18-600 Recitation #12

Malloc Lab - Part 2

November 14th, 2017

REMINDER

- **Malloc Lab checkpoint is due on 11/17**
  - This is Friday (instead of the usual Thursday deadline)
  - No late days available
- **Final submission is due on 11/27**
  - Two late days available
- **Remember:**
  - Revisit any assumptions you make in your code (e.g. initializations)
  - Please follow proper style and **header-comment** guidelines.

AGENDA

- **Recap**
  - Basics
  - Implicit lists, Explicit lists and Segregated lists
- **Design Considerations**
  - Internal and external fragmentation
  - Coalescing
  - Finding a fit
- **Debugging**
  - Heap Checker
  - GDB and HProbe
- **Further Optimization Techniques**

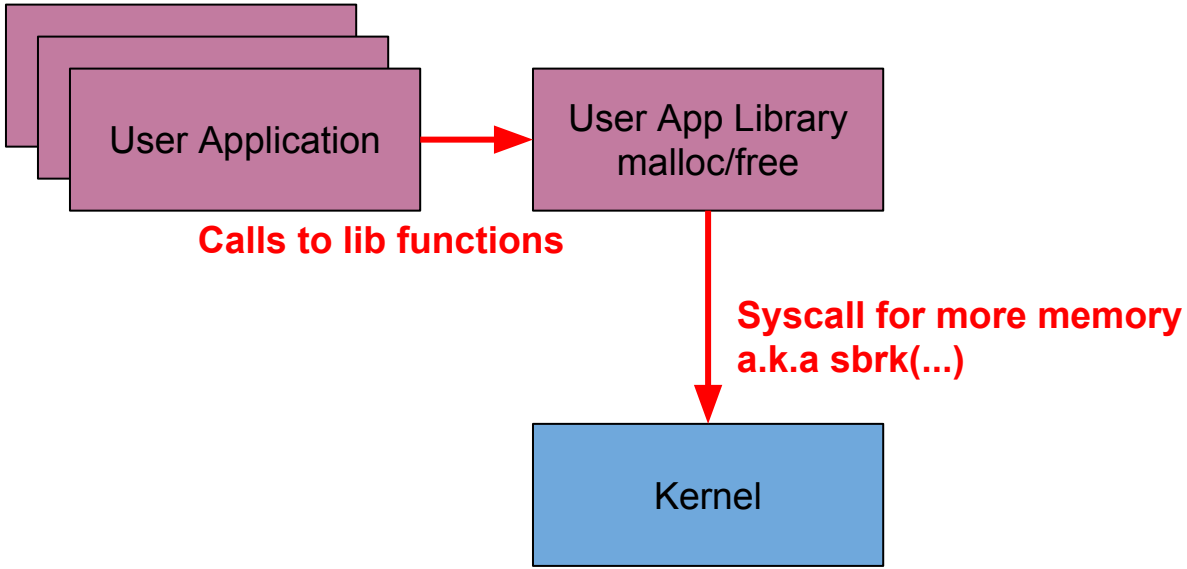
MALLOC: Basics

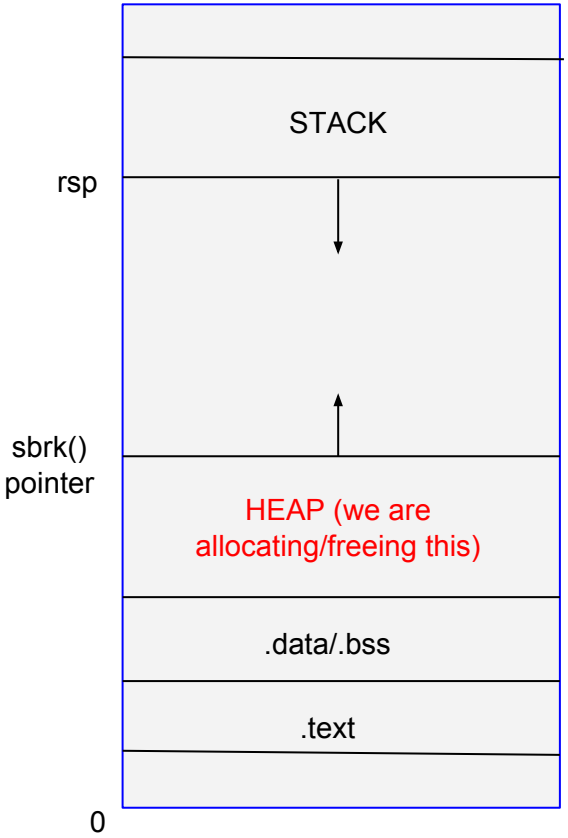
Basics of Memory Allocation

- **When is malloc(), free() used ?**
  - When amount of memory that needs to be used is not known at compile-time
  - When you wish to free up chunks of memory after using them in the program

