Challenge 2 Hints

The Problem

For Challenge 2 you are:

- Starting with a simple singly-linked free list.
- You’ve added a mutex to that single shared data structure.
- You need it to run the tests significantly faster.

Broad Strategy

- In general, optimizing memory allocators is a hard problem.
- You could spend 5 years and a million dollars paying salaries to a team of 10 on this problem – both Google and Facebook have done that.
- That won’t meet the deadline, and you’re not allowed to pay people to do your homework anyway.
- Instead, try the simplest thing that could work first, benchmark it, and leave time to try two or three other things too.

Strategies

You've got two specific problems:

1) A singly linked list is a slow data structure. It tends to force O(n) operations.
2) The test programs are multi-threaded, so having a single shared data structure with a single mutex is going to be a bottleneck.

Avoid Contention

- Of the two problems, I recommend handling the thread contention problem first.
- The two techniques to consider first are:
  - Multiple arenas
  - Thread-local storage

Multiple Arenas

- When we’re talking about memory allocators, an “arena” is “a full copy of the data structures for your allocator”.
- If you’re using a free list, then an arena is a free list plus the associated mutex.
- If you’re using bins/buckets, then an arena is an array of free lists plus whatever mutexes are needed to go with that structure.
- To avoid fragmentation, you’ll want to mark each allocated block with its source arena so that when you free you can free the memory back to where it came from.

Thread-local Storage

- Thread-local storage is a tool that allows you to have a variable that
  - Acts like a global in your code.
  - Is really a separate variable for each thread
- See “man pthread_key_create”
- See the documentation for the gcc-specific “__thread” keyword.
- Having a thread local cache / arena can avoid locks for some operations entirely, at the cost of more difficult coalescing.

Improving the Data Structure

- A linked list is slow – normally O(N).
- We can get to O(1) / O(log n) with strategies like the buddy system or buckets.
- These strategies tend to work by limiting allocations to specific sizes.
- So first thing to do is to look at the test programs and see what they do – is it a bunch of small allocations? Big allocations? One size? Is every size different?

Optimizing a Bucket Allocator

A bucket allocator, called “bins” in Christo’s slides, works like this:

- An array of free lists.
- Each slot in the array points to free blocks of some fixed size.
- Maybe the sizes are powers of two and midpoints, so like {8, 12, 16, 24, 32, …}
- Malloc and free are normally O(1)
  - Blocks of the same size are interchangeable so order doesn’t matter.
  - We can just treat the free lists as stacks implemented as singly linked lists.
  - Malloc is pop, free is push.

Bucket Coalescing (1/2)

- One design question with buckets is what to do if you don’t have a block of the requested size.
- One option is to split a larger block, but then you need to handle merging later.
- Another option is to have each size be completely independent – if you need a size 128 block and the 128 list is empty, allocate a whole chuck (1 or more pages) of 128’s.
- This is simple, but can lead to fragmentation.

Bucket Coalescing (2/2)

- If buckets are separate, then the only way to effectively coalesce is to unmap whole pages / chunks when they’re freed.
- This requires determining if all the blocks in that chunk are free.
  - We could have a “used” field in the block header and scan all blocks on each free. This flag could even be in the low bits of the size field.
  - We could have a bitmap at the beginning of the block and find it with pointer arithmetic.

Optimizing single-size chunks

- If you allocate whole chunks for each single size, you can put metadata at the beginning of the chunk rather than in each block.
- You need to allocate each chunk such that it’s at an aligned memory address – this happens automatically if a chunk is a page, but larger chunks take extra work.

Large Allocations

- If you store metadata at the beginning of chunks and do pointer arithmetic to find that beginning, then all allocations need to have metadata that can be found that way.
- Large allocations have to start with the same chunk alignment and have the same metadata, although they don’t need to fit entirely in one chunk.