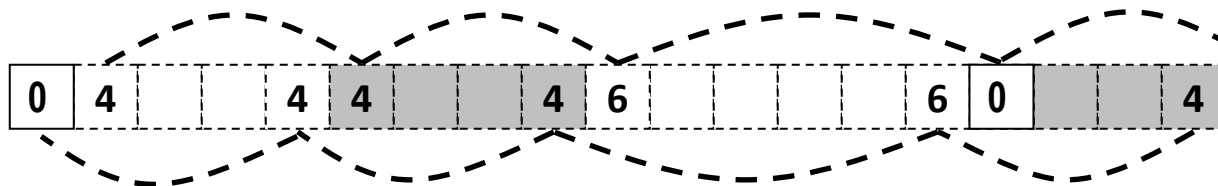
- Logistics
- Malloc Lab
- Checkpoint review
- Activity 1
- Appendix

Understanding Your Code

- Sketch out the heap
- Add Instrumentation
- Use tools

Sketch out the Heap

- Start with a heap, in this case implicit list

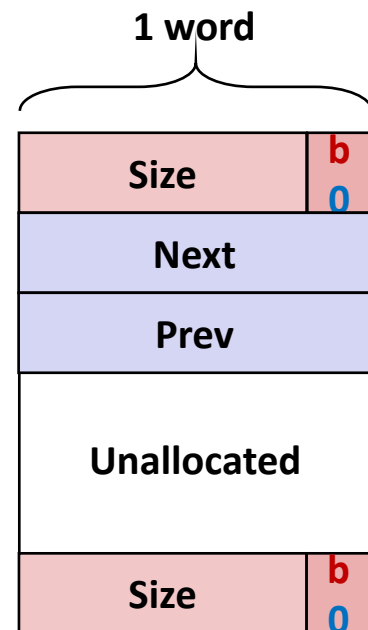


- Now try something, in this case, `extend_heap`

```
block_t *block = payload_to_header(bp);
write_block(block, size, false);
// Create new epilogue header
block_t *block_next = find_next(block);
write_epilogue(block_next);
```

Sketch out the Heap

- **Here is a free block based on lectures 19 and 20**
 - Explicit pointers (will be well-defined see writeup and Piazza)
 - **This applies to ALL new fields you want inside your struct**
 - Optional boundary tags
- **If you make changes to your design beyond this**
 - Draw it out.
 - If you have bugs, pictures can help the staff help you
 - Put a picture of your data structure into your file header
(optional, but we will be impressed)



**Free
Block**

Common Problems

■ **Throughput is very low**

- Which operation is likely the most throughput intensive?
- Hint: It uses loops!
- Solution: ??

Common Problems

■ **Throughput is very low**

- Which operation is likely the most throughput intensive?
- Hint: It uses loops!
- Solution: Instrument your code!

■ **Utilization is very low / Out of Memory**

- Which operation can cause you to allocate more memory than you may need?
- Hint: It extends the amount of memory that you have!
- Solution: ??

Common Problems

■ Throughput is very low

- Which operation is likely the most throughput intensive?
- Hint: It uses loops!
- Solution: Instrument your code!

■ Utilization is very low / Out of Memory

- Which operation can cause you to allocate more memory than you may need?
- Hint: It extends the amount of memory that you have!
- Solution: Instrument your code!

Add Instrumentation

- **Remember that measurements inform insights.**
 - Add temporary code to understand aspects of malloc
 - Code can violate style rules or 128 byte limits, because it is temporary

- **Particularly important to develop insights into performance before making changes**
 - What is expensive throughput-wise?
 - How much might a change benefit utilization?

Add Instrumentation example

- Searching in `find_fit` is often the slowest step
- How efficient is your code? How might you know?
 - Compute the ratio of blocks viewed to calls

```
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
}
```

Add Instrumentation cont.

- **What size of requests?**
 - How many 8 bytes or less?
 - How many 16 bytes or less?
 - What other sizes?