- **Why do we need a dynamic memory allocator ?**
  - Memory to be allocated is a contiguous chunk from the heap.
  - The goal - fit a chunk of memory that can accommodate the size requested by the user
    - In a short span of time (speed optimization)
    - While wasting minimal heap memory (space optimization)

Malloc - The big picture





MALLOC: Implementation Specifics

The Data Structure

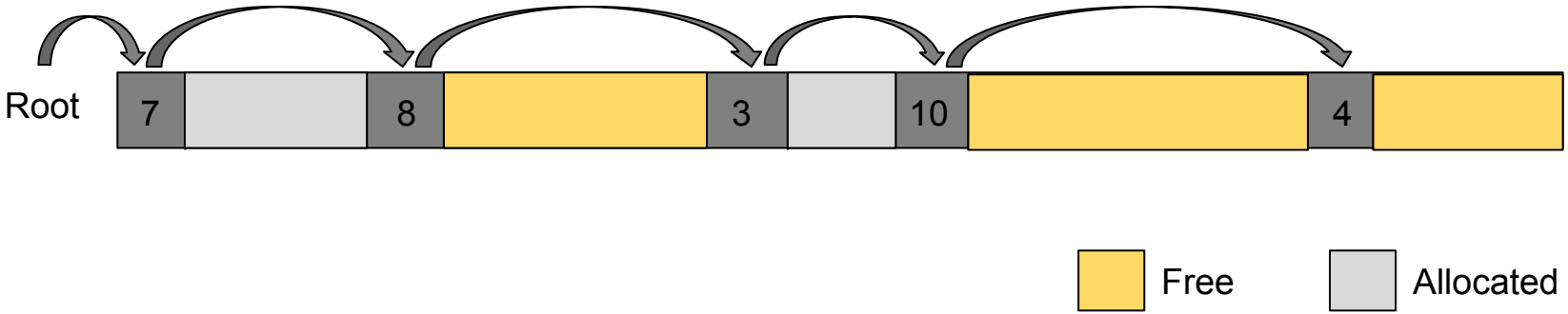
- **Requirements:**
  - The data structure needs to tell us where the blocks are, how big they are, and whether they're free
  - We need to be able to CHANGE the data structure during calls to malloc and free
  - We need to be able to find the **next free block** that is "a good fit for" a given payload
  - We need to be able to quickly mark a block as free/allocated
  - We need to be able to detect when we're out of blocks.
    - What do we do when we're out of blocks?

The data structure

- **The data structure IS your memory!**
- **A start:**
  - <h1> <pl1> <h2> <pl2> <h3> <pl3>
  - What goes in the header?
    - Size ? Allocation status ? Anything else ?
  - Let's say somebody calls free(p2), how can I coalesce?
    - Maybe you need a **footer**? Maybe not?

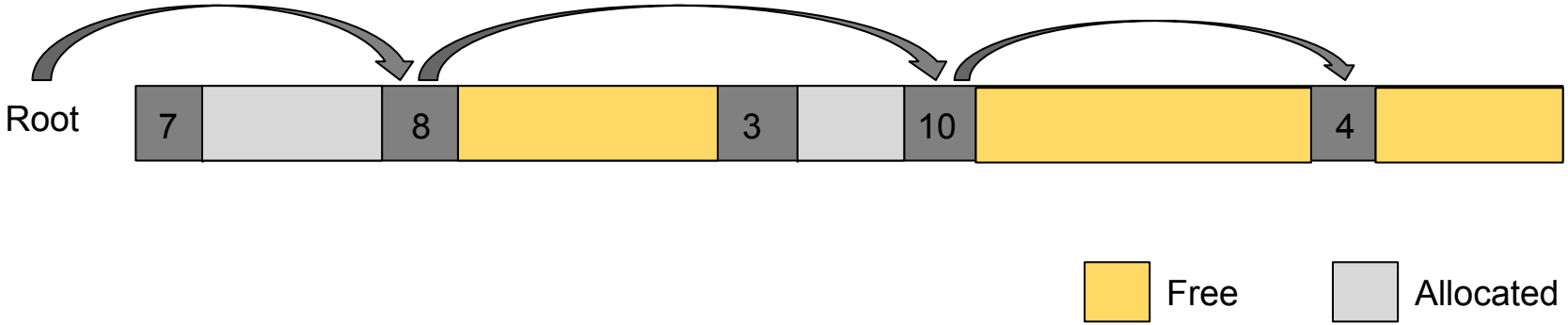
Keeping Track of Blocks

- **Implicit Lists**
  - Implementation - Simple
  - Allocation time - Proportional to total blocks
  - Free time - Constant
  - Memory usage - Depends on implementation



