Assignment 1: ++Malloc
Timur Misirpashayev

README

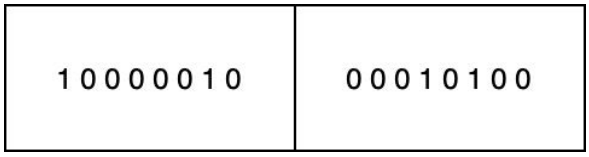
In this assignment, our task was to write our own implementation of the malloc() and free() functions.

Design

One of the challenges of this assignment was finding the most effective way to store the metadata of a memory allocation. Ideally, the metadata should be as compact as possible so as not to occupy too much memory.

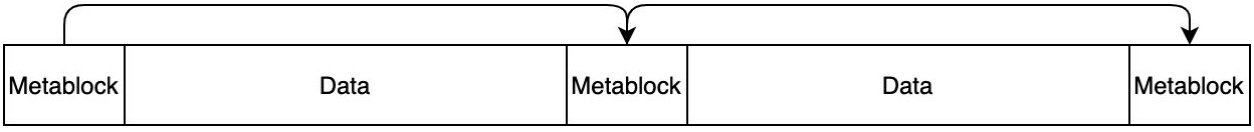
My implementation of malloc() uses 2 bytes (16 bits) of metadata for each allocation. The last 5 bits of the first byte and all 8 bits of the second byte store the size of the allocation (including the metadata size) and the first bit of the first byte marks whether the current memory block is in use. If this bit is set, the current memory block is not available; otherwise, malloc() is free to use it.

For example, if the user requested a malloc() of 532 bytes, on successful allocation, the metablock would contain the following information:

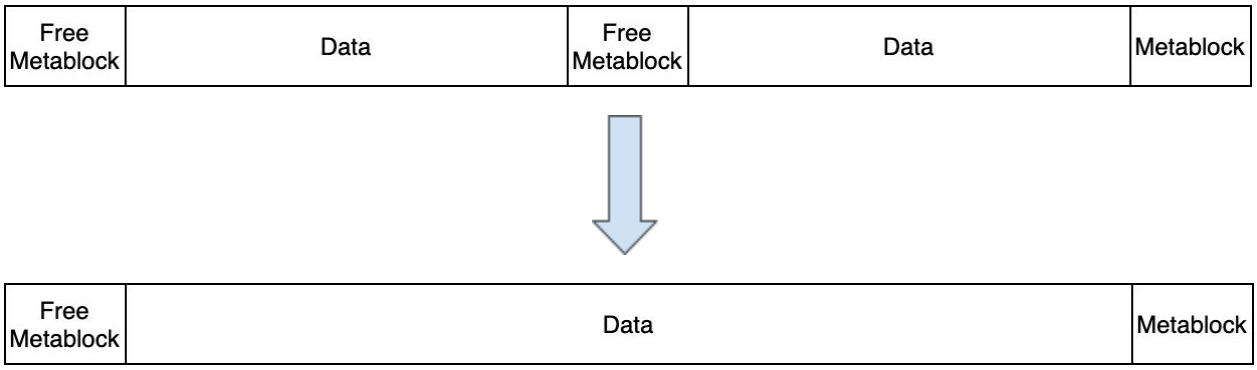


Here, the last 5 bits of the first byte and all 8 bits of the second byte give the binary representation of the decimal number 534 (532 + 2), and the first bit informs malloc() that the 532 bytes following the metadata are currently in use.

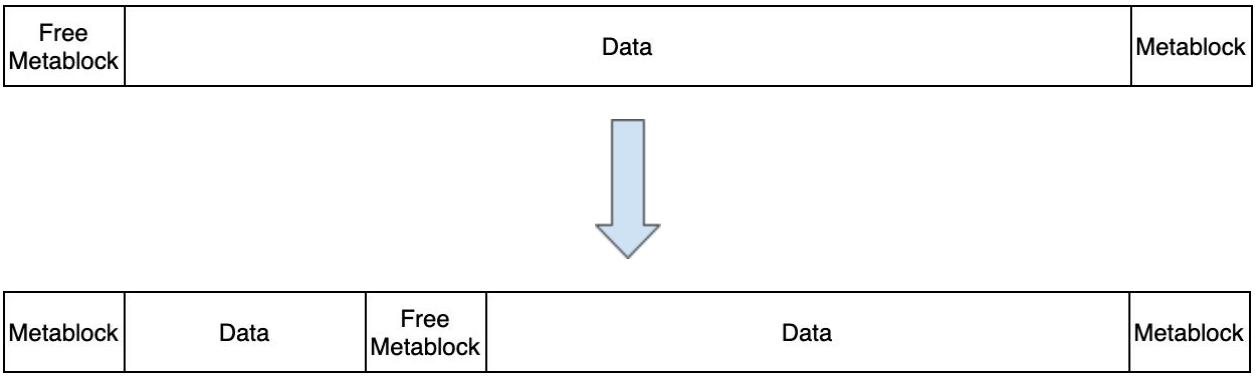
Given that our memory blocks are all contiguous by the first-fit algorithm, this metadata design implies that we can traverse memory blocks just by reading in the value represented by the last 13 bits of metadata and skipping ahead that many bytes.



To free data, free() first determines whether the given pointer is valid, after which it unsets the leading bit of metadata. Neighboring unused memory blocks are then merged so that when the user performs a malloc() in the future, the largest possible amount of contiguous memory is available.



If data is reallocated in an unused memory block, a split is performed if the memory block contains more bytes than necessary.



The only exception is when the new allocation size is 1 less than the original size. In this case, since 1 byte is not enough to store a metablock, the user is given an extra byte and the allocation size remains the same. Another possible solution would be to increase the metadata size to store more information about the memory array and save this byte rather than give it away for free; however, this would not provide any real benefit to the user, as we would be adding at least another byte of metadata in exchange for a byte of memory.

Error Checking

This implementation of malloc() prints informative error messages whenever the user performs an invalid operation. The operations I took into account include:

- Attempting to malloc() more memory than currently available
- Attempting to malloc() 0 memory
- Attempting to free() a NULL pointer
- Attempting to free() a pointer that is out of the bounds of the memory array
- Attempting to free() a pointer that does not point to a block of memory currently in use

In the first two cases, malloc() returns NULL pointers.

Testing

Here is the output of memgrind.c with workloads A-E running on the cheese.cs iLab machine:

The average runtime of workload A is 6.224860 microseconds.
The average runtime of workload B is 108.769340 microseconds.
The average runtime of workload C is 28.168360 microseconds.
The average runtime of workload D is 588.429320 microseconds.
The average runtime of workload E is 243.013040 microseconds.

For timing I decided to use clock_gettime(), a successor to gettimeofday() which is more precise.

Of the three workloads A, B and C, B ran the slowest because it required traversing an entire array of length 120 and merging each byte as it was freed. Although C was similar, randomly freeing char pointers meant that earlier space in the array could be reused, reducing the amount of time needed to find an empty space for allocation. In other words, malloc() would be traversing an array of smaller size.

Workloads D and E were more rigorous and involved lots of merging and splitting. See testcases.txt for more details.