CS2110 Fall 2014
Homework 13 - Malloc

This assignment is due by:

Day: Check T-square for most up-to-date due day
Time: 11:54:59 PM

Rules and Regulations

Academic Misconduct
Academic misconduct is taken very seriously in this class. Homework assignments are collaborative. However, each of these assignments should be coded by you and only you. This means you may not copy code from your peers, someone who has already taken this course, or from the Internet. You may work with others **who are enrolled in the course,** but each student should be turning in their own version of the assignment. Be very careful when supplying your work to a classmate that promises just to look at it. If he turns it in as his own you will both be charged. We will be using automated code analysis and comparison tools to enforce these rules. **If you are caught you will receive a zero and will be reported to Dean of Students.**

Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know ***IN ADVANCE*** of the due time supplying relevant documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. No excuses, what you turn in is what we grade. In addition your assignment must be turned in on T-Square. When you submit the assignment you should get an email from T-Square telling you that you submitted the assignment. If you do not get this email that means that you did not complete the submission process correctly. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over T-Square.

3. There is a random grace period added to all assignments and the TA who posts the assignment determines it. The grace period will last at least one hour and may be up to 6 hours and can end on a 5 minute interval; therefore, you are guaranteed to be able to submit your assignment before 12:55AM and you may have up to 5:55AM. As stated it can end on a 5 minute interval so valid ending times are 1AM, 1:05AM, 1:10AM, etc. **Do not ask us what the grace period is we will not tell you**. So what you should take from this is not to start assignments on the last day and depend on this grace period past 12:55AM. There is also no late penalty for submitting

within the grace period. If you can not submit your assignment on T-Square due to the grace period ending then you will receive a zero, no exceptions.
4. Although you may ask TAs for clarification but you are ultimately responsible for what you submit.

Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files.
2. In addition any code you write must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.
3. When preparing your submission you may either submit the files individually to T-Square (preferred) or you may submit an archive (zip or tar.gz only please) of the files.
4. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want.
5. Do not submit compiled files that is .class files for Java code and .o files for C code.

Overview

Warning:
Your submission must compile with our flags or we will simply not grade it and give it a zero. After you submit download your submission again and unzip in a clean directory and build it.

==========

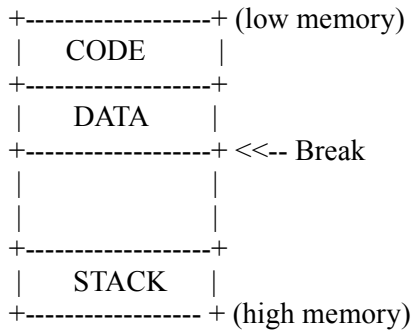
TURN IN THIS ASSIGNMENT ELECTRONICALLY USING T-SQUARE.
SUBMISSIONS WHICH ARE LATE WILL NOT BE ACCEPTED.

==========

For this assignment you will be implementing a memory allocator. A memory allocator is code that is linked with user programs and provides functions to allocate and deallocate blocks of dynamic memory. In other words, you will be writing the heart and soul of the big three dynamic memory allocation functions: malloc(), calloc(), and free(). It is worth noting that the remainder of this document assumes a thorough knowledge of how to use these functions. If you have any questions about them, resolve them *before* attempting to write the assignment.

The Basics

It is the job of the memory allocator to process and satisfy the memory requests of the user. But where does the allocator get *its* memory? Let us recall the structure of a program's memory footprint.



When a program is loaded into memory there are various "segments" created for different purposes: code, stack, data, etc. In order to create some dynamic memory space, otherwise known as the heap, it is possible to move the "break", which is the first address after the end of the process's uninitialized data segment. A function called "brk" is provided to set this address to a different value. There is also a function called "sbrk" which moves the break by some amount specified as a parameter.

For simplicity, a wrapper for the system call sbrk has been provided for you in the file named 'my_sbrk.c'. Make sure to use this call rather than a real call to sbrk, as doing this can potentially cause a lot of problems. Note that any problems introduced by calling the real sbrk will not be regraded, so make sure that everything is correct before turning in.