Keeping Track of Blocks

- **Explicit Lists**
  - Implementation - Slightly more complicated
  - Allocation time - Proportional to number of free blocks
  - Free time - Depends upon implementation (constant/linear)
  - Memory usage - Depends on implementation



Explicit Lists

- **Improvement over implicit list implemented by mm-baseline.c**
- **From a root, keep track of all free blocks in a (doubly) linked list**
  - Remember a doubly linked list has pointers to next and previous
  - Do we therefore use more space than in implicit list implementation ?

Explicit Lists

- **Improvement over implicit list implemented by mm-baseline.c**
- **From a root, keep track of all free blocks in a (doubly) linked list**
  - Remember a doubly linked list has pointers to next and previous
  - Do we therefore use more space than in implicit list implementation ?
  - Perhaps not!
    - What data is common between allocated block and free block ?

Explicit Lists

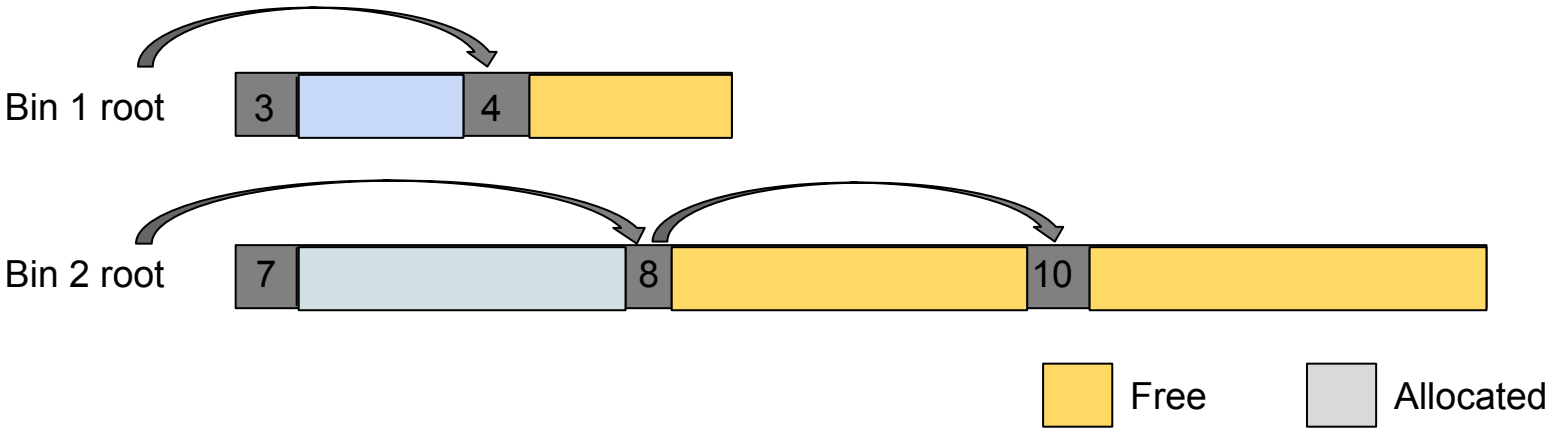
- **Improvement over implicit list implemented by mm-baseline.c**
- **From a root, keep track of all free blocks in a (doubly) linked list**
  - Remember a doubly linked list has pointers to next and previous
  - Do we therefore use more space than in implicit list implementation ?
  - Perhaps not!
    - What data is common between allocated block and free block ?
      - Header, Payload, Footer
    - Does a free block need data to be stored in payload ? Can we reuse this space ?

Explicit Lists

- **Improvement over implicit list implemented by mm-baseline.c**
- **From a root, keep track of all free blocks in a (doubly) linked list**
  - Remember a doubly linked list has pointers to next and previous
  - Do we therefore use more space than in implicit list implementation ?
  - Perhaps not!
    - What data is common between allocated block and free block ?
      - Header, Payload, Footer
    - Does a free block need data to be stored in payload ? Can we reuse this space ?
      - How can we overlap two different types of data at the same location ?
    - Does an allocated block need next and previous pointers to be stored ?
    - Does an allocated block need a footer ?

