Gemuele (Gem) Aludino
01:198:214 Systems Programming
Fall 2019

Asst1: ++malloc (plusplusmalloc)

Usage:

Using a `Terminal`, navigate to the directory where the `Asst1.tar.gz` file resides.

`$ mkdir [folder_name]`
ex. `mkdir Asst1_GemAludino`

`$ cd [folder_name]`
ex. `cd Asst1_GemAludino`

Use this command to decompress the tar file:

`$ tar -xf Asst1.tar.gz`

The containing folder `[folder_name]`, will have the following directories/files from the `.tar.gz` file:
(this directory structure is one that I use for all of my C or C++ projects)

- `_debug`
  - different versions of headers/sources go here during development
- `bin`
  - binary executables may go here, sometimes for different platforms
- `build`
  - object files go here
- `client`
  - client source files (with `main()`) go here
- `dat`
  - any input files relevant to the project
- `doc`
  - `README.pdf`
    - PDF documentation
  - `notes.txt`
    - my scratch notes
  - `testplan.txt`
    - describes test E and F, my own workloads
- `include`
  - header files
- `lib`
  - external libraries
- `src`
  - source files
- `temp`
  - holding place for files (may or may not be used)
- `tests`
  - test clients
- `README.md`
- `exclude.txt`
- `Makefile`
- `maketarball`

To build **`memgrind`**,
`$ make`
or
`$ make all`
should be invoked in the Terminal
`$ make clean`
can be used to remove any old builds of `memgrind`, including object files in the `build` folder.

Design of `struct header (header_t) - memory block metadata.`

`struct header` went through a few iterations, the first of which was a singly-linked list node:

```
struct header {
	int size;
	bool free;
	struct header *next;
};
```

It was 13 bytes, field by field, but with padding (the struct will want to align on a byte-boundary that is a multiple of 4) — this version of `header_t` was 16 bytes.

As I was doing tests with malloc/free, I quickly realized that 4096, in the grand scheme of things, really isn't a lot to work with — especially since each block of data needs metadata to go with it. I sought out to make the metadata (`header_t`) as small as possible.

The first thing was removing the `(struct header *)` pointer — since myblock, the buffer storing all allocations, is contiguous…although random access isn't possible, due to each block being different sizes, you can still use pointer arithmetic to traverse from header to header.

```
struct header {
	int size;
	bool free;
	~~struct header *next;~~
};
```

We traverse from one header to the next like this:

```
header_t *position = (header_t *)(myblock);
position = (header_t *)((char *)(position) + sizeof(header_t) + position->size))
```

We treat header as a `(char *)`, so it can be traversed byte-by-byte.
We skip over `(sizeof(header_t) + position->size)` bytes so we can find the next header.

We'll know if we have exceeded the bounds of my block of the next header retrieved is outside the addresses range of `[myblock, myblock + 4096)`

We can get struct header even smaller by reducing int, a 32-bit integer, to a short, a 16-bit integer.
I've typedef'ed short to uint16_t. (sizes should be positive, always…right?)

```
struct header {
	uint16_t size;
	bool free;
};
```

Now we are down to 4 bytes. (1 byte of padding). I worked this way for a while, and it was respectable…but I thought I could get this down even lower.

How?

We can change `uint16_t` to a `int16_t (signed short),`
and remove the free field. But how will we know what is a consumed block, and what is a free block?

Used blocks will be denoted by a negative size, positive blocks will be denoted by a positive size.
But how will traversals work? Won't you end up going backwards unintentionally when you find a used block?
Well, no — just take the absolute value of the size when traversing from header to header.

The final size of header_t is….2 bytes.
`struct header { int16_t size; };`

How does mymalloc work?

If mymalloc has not been called yet,
we initialize myblock by giving it a size of 4094.

If the size request is 0, or greater than 4094, we reject it outright.

We traverse myblock, header by header, until we find a free block.
While we are traversing myblock, if we find two adjacent free blocks, we merge them.
I try to find every opportunity to merge free blocks, and since we are traversing myblock,
we might as well do it — merging can be done in a few instructions anyway.

