Mymalloc

Objective

To design our own malloc() and free() methods using a contiguous character array as simulated memory.

Design and Implementation

Through this project, our goal was to minimize metadata space as much as possible so that actual usable memory space could be maximized. To achieve minimal metadata space, we initially limit our metadata to a singular byte. With correct configuration and bit manipulation, we can easily get to a point where we can represent memory up to 63 bytes, given only a single byte for metadata. Therefore, for memory sizes greater than 63 bytes, we are forced to use two bytes as metadata.

Metadata

A byte is equivalent to eight bits; therefore, we have eight bits (or flags) that we can define and utilize within our metadata. We dedicate the first bit to represent the "in use" flag, where zero represent a memory chunk that is free, and a one represents a memory chunk that we have already allocated for the user. The following bit has a special purpose: to tell our metadata processor whether we should check the next byte for further metadata. Zero represents metadata of one byte, while a one represents metadata of two bytes. The last six bits in our first byte are used to store the total size of the memory chunk and metadata in binary. Storing the size of the metadata with the memory size allows us to avoid the potential issues of having a free memory chunk of 63 but requiring 2 bytes of metadata. This will be discussed in further details below.

Examples

Assume allocation of 28 bytes:

1 - The "in use" flag is switched on

0 - This bit tells us for future reference that this is only 1 byte of metadata.

[011101] - These 6 bits all represent the number 29 in binary. The final binary value ends up being [01110101]

Assume allocation of 2000 bytes:

1 - The "in use" flag is switched on

1 - This bit is now 1, showing that we need to check the next byte afterwards for metadata as well

[00011111][010010] - These 6 + 8 bits all represent the number 2002 in binary. The final binary value ends up being [0001111101001011]

Free

With the limited metadata space we are working with, we do not have much play in terms of freeing memory; however, our algorithm still allows us to free correctly (and catch any potential errors). Our free() works as follows:

1. Given the pointer to the memory location, we cannot immediately determine the size of the allocated memory due to our dynamic metadata size. To counteract this problem, we loop through the whole memory, and keep a pointer to the start of the metadata and determine where the memory would be. If the pointer to where the memory would be matches with the pointer given to us as input to free(), then we can say with confidence that this is the memory the user wants freed. Now, we have a pointer to the corresponding metadata, and so we flip the "in use" tag to 0 and have now successfully freed the data.

2. The other major benefit of iterating through the memory array is the simplicity of merging two free memory chunks together. While iterating, we also have a pointer to the last iterated free memory chunk. Thus, if the two memory chunks are next to each other (i.e. the pointer of the previous free memory chunk + the size is equal to the pointer of the current free memory chunk), then we can merge the two memory chunks together to avoid heavy fragmentation.

Implementation Caveats

Assume we have 66 bytes of free data with 2 bytes for the metadata for a total of 68 bytes. Assume that the user then decides to malloc() 2 bytes of data (and 1 byte of metadata), leaving us with 65 bytes. Then, we are unable to create the metadata for the remaining 65 bytes. If we choose to use 2 bytes, then the stored memory size is 63, which would not be represented correctly with 2 bytes. If we choose to use 1 byte, then the stored memory size is 64, which would require 2 bytes of metadata to represent.

The previous issue would be solved if we stored total size instead of just memory size. In other words, we would include our metadata size as a part of our memory size. Using the same example:

Assume we have 66 bytes of free data with 2 bytes for the metadata. Our metadata now will say that the space is 68 bytes. Assume the user then decides to malloc() 2 bytes of data, so we lose 3 bytes of data. This leaves us again at 65 bytes. The 65 bytes will have 2 bytes for the metadata, but it will not run into the same issue because it will have 65 stored.

In conclusion, we will never be storing the number 64. This does not mean the user cannot allocate 64 bytes of data, it just means that when the user goes to allocate 63 bytes of data, it will go to 64 bytes in total, which cannot be stored in 1 byte of metadata, so it the total will increase to 65 bytes. This works out because 65 bytes is greater than 64 so it requires those 2 bytes.

Workload and findings

For analysis and comparison, we also ran our memgrind with the build in malloc() function. The results will be listed under the respective workloads under a section named "default time"

A. 8 Microseconds (0.000008 seconds)

- Default Time: 2 Microseconds (0.000002 seconds)
- Our malloc() function seems to be running decently close to that of the default malloc() function

B. 60 Microseconds (0.000060 seconds)

- Default Time: 2 Microseconds (0.000002 seconds)
- Workload B was going to be quite a bit more complicated and time inefficient, and that was certainly reflected in the time. We hypothesize that the increase in time is due to the merging of free memory chunks.

C. 366 Microseconds (0.000366 seconds)

- Default Time: 44 Microseconds (0.000044 seconds)
- Workload C was hard to predict where it would lie in terms of time efficiency. This was partially due to the program randomly choosing between malloc() and free(). However, through this workload, since we always used free() on the last element in the array, we could still see that our free() was slower than the default.

D. 8 Microseconds (0.000008 seconds)

- Default Time: 2 Microseconds (0.000002 seconds)
- Workload D was also surprising at first. We thought that the time would be similar to workload C, as it also had randomization between using malloc() and free(). However, we later came to the realization that since it was terminating as soon as we reached 50 malloc() calls and not 50 stored pointers.

E. 21 Microseconds (0.000021 seconds)

- Default Time: 1 Microseconds (0.000001 seconds)
- Workload E was another case of the merging during the free() call slowing the time down.

F. 2875 Microseconds (0.002875 seconds)

- Default Time: 116 Microseconds (0.000116 seconds)
- This result was extremely surprising to us. This workload highlighted the extreme time inefficiency in our merging and free() functions.

Other Implementations Considered

1. Our first draft was very similar to our current working design. However, at that time we had not considered the potential error highlighted before, so it took us many hours of debugging, followed with a complete rewrite, and finally, discovering and fixing the issue.

2. Our second idea was thought to be extremely space efficient. We were going to pre-allocate 512 bytes (4096 bits). Each bit was to represent a byte in memory, and we would flip a bit to 1 if the respective byte in memory was in use. The issue that came up was that when doing free(), we would have no clue where the memory would end, since we had no other way of knowing the size. Another potential issue with implementation 2 was that it would be extremely inefficient when allocating a large memory sizes. If we were to allocate 3,584 bytes in one malloc() call, then normally we would only need 2 bytes for metadata. Implementation 2 pre-allocated 512 bytes no matter the allocation size, so that would be 512 bytes of metadata for one allocation. Potential benefits of implementation 2 was due to the simplicity. For very small allocation sizes (1 or 2 bytes), we would have a fixed 512 bytes. If we were to allocation 3,584 1 byte malloc() calls, then the 512 bytes would be all we need. However, in our other implementation, we would not even be able to make 3,854 1 byte calls.