Keeping Track of Blocks

- **Segregated Lists**
  - Implementation - Extension of explicit lists
  - Allocation time - Proportional to number of free blocks in the bin
  - Free time - Depends upon implementation
  - Memory usage - Better usage with less allocation time

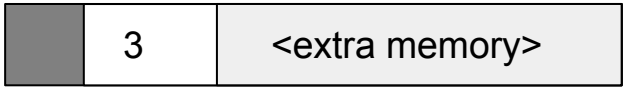


Segregated Lists

- **Can be thought of as multiple explicit lists**
  - What should we group by?
- **Grouped by size – let's quickly find a block of the size we want**
- **What size/number of buckets should we use?**
  - This is up to you to decide

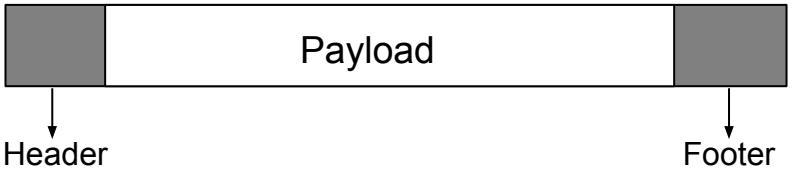
Fragmentation

- **Internal Fragmentation**
  - Occurs due to :
    - Alignment requirement. Payload is not a multiple of block size (not avoidable)



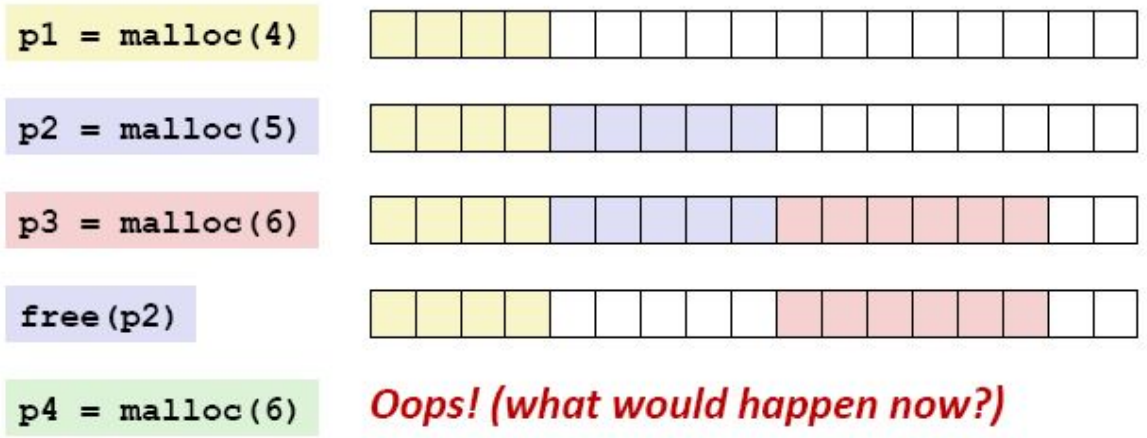
Example: malloc(3) will return a chunk of at least 16 bytes

    - Data structures used for allocation (avoidable)