Once we have left the loop:
    if the block we found is eligible for a split,
    meaning the split will result in the second half having a block size of at least 1
    (there is no point to a split if splitting the block means that the second half will have no space,
     or if the header itself can't even fit — this is why getting the header down to a small size was important)
    …then we split it.
    Otherwise, the block that we hand to the user will have a few extra bytes, unbeknownst to them.

    If we didn't find a free block/big enough block,
    we tell the user and return a NULL pointer.

Note that when we return a pointer to the user (assuming the block is good) -
we must return the base address of the memory they asked for…don't include the header with it!
We return the address

```
(header + 1),
```

or

```
((char *)(header) + (sizeof(struct header))
```

How does myfree work?

First, we run a sanity check:
    is the pointer the client gave us
        a NULL pointer?
        an already freed pointer?
        a pointer unrelated to mymalloc/myfree?
If it passes the sanity check, we continue.

Next,
we decrement size of `struct header` bytes,
so that we can actually work with the size field that we need to review.
(the user gave us their data block.)

If the header is free,
    we notify the user and return. (no double frees allowed)
else
    if the block is in the proper address range (between [myblock + sizeof(header_t), myblock + 4096))
        we toggle it free, and check to see if the adjacent header to the right is also free.
            if they are both free, we merge them.

    Next, we traverse the entirety of myblock to search for adjacent blocks to merge.

Sample times of two separate invocations of memgrind:

```
[ga354@pwd plusplusmalloc]$ ./memgrind

mymalloc allocator stress tests
Each individual test is run 100 times and wall-clock time averaged.

All times are expressed in microseconds (µs)

------------------------------------------------------------
test            mean            slowest         total
------------------------------------------------------------
a               4.73088 µs      10.49600 µs     473.08800 µs
b               49.01120 µs     141.56800 µs    4901.12000 µs
c               15.30368 µs     30.46400 µs     1530.36800 µs
d               15.02208 µs     28.92800 µs     1502.20800 µs
e               27.42528 µs     40.44800 µs     2742.52800 µs
f               194.21696 µs    227.32800 µs    19421.69600 µs

[ga354@pwd plusplusmalloc]$ ./memgrind

mymalloc allocator stress tests
Each individual test is run 100 times and wall-clock time averaged.

All times are expressed in microseconds (µs)

------------------------------------------------------------
test            mean            slowest         total
------------------------------------------------------------
a               2.49088 µs      14.08000 µs     249.08800 µs
b               82.70592 µs     105.47200 µs    8270.59200 µs
c               48.81408 µs     68.86400 µs     4881.40800 µs
d               51.55584 µs     66.81600 µs     5155.58400 µs
e               95.82848 µs     111.10400 µs    9582.84800 µs
f               322.81600 µs    763.39200 µs    32281.60000 µs
```

Test F, the vector test, was especially busy, but is the best representation of a possible real-world use case. (dynamic buffer that expands multiple times, with random deletions/insertions in between)

(I'm hoping my actual vector data structure isn't inefficient, and it's just my allocator policy, but it was still interesting to see it work with my own memory allocator, as opposed to the cstdlib malloc).

realloc was replaced with a malloc of a new buffer (greater size), copying over the contents from the old buffer to the new, and freeing the old buffer, while assigning vector's buffer pointer to the new buffer.

malloc has a memset with 0's added to it, to simulate calloc.

