Oliver Gilan & Rishab Ravikumar
Asst 01: ++malloc
Francisco
15 October 2019

README:

.metadata

size: 1 byte

allocated space terminator: '\x1F'

freed space terminator: '\x1E'

.malloc

Pass in requested allocation size. Pass it first available block of space as pointer and add allocated space terminator to (pointer+size).

To find first available block of space:

1. Start at beginning of block and iterate one byte at a time. tsize = size
2. If index does not contain allocated space terminator decrement tsize by 1. Set pointer to index address.
3. If index contains allocated space terminator, reset tsize to original size and continue.
4. If index contains freed space terminator, check ahead
    a. If next terminator is allocated space terminator, break and use index after allocated space terminator
    b. If next terminator is freed space terminator, break and continue using free space.
5. If tsize = 0, enough space has been found. Set index to allocated space terminator.
    a. Return pointer
6. If no spaces exist for size: communicate overflow error

ex. if each "0" is a byte, '@' is allocated space terminator, '$' is freed space terminator:

0 0 0 0 0 @ 0 0 0 0 $ 0 0 0 @ 0 0 0 0 $ 0 0 0 0 0 $ 0 0 0 0 0,
still has 4 bytes, 15 bytes sections of space. if you need to alloc 3 bytes, it looks like:

0 0 0 0 0 @ 0 0 0 @ $ 0 0 0 @ 0 0 0 0 $ 0 0 0 0 0 $ 0 0 0 0 0,
where now there is 1 byte that is "trapped" (we can't store anything before or after that), but it's still great considering how small our metadata is.

if you want to allocate 5 bytes:
0 0 0 0 0 @ 0 0 0 @ $ 0 0 0 @ 0 0 0 0 0 @ 0 0 0 0 $ 0 0 0 0 0,

if you want to free that second ptr:
0 0 0 0 0 @ 0 0 0 $ $ 0 0 0 @ 0 0 0 0 0 @ 0 0 0 0 $ 0 0 0 0 0,
where now you get that "trapped" byte back.

.free

if ptr is out of bounds of myblock, return invalid pointer.

if index directly before pointer does not contain a terminator, return invalid pointer.

Iterate from pointer towards first terminator after it.

1. if terminator is freed space terminator, return error: already freed pointer
2. if terminator is allocated space terminator, change to freed space terminator and break

.workloads

Test A average runtime: 0.000005 seconds

Test B average runtime: 0.000471 seconds

Test C average runtime: 0.000070 seconds

Test D average runtime: 0.000115 seconds

Test E average runtime: 0.000110 seconds

Test F average runtime: 0.000112 seconds

These were consistently the average runtimes we got for our test workloads over 100 iterations. The thing that stands out to us is test A running so fast. We double checked our block to make sure it was properly allocating and freeing and it was so the runtime is accurate and is simply a really fast implementation of test A. The longest workload was test B and this is quite surprising to be honest. Test B is quite a simple test, similar to A, yet it takes way longer than any of the other tests. We expected tests C and D to take the longest because they allocate a random number of times which should ideally lead to higher runtimes but they never approached the runtime of test B. In fact, test C was *consistently* lower than almost all the other tests apart from A. Tests E and F were also simple tests that we'd expect to take a similar time as A but instead took a similar time as D. Very surprising. We wonder if there is some sort of compiler optimization going on under the hood because on the surface, these results don't make too much sense. Especially test C being so low and test B being so high.