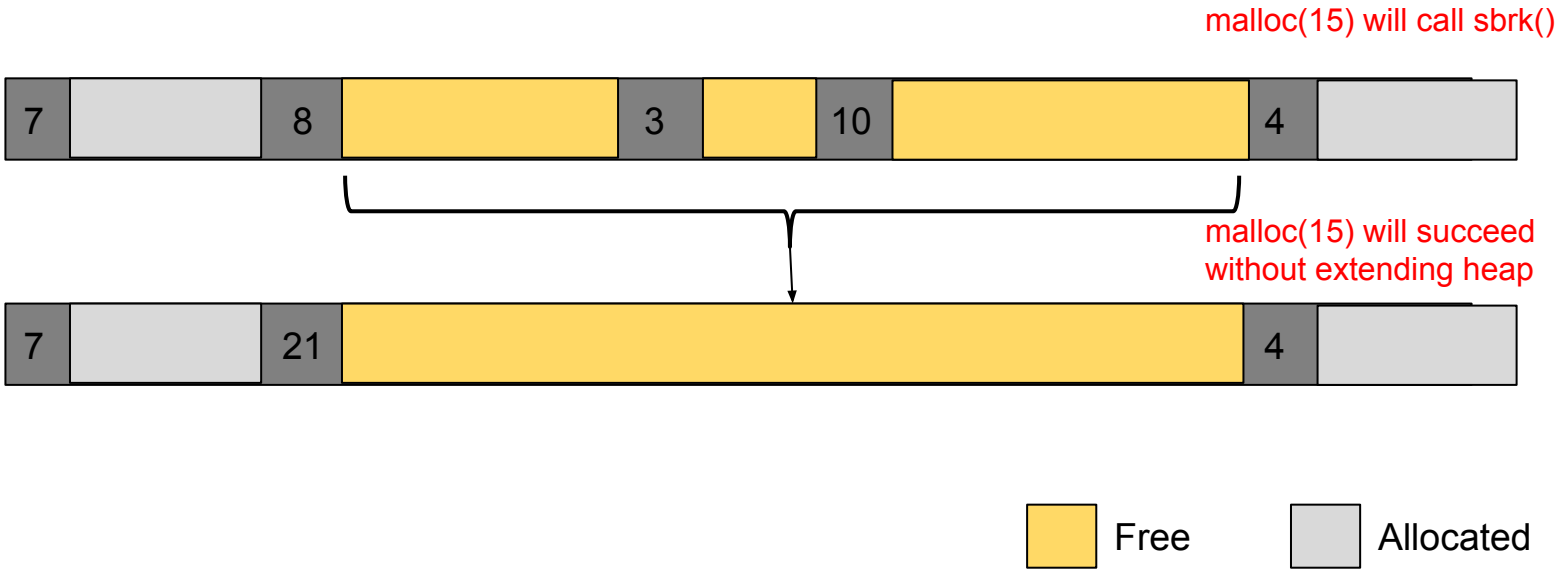
Fragmentation

- **External Fragmentation**
  - Occurs due to total free heap memory being large enough, but no single free block is big enough
  - Depends on patterns of requests



Coalescing

- **How to reduce external fragmentation ? Coalescing !**
  - Group adjacent free blocks together to give larger chunks of free blocks
  - Gets rid of false external fragmentation



Finding a fit

- **First-Fit / Next-Fit / Best-Fit**
  - The policy you choose is up to you ! There is no absolute right/wrong.
  - Has space v/s allocation time tradeoffs
  - Can customize/find a combination of them too

- **Free Block Ordering**
  - FIFO, LIFO or address-ordered ?

- **Memory requested at sbrk() call ?**
  - Smaller requests can result in multiple requests => more time
  - Larger requests => can lead to space wastage

MALLOC: Debugging

Heap Checker

- **Heaper Checker is a graded part of the lab**
  - But write it **first** and use it. Don't write it just before final submission!

- **Heap Checker tips :**
  - Is meant to be correct, not to be efficient.
  - Heap checker should run silently until it finds an error
    - Otherwise you will get more output than is useful
    - You might find it useful to add a "verbose" flag, however
  - Consider using a macro to turn the heap checker on and off
    - This way you don't have to edit all of the places you call it
  - There is a built-in macro called `__LINE__` that gets replaced with the line number it's on
    - You can use this to make the heap checker tell you where it failed
  - Call the heap checker at places that have a logical end. Eg: End of malloc(), free(), coalesce()
  - Call heap checker at the start and end of these functions

Debugging

- **Common Errors :**
  - Dereferencing invalid pointers / reading uninitialized memory
  - Overwriting memory
  - Freeing blocks multiple times (or not at all)
  - Referencing freed blocks
  - Incorrect pointer arithmetic

- **Debugging Tips using mm-baseline.c**
  - We have injected a small bug in mm-baseline.c
  - We attempt to trace it using
    - GDB
    - heapchecker
    - hprobes

Debugging using GDB

- Set the optimization level to 0 before debugging
- Reset the optimization level back after debugging

```
#
# Makefile for the malloc lab driver
#
# Regular compiler
CC = gcc
# Compiler for mm.c
CLANG = clang
# Change this to -O0 (big-Oh, numeral zero) if you need to use a debugger on your code
COPT = -O0
CFLAGS = -Wall -Wextra -Werror $(COPT) -g -DDRIVER -Wno-unused-function -Wno-unused-parameter
LIBS = -lm -lrt
```

Bug Type I: Segmentation Faults

- Recollect the recitation on debugging using GDB
- Very useful to obtain the backtrace
- Examine values of variables

