README

Design

Our meta-data size is variable. It is two bytes for malloc calls of size>128 and one byte for malloc calls of size<=128. How is explained below.

Our mymalloc function focuses solely on meta-data size and sacrifices some efficiency to do so. The minimal number of bytes needed to represent size's ranging from 1 to 4095 would be 2 because you cant represent 4095 unique values in 1 byte. However if the size our meta-data is representing is 4095 it would take 12 bits to store that number leaving 4 extra bits left over for us to use as we like. We COULD use these bits to represent if the region is free or not or as a check to make sure that this is a proper meta-data tag and not random junk memory. However if we don't need to worry about these things we can actually get our meta-data smaller than two bytes per allocation.

In order to get the meta-data as small as possible we needed to make sure we always knew the state of the array even from the beginning of the program as this allows us to not worry about if a region is free or not and that we aren't looking at junk memory. For this purpose we manually declared the array to have all 4096 entries as 0. Now we are free to use those extra four bits in the pursuit of smaller meta-data. Our program utilizes two different types of meta-data and uses one of the extra bits as a flag to determine which type of meta-data it is. Depending on the size of the region being malloc'd our meta-data is either 1 byte or 2 bytes with the most significant bit of the first byte encountered being 1 for 1 byte meta-data and 0 for two byte meta-data. Given that we sacrifice one bit of our one byte meta-data for the flag we have 7 other bits to represent the size of the region this meta-data represents allowing us to represent 128 unique numbers. Thus when the size being malloc'd is greater than 128 bytes our meta data size is 2 bytes. But when it is less than or equal to 128 our meta-data size is 1 byte.

The two byte meta-data's are interpreted one byte at a time with the first byte only having the 4 least significant bits set which represent the 256, 512, 1024, and 2048 places and the 4 most significant bits being 0. The second byte's bits correspond to their normal place values i.e. 1, 2, 4, 8, 16, 32, 64, 128.

In essence our two byte meta-data is merely a short int stored in big-endian order.

Since we won't allow regions bigger than 4094 to be malloc's (as anything bigger plus the meta-data would exceed the size of our array) we can be sure that those 4 most significant bits of the two byte meta-data will always be 0. Thus whenever we encounter a meta-data whose first byte interpreted as an unsigned char is greater than or equal to 128 we know that the first bit is 1 and therefore it is a 1 byte meta-data and to interpret it as such.

So to recap our meta-data only stores the size of the region it is managing and 1 bit of meta-data about itself. It is also important to note here that this meta-data is stored directly above the region that it manages. The reason why this is all we need is because when the array is initialized we made everything 0 entries and when we free a pointer we zero out its meta-data AND we zero out the following n bytes where n is the size of the region the meta-data was managing. This guarantees that the only non-zero entries in our array will either be meta-data for a block that is currently being managed OR user data that is in a region that is being managed. Because of this when malloc is called we can traverse through our array starting at the top and look for enough consecutive 0 entries to store the meta-data plus the memory the user asked for. If we encounter a non-zero entry before we have

enough space we know that this must be meta-data for an in use block because meta-data always precedes user data and this was the first non-zero entry we found. Since the meta-data we just encountered tells us how large of a region the user data resides in we can then skip that many elements ahead in our array so we never see what is stored in the user-data section. We repeat this until either we found enough consecutive zeros to store the meta-data plus the memory the user requested OR we hit the end of the array. If we found enough zeros we use look at the first 1/2 bytes of this region and set them to be our meta-data for the region then we return the address of the first zero below our meta-data If we hit the end of our array we let the user know that there was not enough space for their allocation and return NULL.

When free is called rather than looking at the entry directly above the pointer they freed (because this could be user data that we have no control over and could easily be interpreted as meta-data) we instead traverse through our array from the beginning and whenever we encounter meta-data we check to see if the address of the first byte after this meta-data is the address we are trying to free, If so we zero out the meta-data and the entire region this meta-data was managing (as was previously stated) If not we skip ahead by the number of bytes that meta-data represents (just like in malloc) and continue traversing. Our traversal stops once we reach the end of the array OR we are looking at an address further down than the address that we are trying to free since we already passed it.

Note that since meta-data is zeroed out when it is freed we do not need to keep track of if the region the meta-data is managing has been freed or not since there is no such thing as meta-data for a free region.

Also note that since we always know they state of the array except for user data stored in it; We can guarantee that the only non-zero thing we see is meta-data by then using that meta-data to skip the regions of memory in which we could possibly find user data.

Workload data (times are in microseconds, and error messages were omitted):

Ran with cmd: ./memgrind | tail -n 6 >> readme.pdf

Run 1:

The average time for taskA to run is: 3

The average time for taskB to run is: 52

The average time for taskC to run is: 3

The average time for taskD to run is: 11

The average time for taskE to run is: 30

The average time for taskF to run is: 154

Run 2:

The average time for taskA to run is: 2

The average time for taskB to run is: 53

The average time for taskC to run is: 5

The average time for taskD to run is: 10

The average time for taskE to run is: 32

The average time for taskF to run is: 290

Run 3:

The average time for taskA to run is: 2

The average time for taskB to run is: 54

The average time for taskC to run is: 3

The average time for taskD to run is: 11

The average time for taskE to run is: 30

The average time for taskF to run is: 76

Ran with cmd: ./memgrind

Run 4:

The average time for taskA to run is: 13

The average time for taskB to run is: 70

The average time for taskC to run is: 5

The average time for taskD to run is: 11

The average time for taskE to run is: 337

The average time for taskF to run is: 8470

Run 5:

The average time for taskA to run is: 13

The average time for taskB to run is: 73

The average time for taskC to run is: 4

The average time for taskD to run is: 11

The average time for taskE to run is: 306

The average time for taskF to run is: 13853

Run 6:

The average time for taskA to run is: 3

The average time for taskB to run is: 52

The average time for taskC to run is: 4

The average time for taskD to run is: 12

The average time for taskE to run is: 340

The average time for taskF to run is: 4813

Findings:

In the first 3 runs, the cmd "./memgrind | tail -n 6 >> readme.pdf" is run, and in the last 3 runs, "./memgrind" is run and the output is copied manually from the terminal to this file.

First, (looking at the piped output, so runs 1-3) tasks B, E and F take comparably more time than A, C and D. This is because E and F do more mallocs/frees, while B does the same number of mallocs and frees as A. However, storing the pointers in an array, and traversing the array adds to the runtime, which explains why B takes more time than A.

Notice that there is a big difference in the average runtime of tasks E and F when you run ./memgrind vs when you run ./memgrind | tail -n 6 >> readme.pdf.

We suspect that this happened because when the file was run without the piping, the error messages (in the form of printf statements) took time to print on the terminal. On the other hand, when the file was run with the piping to tail, there was no need for the terminal to display anything, which drastically reduced the runtime of tasks E and F.

Having malloc and free display error messages and be "nice" when the user does "bad things" is very convenient. Besides the obvious fact that it makes debugging easier, I was able to free arrays of pointers that had already-freed elements in them, rather than removing the elements that have already been freed first before freeing everything else. However, as you can see by the runtimes of task E and F, the error messages significantly increased the runtimes of those tasks, so even if malloc and free were implemented to be nice, it still would not be advisable to run free on already-freed pointers.