See testplan.txt for more on test F.

```
PROBLEMS   TERMINAL   ...        1: bash
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:427] : will free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:445] : nothing to free
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:488] : malloced
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:399] : --- max allocs met ---
----------------------------------------
Block Address   Status          Block Size
------------    ------          ----------
0x60eaa2        in use          400
0x60ec34        in use          2
0x60ec38        in use          62
0x60ec78        in use          37
0x60ec9f        free            3585
----------------------------------------
Used blocks in list:    4 blocks
Free blocks in list:    1 blocks

Free space:             3585 of 4096 bytes
Available for client:   3585 of 4086 bytes

Total data in use:      511 of 4096 bytes
Client data in use:     501 of 4086 bytes

Largest used block:     400 of 4086 bytes
Largest free block:     3585 of 4086 bytes

Size of metadata:       2 bytes

[client/memgrind.c:402] :
Oct 20 2019 22:21:30
----------------------------------------

Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:415] : freed
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:415] : freed
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:415] : freed
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped
Oct 20 2019 22:21:30 [LOG] [client/memgrind.c:419] : skipped

total iterations: 3364

----------------------------------------
Block Address   Status          Block Size
------------    ------          ----------
0x60eaa2        free            4094
----------------------------------------
Used blocks in list:    0 blocks
Free blocks in list:    1 blocks

Free space:             4094 of 4096 bytes
Available for client:   4094 of 4094 bytes

Total data in use:      2 of 4096 bytes
Client data in use:     0 of 4094 bytes

Largest used block:     0 of 4094 bytes
Largest free block:     4094 of 4094 bytes

Size of metadata:       2 bytes

[client/memgrind.c:503] :
Oct 20 2019 22:21:30
----------------------------------------

[ga354@pwd plusplusmalloc]$
```

Interesting note: why coalescence is important:

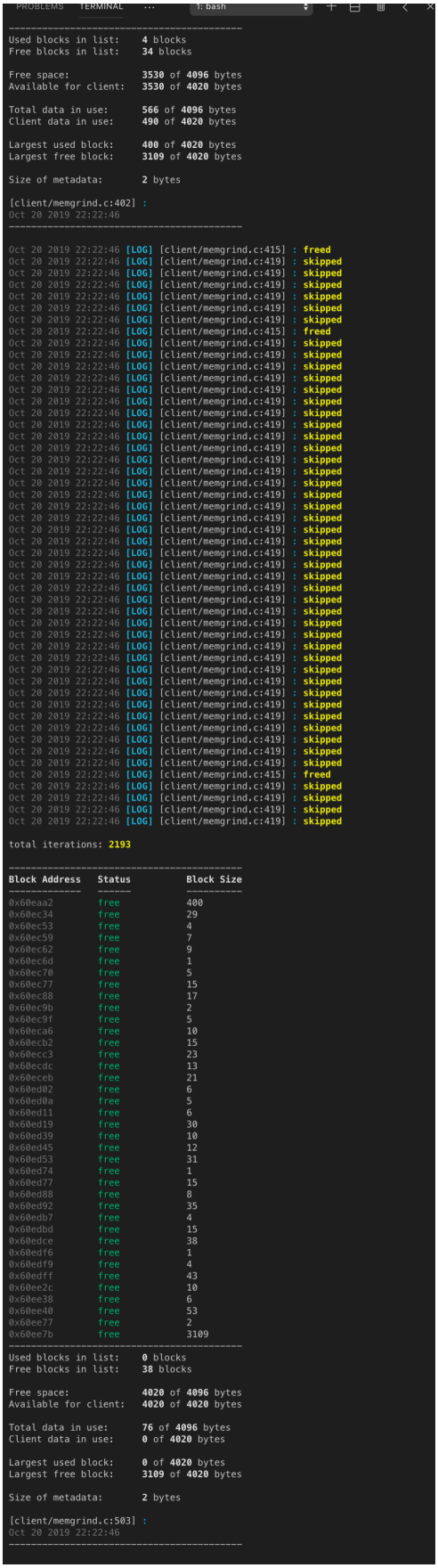
Test D, block merging on:

Here's the snippet:
After 50 allocations have been met,
I print the state of myblock.

Over the course of random allocations and frees,
blocks have been merged along the way, leaving a lot of free space.

You'll see that pointers that can be skipped (due to them not being occupied) are skipped, and the ones that can be freed are freed.

At the end of test D (one instance),
all blocks are freed and merged into one, as it should be.



Without merging, the largest block available at the end of one instance of test D is 3109, which is fine, but it could be much worse (this is just one example).