Segmentation Fault

```
bash-4.2$ gdb --args ./mdriver -c traces/syn-array.rep
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-80.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/andrew.cmu.edu/usr5/preetium/private/labs/malloclabcheckpoint-hand
(gdb) run
Starting program: /afs/andrew.cmu.edu/usr5/preetium/private/labs/malloclabcheckpoint-handout
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Found benchmark throughput 19868 for cpu type Intel(R)Xeon(R)CPUE5-2680v2@2.80GHz, benchmark

Throughput targets: min=9934, max=17881, benchmark=19868

Program received signal SIGSEGV, Segmentation fault.
0x000000000040634d in find_prev (block=0x800000000) at mm.c:628
628         size_t size = extract_size(*footerp);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-106.el7_2.8.x86_64
(gdb) bt
#0  0x000000000040634d in find_prev (block=0x800000000) at mm.c:628
#1  0x0000000000405b92 in coalesce (block=0x800000000) at mm.c:417
#2  0x000000000040560f in extend_heap (size=4096) at mm.c:406
#3  0x00000000004054f0 in mm_init () at mm.c:219
#4  0x000000000040322a in eval_mm_valid (trace=0x61d4c0, ranges=0x61d480) at mdriver.c:1032
#5  0x00000000004015ad in run_tests (num_tracefiles=1, tracedir=0x60c1e0 <tracedir> "./", tr
#6  0x0000000000401d64 in main (argc=3, argv=0x7fffffffdfd8) at mdriver.c:506
(gdb) p footerp
$1 = (word_t *) 0x7fffffff8
(gdb) p mem_heap_hi()
$2 = (void *) 0x80000100f
(gdb) p mem_heap_lo()
$3 = (void *) 0x800000000
(gdb)
```

- Notice the footer value
- It is outside the range of the heap

Bug Type 2: Correctness error report by driver

```
-bash-4.2$ ./mdriver -p -V -D -f traces/syn-array.rep
Found benchmark throughput 17422 for cpu type Intel(R)Xeon(R)CPUE5-2680v2@2.80GHz, benchmark checkpoint

Throughput targets: min=3484, max=15680, benchmark=17422

Testing mm malloc
Reading tracefile: traces/syn-array.rep
Checking mm_malloc for correctness, ERROR [trace ./traces/syn-array.rep, line 8]: Payload (0x800000740:0x800001213) overlaps another payload (0x800000740:0x800002127)


Results for mm malloc:
  valid    util     ops   msecs    Kops  trace
 *  no      -         -      -         - ./traces/syn-array.rep
            -         -      -
```

Setting breakpoints

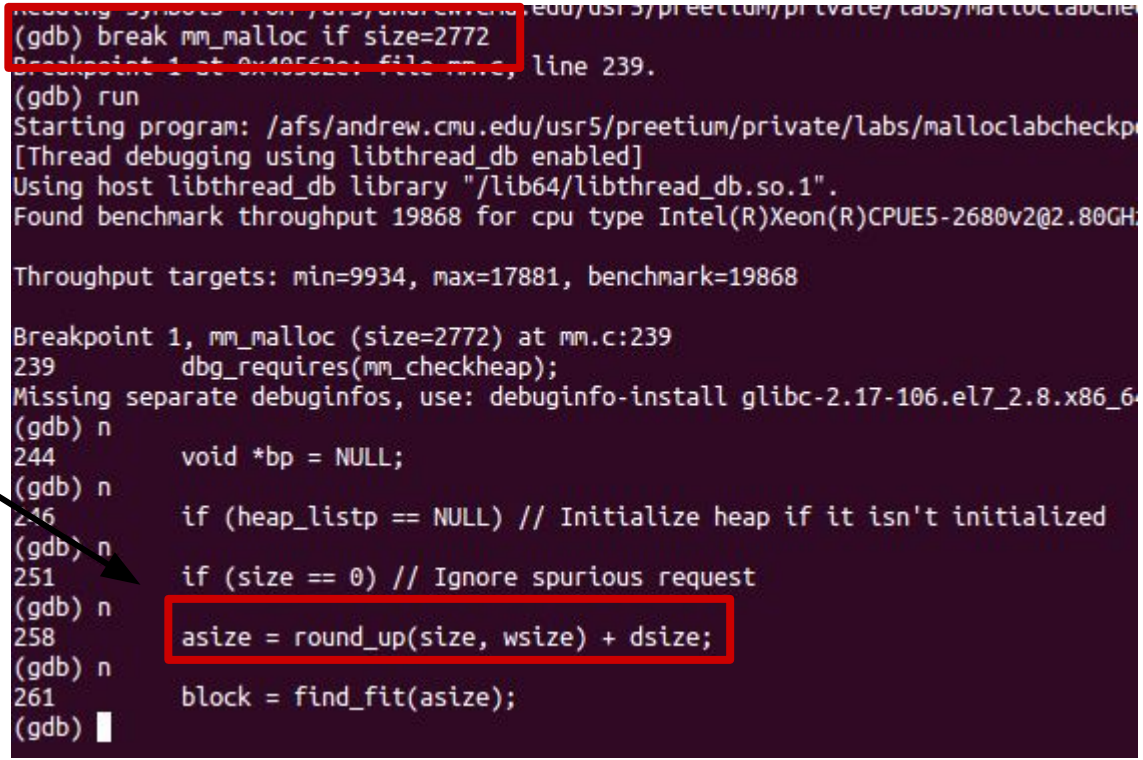
- The tracefile contains a lot of allocations and few frees
- Most likely mm_malloc() has the issue
- Set breakpoint at every call to malloc