If you glance at the code for my_sbrk, you will quickly notice that upon the first call it always allocates 8 KB. For the purposes of your program, you should treat the returned amount as whatever you requested. For instance, the first time I call my_sbrk it will be done like this:

```
my_sbrk(SBRK_SIZE); /* SBRK_SIZE == 2 KB */

 ----------------------------------------
|                      8 KB              |
 ----------------------------------------
^
|
`------ The pointer returned to me by my_sbrk
```

Even though you have a full 8 KB, you should treat it as if you were only returned SBRK_SIZE bytes. Now when you run out of memory and need more heap space you will need to call my_sbrk again. Once again, the call is simply:

```
my_sbrk(SBRK_SIZE);

 ----------------------------------------
|    2 KB    |               6 KB          |
 ----------------------------------------
             ^
             |
             `---- The pointer returned to me by my_sbrk
```

Notice how it returned a pointer to the address after the end of the 2 KB I had requested the first time. my_sbrk remembers the end of the data segment you request each time and is able to return that value to you as the beginning of the new data segment on a following call. Keep this in mind as you write the assignment!

Block Allocation

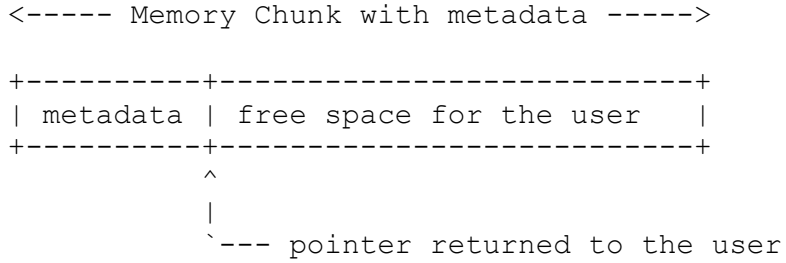
Trying to use sbrk (or brk) exclusively to provide dynamic memory allocation to your program would be very difficult and inefficient. Calling sbrk involves a certain amount of system overhead, and we would prefer not to have to call it every single time a small amount of memory is required. In addition, deallocation would be a problem. Say we allocated several 100 byte chunks of memory and then decided we were done with the first. Where would the break be? There's no handy function to move the break back, so how could we reuse that first 100 byte chunk?

What we need are a set of functions that manage a pool of memory allowing us to allocate and deallocate efficiently. Typically, such schemes start out with no free memory at all. The first time the user requests memory, the allocator will call sbrk as discussed above to obtain a relatively large chunk of memory. The user will be given a block with as much free space as he requested, and if there is any memory left over it will be managed by placing information about it in a data structure where information about all such free blocks is kept. We'll call this structure the free list, and we'll revisit it a little later.

In order to keep track of allocated blocks we will create a structure to store the information we need to know about a block. Where should we put this structure? Can we simply call malloc to allocate space for the information?

Not quite - we are writing malloc to allocate space, and calling malloc inside malloc to get bookkeeping space before we get space for books is a little bizarre, not to mention impossible. This isn't recursion... it's like performing brain surgery on yourself! (Not recommended.)

The trick we will use is that we will store this information, called metadata, about the block inside the block itself! We still want the user to have as much space as he requested, though, so when he wants a block of size x we will actually allocate a block of size "x + sizeof(the metadata) + maybe a little extra for alignment". What this actually looks like:



The user still only sees a block of the size he requested (there may be a bit more in there for alignment, but the user shouldn't *expect* anything more than what he requested), but the metadata hangs out in the space right before that block so our allocation and deallocation algorithms can use it. Now that you know this, you may see why we're so bothered by writing over the bounds of dynamically allocated blocks: write over the metadata and chaos ensues!

