Debugging Malloc Lab

15-213: Introduction to Computer Systems
Recitation 12: Monday, Nov. 11th, 2013
Marjorie Carlson
Section A
News

- Shell Lab:
  - Grades will be released todayish.
  - If you saw them this weekend, that was a mistake, and the grade you saw was probably not the final grade.
  - We can add, honest.

- Malloc Lab:
  - Due Thursday.

- Proxy Lab:
  - Goes out the same day, due Dec. 3.
  - Last lab of the semester! 😊 / 😞
My Thoughts on Grading Shell Lab

- Y’all are losing a lot of points on things that are really easy to fix.
  - Consistent indentation, ≤ 80 characters per line.
  - Program header comments: easiest 2 points ever?

  tsh is a simplistic shell. It provides a command line and allows the user to input very basic Unix commands, which it runs by forking & execing. Acceptable inputs:
  * basic Unix commands, including path, e.g. /bin/ls -l or /bin/echo "Hello world".
  * typing % at the end of the command runs it as a background job.
  * supports redirection (< or >) but not pipes (|).
  * built-in commands:
    * jobs: lists all currently running or stopped processes.
    * fg x: moves job x to the foreground.
    * bg x: moves job x to the background.
    * quit: exits the shell.
  * Ctrl-C and Ctrl-Z are appropriately passed to the foreground job and its children.

- Error-checking system calls.
  - e.g., what if you try to open(filename) but don’t have read permissions on that file, or the filename is too long, or you’re out of file descriptors, or or or...?
Agenda: Debugging Malloc Lab

1. Errors you might get & what might cause them
2. Your best friend, the heap checker
3. Other useful tools
4. Beyond debugging: error prevention; version control
5. Optimization: good coding; gprof
Errors

- Some errors are identified by the driver

```
yixunx@hammerheadshark:~/private/15213/malloclab-handout$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 2261.0 MHz
ERROR [trace ./traces/alaska.rep, line 44]: block 8 has 1 garbled byte, starting at byte 0
ERROR [trace ./traces/alaska.rep, line 48]: block 38 has 1 garbled byte, starting at byte 0
ERROR [trace ./traces/alaska.rep, line 6]: Payload address (0x80000005b) not aligned to 8 bytes
ERROR [trace ./traces/amptjp.rep, line 5]: Payload address (0x800000043) not aligned to 8 bytes
ERROR [trace ./traces/bash.rep, line 9]: Payload address (0x8000000d3) not aligned to 8 bytes
ERROR [trace ./traces/alaska.rep, line 7]: Payload (0x800000718:0x80000be9) lies outside heap (0x80000000:0x800000717)
ERROR [trace ./traces/amptjp.rep, line 6]: Payload (0x800000a40:0x80001237) lies outside heap (0x80000000:0x800000a3f)

.................ERROR: mem_sbrk failed. Ran out of memory...
ERROR [trace ./traces/needle.rep, line 95411]: mm_malloc failed.
```

- The error message is straightforward in most cases
  - “garbled byte” means part of the payload returned to the user has been overwritten by your allocator. (Check your pointer arithmetic!)
  - “out of memory” occurs when the memory is used very inefficiently. (Check whether you’re losing track of blocks.)
Errors

- But most of the time...

```
yixunx@hammerheadshark:~/private/15213/malloclab-handout$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~ 2261.0 MHz
Segmentation fault
yixunx@hammerheadshark:~/private/15213/malloclab-handout$
```

- Why did you segfault? Probably either:
  - Pointer arithmetic error.
  - Violating an invariant.
Fixing a Segfault

- As always, you can use printf and gdb to find out which line segfaulted.
  - BUUUUUUT the line that segfaults is likely not where the error is.
  - What you need to know is the moment the heap went wrong, not the moment that it became obvious that the heap had gone wrong.

- You could print the whole heap before/after every function that modifies it.
  - Scroll up from the point of segfault and find the earliest operation that makes the heap look wrong.
  - This will require you to manually comb through a tremendous amount of information.

- Easiest solution: USE YOUR HEAP CHECKER.
Agenda: Debugging Malloc Lab

1. Errors you might get & what might cause them
2. Your best friend, the heap checker
3. Other useful tools
4. Beyond debugging: error prevention; version control
5. Optimization: good coding; gprof
Heap Checker

- Once you’ve settled on a design, write the heap checker that checks all the invariants of the particular design.
- The checking should be detailed enough that the heap check passes if and only if the heap is truly well-formed.
- Call the heap checker before and after the major operations — whenever the heap should be well-formed.
- Define macros to enable/disable it conveniently.
  - e.g.

```c
#define CHECKHEAP(verb) printf("%s\n", __func__); mm_checkheap(verb);
```
Invariants (Non-Exhaustive)

■ Block level:
  ▪ Header and footer match.
  ▪ Payload area is aligned.