Setting breakpoints

```
(gdb) break mm_malloc
Breakpoint 1 at 0x40562c: file mm.c, line 235.
(gdb) run
Starting program: /afs/andrew.cmu.edu/usr5/preetium/private/labs/malloclabcheckpoint-handout/./mdriver -c traces/syn-array.rep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Found benchmark throughput 19868 for cpu type Intel(R)Xeon(R)CPUE5-2680v2@2.80GHz, benchmark regular

Throughput targets: min=9934, max=17881, benchmark=19868

Breakpoint 1, mm_malloc (size=1820) at mm.c:235
235         void *bp = NULL;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-106.el7_2.8.x86_64
(gdb) c
Continuing.

Breakpoint 1, mm_malloc (size=6632) at mm.c:235
235         void *bp = NULL;
(gdb) c
Continuing.

Breakpoint 1, mm_malloc (size=12) at mm.c:235
235         void *bp = NULL;
(gdb) c
Continuing.

Breakpoint 1, mm_malloc (size=2772) at mm.c:235
235         void *bp = NULL;
(gdb) c
Continuing.
ERROR [trace ./traces/syn-array.rep, line 8]: Payload (0x800000740:0x800001213) overlaps another payload (0x800000740:0x80000212


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

Terminated with 1 errors
[Inferior 1 (process 14430) exited normally]
(gdb)
```

Setting conditional breakpoints

Should have been:
`asize = round_up(size, dsize) + dsize;`



Heapchecker

- The above problem is easy to identify using heap checker

```
bash-4.2$ ./mdriver -p -V -D -f traces/syn-array.rep
Found benchmark throughput 17422 for cpu type Intel(R)Xeon(R)CPUE5-2680v2@2.80GHz, benchmark checkpoint

Throughput targets: min=3484, max=15680, benchmark=17422

Testing mm malloc
Reading tracefile: traces/syn-array.rep
Checking mm_malloc for correctness, Line 0, Heap error in block 0x800000738.  Header (0x19f1) != footer (0x19f9)
ERROR [trace ./traces/syn-array.rep, line 7]: mm_checkheap returned false


Results for mm malloc:
  valid    util     ops    msecs     Kops  trace
 *  no      -         -        -         - ./traces/syn-array.rep
             -         -        -

Terminated with 1 errors
```