- **What else could you measure? Why?**

- **Remember that although the system's performance varies**
 - The mdriver's traces are deterministic
 - Measured results should not change between runs

Use tools

■ Use `mm_checkheap()`

- Write it if you haven't done so already
- Add new invariants when you add new features
- Know how to use the heap checker.
 - Why do you need a heap checker? 2 reasons.

■ Use `gdb`

- You can call `print` or `mm_checkheap` whenever you want in `gdb`. No need to add a whole lot of `printf`'s.
- Offers useful information whenever you crash, like backtrace.
- Write helper functions to print out free lists that are **ONLY** called from GDB

Write your own traces!

- **Write short traces that test simple sequences of malloc and free**
- **Read the README file in the traces directory and the writeup from the traces assignment to see how trace files need to be written**

mdriver-emulate

- Testing for 64-bit address space
- Use correctly sized masks, constants, and other variables
- Be careful about subtraction between size types (may result in underflow/overflow)
- Reinitialize your pointers in mm_init

Garbled Bytes

- **Malloc library returns a block**
 - mdriver writes bytes into payload (using memcpy)
 - mdriver will check that those bytes are still present
 - If malloc library has overwritten any bytes, then report garbled bytes
 - Also checks for other kinds of bugs

- **Now what?**

- **The mm_checkheap call is catching it right?**
- **If not, we want to find the garbled address and watch it**

Garbled Bytes GDB and Contracts

- Get out a laptop
- Login to shark machine
- `wget http://www.cs.cmu.edu/~213/activities/rec9.tar`
- `tar -xf rec9.tar`
- `cd rec9`
- **mm.c is a fake implicit list implementation.**
 - Source code is based on mm.c starter code

GDB and Contracts Exercise

- First, let us run without contracts and gdb
- `./mdriver -c ./traces/syn-array-short.rep`

(example output)

```
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

Using watchpoints in GDB

- `gdb --args ./mdriver-dbg1 -c ./traces/syn-struct-short.rep`
- What is the first address that was garbled?
 - Use `gdb watch` to find out when / what garbled it.

```
(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
```

We just broke in
after overwriting



- Tells us to take a closer look at `write_block()`

Contracts Exercise cont.

- Now let us see what happens, when we use the file with 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)
```

- Contract failed on line 331, which gives us a better idea of the source of the issue
 - Open mm.c and try to find what is causing the contract to fail
 - Writing effective contracts can save a lot of debugging time!

Tips for using our tools

- Run `mdriver` with the `-D` option to detect garbled bytes as early as possible. Run it with `-V` to find out which trace caused the error.
- Note that sometimes, you get the error within the first few allocations. If so, you could set a breakpoint for `mm_malloc` / `mm_free` and step through every line.
- Print out local variables and convince yourself that they have the right values.
- For `mdriver-emulate`, you can still read memory from the simulated 64-bit address space using `mem_read(address, 8)` instead of `x / gx`.

Style

- **Well organized code is easier to debug and easier to grade!**
 - Modularity: Helper functions to respect the list interface.
 - Documentation:
 - File Header: Describes all implementation details, including block structures.
 - Code Structure:
 - Minimal-to-no pointer arithmetic.
 - Loops instead of conditionals, where appropriate.
 - Use git!
 - Make sure you ***commit and push*** often and write descriptive commit messages

MallocLab

- **Due next Tuesday**
- **7% of final grade (+ 4% for checkpoint)**
 - Style matters! Don't let all of your hard work get wasted.
 - There are many different implementations and TAs will need to know the details behind your implementation.
- **Read the writeup. It even has a list of tips on how to improve memory utilization.**
- **Read the malloc roadmap posted on Piazza**
- **Rubber duck method**
 - If you explain to a rubber duck what your function does step-by-step, while occasionally stopping to explain why you need each of those steps, you'd may very well find the bug in the middle of your explanation.
 - Remember the “debug thought process” slide from last recitation?