■ List level:
  ▪ Next/prev pointers in consecutive free blocks are consistent.
  ▪ Free list contains no allocated blocks.
  ▪ All free blocks are in the free list.
  ▪ No contiguous free blocks in memory (unless you defer coalescing).
  ▪ There are no cycles in the list (unless you use circular lists).
  ▪ Segregated list contains only blocks that belong to the size class.

■ Heap level:
  ▪ Prologue/Epilogue blocks are at the boundaries and have special size/alloc fields.
  ▪ All blocks stay in between the heap boundaries.

■ And your own invariants (e.g. address order)
Hare and Tortoise Algorithm

- Detects cycles in linked lists.
- Set two pointers, “hare” and “tortoise,” to the beginning of the list.
- During each iteration, move the tortoise forward one node, the hare two.
- If they ever point at the same node, the list has a cycle.
- If the tortoise reaches the end, there are no cycles.

Pictures based on those at http://blog.kyletraff.com/infinite-loops-finding-cycles-in-a-linked-list/
Other Things to Watch For

- Uninitialized pointers and/or memory.
- Make sure `mm_init()` initializes everything.
  - It is called by the driver between every trace.
  - If something is overlooked, you might be able to pass every single trace file, but the complete driver test will fail.
Agenda: Debugging Malloc Lab

1. Errors you might get & what might cause them
2. Your best friend, the heap checker
3. Other useful tools
4. Beyond debugging: error prevention; version control
5. Optimization: good coding; gprof
Useful Tools: Valgrind and GDB

- **Valgrind**
  - The default check (memcheck) will let you know if there are any illegal memory accesses or uninitialized values.
  - A little less useful than in other labs, since you’re managing your own memory.

- **GDB**
  - You know how to stop at a line of code using *breakpoints*.
  - You can also stop when a particular piece of memory is accessed, using *watchpoints*.
    - `watch expr` breaks when that expression is *modified*.
    - `rwatch expr` breaks when `expr` is *read*.
    - `awatch expr` breaks when it’s *read or modified*.
    - To break when the `int` at `0x12345678` is modified: `watch *((int *) 0x12345678)`
Useful Tools: Your Friendly Neighborhood TA

- It can be hard for the TAs to debug your allocator, because this is a more open-ended lab.
- Before asking for help, ask yourself some questions:
  - What part of which trace file triggers the error?
  - Around the point of the error, what sequence of events do you expect?
  - What part of the sequence already happened?
- If you can’t answer them, gather more information.
  - How can you figure out which step(s) worked OK?
  - `printf`, breakpoints, watchpoints...
- Bring us a detailed story, not just a “plot summary.”
  - YES: “Allocations of size blah corrupt my heap after coalescing the previous block at line number blah”
  - NO: “It segfaults.”
Agenda: Debugging Malloc Lab

1. Errors you might get & what might cause them
2. Your best friend, the heap checker
3. Other useful tools
4. **Beyond debugging: error prevention; version control**
5. Optimization: good coding; gprof

Debugging is like being the detective in a crime movie where you are also the murderer.

Filipe Fortes
@fortes

7:57 PM - 9 Nov 13 from Palo Alto, CA
Beyond Debugging: Error Prevention

- It is hard to write code that is completely correct the first time, but certain practices can make your code less error-prone.

- Plan what each function does before writing your code.
  - Draw pictures when a linked list is involved.
  - Consider edge cases (when the block is at start/end of list; when you only have one item in your free list; etc.).

- Write pseudocode first.

- Document your code as (or before!) you write it.
Beyond Debugging: Version Control

- “I had 60 util points just 5 minutes ago!”

- Save mm.c after each major milestone.
  - Most basic: copy files around using the cp command.
  - More efficient: keep different versions in separate c files, and use `ln -sf mm-version-x.c mm.c` to start using a particular version
  - Better: use git/svn/cvs... *Make sure your repository is private.*
Agenda: DebuggingMalloc Lab

1. Errors you might get & what might cause them
2. Your best friend, the heap checker
3. Other useful tools
4. Beyond debugging: error prevention; version control
5. **Optimization: good coding; gprof**
Optimization: Good Coding

- To achieve better performance, sometimes you’ll want to tweak certain parameters.
  - Number of size classes
  - Size parameters of size classes
  - CHUNKSIZE
  - ...

