Scribe notes

Goals for today

- **Build our own memory allocator**
- **Building our own garbage collector**

Dynamic memory

1. Fixed- size allocator
2. Conservative garbage collector

Side note → cs61-lectures repository is constantly updated, README.txt has quick quizzes

Memory benchmark (membench). What does it do?

1. allocates an arena (arena is an object that manages a set of allocations)
2. allocates 4096 “memnodes” from the arena
3. For some loop, free one node and then allocate another node
4. Free 4096 nodes

membench should leave the state of memory unchanged

To implement this, we need to implement the functions memnode_arena_new, memnode_arena_free, memnode_alloc, memnode_free.

See code mb-mymalloc.c in cs61-lectures/l06

The type memnode_arena is opaque, meaning that the user of the library doesn’t know what memnode_arena is. It could even be null! e.g. malloc uses a single global arena. Opaque types have no size.

Side note → When you have a warning, make it go away. In C, (void) variable makes the warning about the unused variable memnode_arena go away.

The only thing that we can do with memnode_arena is alloc memnodes.

typedef struct memnode_arena memnode_arena means that we have a struct memnode_arena and we are just going to use it by calling memnode_arena. (meant to be opaque)

Why is malloc slower than memnode_alloc?

1. malloc is more general than we need (malloc executes more complicated code than we need)
2. malloc has high per “call” overhead.

We want to reduce the overhead by calling malloc fewer times. This strategy is called **Batching!** Example of this is eating at meals instead of every second (reduces the constant overhead of cooking, bringing food to your mouth, etc.)

So let's design an arena so that we call malloc once per 4096 memnode allocs. That reduces the per-call overhead of malloc (where 4096 is an arbitrary number). i.e. we malloc a bunch of nodes at once, and then use those for all our memory needs until we run out.

Idea is that for an arena, a group has an array of nodes, say 4096 nodes. The arena is just a linked list of groups. A small structure, the arena, has 2 pointers: a list of groups, and a free list. This free list tells us which memnodes are free. When a memnode is in use, the allocator doesn't know what it is. When a memnode isn't in use, it is a pointer to the next free memnode!

See mb-arena.c for correct implementation of membench arena.

Garbage collection

See testgc.c

Garbage collection is a technique that allows us to never free. At the same time, the program never runs out of memory. Garbage collection is an algorithm that detects objects that are no longer useful and deletes them, thus freeing up memory, and you don't have to deal with it yourself (e.g. no need to watch for double frees!). Languages like Java already have them implemented, so they have fewer nasal demons than C.

In *gc61.c*, there is a malloc that stores metadata in an array. Similar to pset 1.

Since we are storing data in an array, when we free, we should also remove it from the array. To do this, we loop over all of the array, check to see if the pointer in question is between the start of a ptr (in the array) and the ptr (in the array)+sz (associated with ptr and stored alongside it in the metadata array) (*this is done in m61_find_mr)*.

See m61_free in gc61.c which uses vector patterns in C patterns page.

m61_find_allocations gets the address for the allocations of the array. We want it to examine all of the memory in [base, base+sz], and print out every pointer in that memory that was allocated by m61_malloc, or that points inside an allocation made by m61_malloc.

Basically looking for things that look like pointers in that array of memory (trigger-happy garbage collector).

You can see different versions of the code in gc61-01, -02, etc. Sort of like, as it gets done versions.

This function is what we need for a garbage collector.

Since the stack is a chunk of memory, we can find the allocations in the stack with the find allocations function. We start from the stack, look at the pointers in the stack, and then we look at the stuff that is pointed to from the stack (this takes care of heap objects). For this, we must change m61_find_allocations to call itself recursively when it finds a pointer on the region pointed to.

A mark system can be used to make sure that we don't find allocations multiple times.

How can we find the stack? It is a region of memory, and it grows down.

stack_bottom is the address of the local variable in main.
stack_top is the address of m61-gc, so all we need to do is to let it equal a variable (itself in m61-gc).