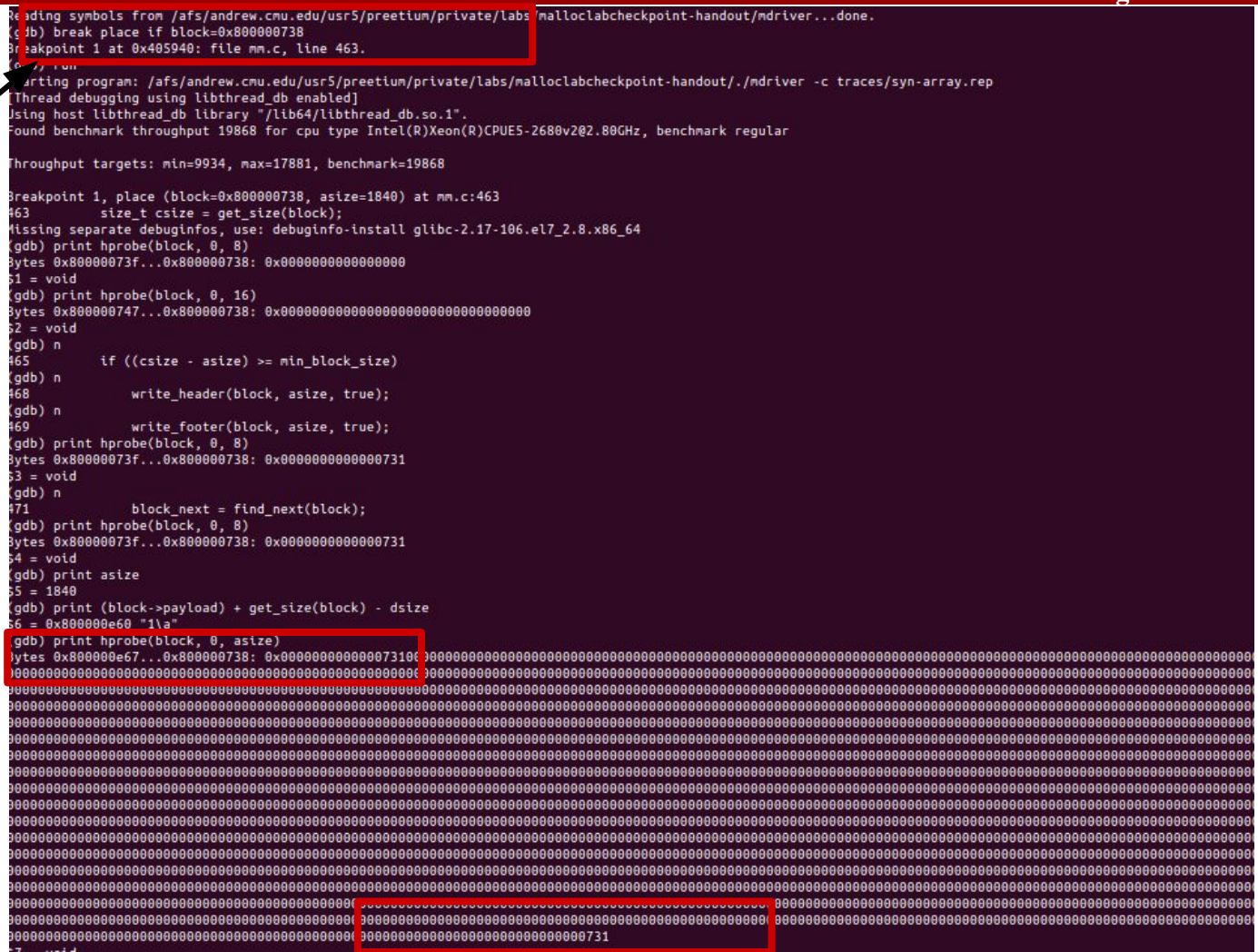
Using Hprobes

- Use hprobes as mentioned in the handout on the defaulting block
- Useful to check the contents of the heap

(gdb) break place if block = 0x800000738

Examine header and footer

(gdb) print hprobes(block, 0, asize)



Using watchpoints

- **Now use watchpoints to observe when the header and footer values change**
  - watch *0x800000e67, where 0x800000e67 is the address of the header as shown by hprobes
  - watch *0x800000738, where 0x800000738 is the address of the footer as shown by hprobes

MALLOC: Optimizations

Basic Optimizations

- **Optimize step-by-step. Don't go all in at once.**

- **Basic optimizations -**
  - Segregated Lists
    - **Note**: A decent implementation of explicit lists is enough to cross the checkpoint.
  - Optimizing the free block finding strategy
  - Basic block splitting (when a larger size is requested than the size of the free block)
  - Coalescing of free blocks

Further Optimizations

- **Eliminate footers in allocated blocks**
  - But, you still need to be able to implement coalescing
- **Decrease the minimum block size**
  - You must then manage free blocks that are too small to hold the pointers for a doubly linked free list
- **Reduce headers below 8 bytes**
  - But, you must support all possible block sizes.
  - Must then be able to handle blocks with sizes that are too large to encode in the header
- **Set up special regions of memory for small blocks**
  - Need to manage these and be able to free a block when given only the starting address of its payload

SUMMARY

- There is no definite optimal solution, everything has trade offs.
  Choose your pick !

- Start early
- Write the heapchecker as you go
- Use gdb and the heapchecker generously
- Modularise your code
- Optimize gradually
- Finish early and enjoy the Thanksgiving break :)