- It’s better to write modular and encapsulated code so that changing the parameters only requires changing a few lines of code.
  - Use macros wisely!
Optimization: gprof

- When you hit a bottleneck, find which part is limiting your performance.
- A profiler is good for this kind of job.
- To use gprof:
  - Change the Makefile to add -pg to the compilation flag.
  - Type make to recompile the driver. (You may need to change something in your file to force it to recompile, since it won’t detect changes.)
  - Run the driver. This will generate a file called gmon.out.
  - Run gprof ./mdriver to see the result.
  - Don’t forget to change the Makefile back afterwards!
## Optimization: gprof flat profile

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>51.81</td>
<td>3.92</td>
<td>3.92</td>
<td>add_range</td>
</tr>
<tr>
<td>15.46</td>
<td>5.09</td>
<td>1.17</td>
<td>run_tests</td>
</tr>
<tr>
<td>8.99</td>
<td>5.77</td>
<td>0.68</td>
<td>randomize_block</td>
</tr>
<tr>
<td>7.93</td>
<td>6.37</td>
<td>0.60</td>
<td>check_index</td>
</tr>
<tr>
<td>7.20</td>
<td>6.92</td>
<td>0.55</td>
<td>get_counter</td>
</tr>
<tr>
<td>2.38</td>
<td>7.10</td>
<td>0.18</td>
<td>access_counter</td>
</tr>
<tr>
<td>2.38</td>
<td>7.28</td>
<td>0.18</td>
<td>callibrate</td>
</tr>
<tr>
<td>1.45</td>
<td>7.39</td>
<td>0.11</td>
<td>2370377</td>
</tr>
<tr>
<td>0.79</td>
<td>7.45</td>
<td>0.06</td>
<td>2169016</td>
</tr>
<tr>
<td>0.66</td>
<td>7.50</td>
<td>0.05</td>
<td>eval_mm_speed</td>
</tr>
<tr>
<td>0.26</td>
<td>7.52</td>
<td>0.02</td>
<td>4261340</td>
</tr>
<tr>
<td>0.20</td>
<td>7.53</td>
<td>0.02</td>
<td>start_counter</td>
</tr>
<tr>
<td>0.13</td>
<td>7.54</td>
<td>0.01</td>
<td>4320821</td>
</tr>
<tr>
<td>0.13</td>
<td>7.55</td>
<td>0.01</td>
<td>clear</td>
</tr>
<tr>
<td>0.13</td>
<td>7.56</td>
<td>0.01</td>
<td>main</td>
</tr>
<tr>
<td>0.13</td>
<td>7.57</td>
<td>0.01</td>
<td>set_fcyc_epsilon</td>
</tr>
<tr>
<td>0.00</td>
<td>7.57</td>
<td>0.00</td>
<td>2118313</td>
</tr>
<tr>
<td>0.00</td>
<td>7.57</td>
<td>0.00</td>
<td>52343</td>
</tr>
<tr>
<td>0.00</td>
<td>7.57</td>
<td>0.00</td>
<td>6761</td>
</tr>
<tr>
<td>0.00</td>
<td>7.57</td>
<td>0.00</td>
<td>185</td>
</tr>
</tbody>
</table>
## Optimization: gprof call graph

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>51.8</td>
<td>3.92</td>
<td>0.00</td>
<td></td>
<td>add_range [1]</td>
</tr>
<tr>
<td>[2]</td>
<td>16.5</td>
<td>1.17</td>
<td>0.08</td>
<td></td>
<td>run_tests [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;spontaneous&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_mallo c [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>coalesce [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_realloc [17]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_init [18]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_free [19]</td>
</tr>
<tr>
<td>[3]</td>
<td>9.0</td>
<td>0.68</td>
<td>0.00</td>
<td></td>
<td>randomize_block [3]</td>
</tr>
<tr>
<td>[4]</td>
<td>7.9</td>
<td>0.60</td>
<td>0.00</td>
<td></td>
<td>check_index [4]</td>
</tr>
<tr>
<td>[5]</td>
<td>7.2</td>
<td>0.55</td>
<td>0.00</td>
<td></td>
<td>get_counter [5]</td>
</tr>
<tr>
<td>[6]</td>
<td>2.4</td>
<td>0.18</td>
<td>0.00</td>
<td></td>
<td>access_counter [6]</td>
</tr>
<tr>
<td>[7]</td>
<td>2.4</td>
<td>0.18</td>
<td>0.00</td>
<td></td>
<td>callibrate [7]</td>
</tr>
<tr>
<td>[8]</td>
<td>2.3</td>
<td>0.05</td>
<td>0.12</td>
<td></td>
<td>eval_mm_speed [8]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;spontaneous&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_mallo c [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>coalesce [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_realloc [17]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mm_init [18]</td>
</tr>
</tbody>
</table>
Final Words

- Start *now* (if not already)!
- Come to office hours *early*.
- Write the heap checker *well*.
- Be prepared to start over several times.

**Before handing in, check:**

- Does the header comment contain a detailed description of your approach?
- Is each function commented?
- Is the indentation correct? (Configure your text editor to use spaces instead of tabs.)
- Are any line over 80 characters? (Go to autolab to verify these.)
Questions?

- Good luck!