Here's a list of the things that your metadata will need to contain:

1) the size of the block (again, this includes the size of the metadata structure itself)
2) whether or not the block is free
3) a pointer to the previous block in the freelist
4) a pointer to the next block in the freelist

For ease of reading, the included struct definition found in 'my_malloc.h' has been pasted below for a better overview of what we're dealing with.

```
typedef struct metadata
{
  short in_use;
  short size;
  struct metadata *prev;
  struct metadata *next;
} metadata_t;
```

The Freelist

When we allocate memory or take pieces of blocks we already allocated, there may be blocks we don't automatically use. For this reason, we keep a structure called the freelist that holds metadata about blocks that aren't currently in use.

The freelist is an array with eight elements - eight lists of blocks of distinct sizes. For the purposes of this assignment, the smallest block should be of size 16 bytes. You should store this array as a file static variable. To help you out we have already defined a variable 'freelist' in the file 'my_malloc.c' for you.

static metadata_t* freelist[8];

Remember that since this variable is declared static it will be automatically initialized to 0 by the compiler for us! This means that we do not have to do any special work to initialized the pointers inside of this array to NULL.

```
This is what your freelist array should look like:
-------------------------------------------
|pointer to free list of blocks of size 16  |
-------------------------------------------
-------------------------------------------
|pointer to free list of blocks of size 32  |
-------------------------------------------
-------------------------------------------
|pointer to free list of blocks of size 64  |
-------------------------------------------
-------------------------------------------
|pointer to free list of blocks of size 128 |
-------------------------------------------
.
.
.
.
-------------------------------------------
|pointer to free list of blocks of size 2048|
-------------------------------------------
```

Since the freelist array actually holds pointers to metadata, and metadata structures all have pointers to the 'previous' and 'next' metadata, the freelist lends itself to a linked list implementation. Keep in mind that 'previous' and 'next' don't necessarily mean consecutive - these pointers just exist to maintain links between blocks that are free. It is okay if a high address comes before a low address in the freelist for any block size, so long as the freelist holds all the blocks that are free at any time.

The Buddy System – Allocating

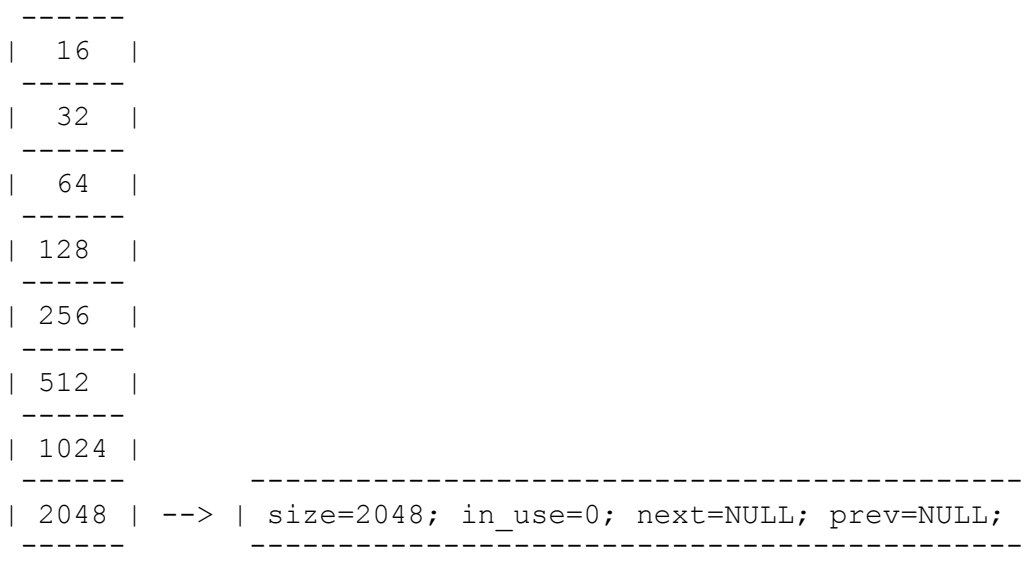
For this assignment we will use an algorithm known as the "Buddy System". This is one of the better known memory management algorithms, despite the fast that it is typically not used for heap management. Although it is very fast, with both allocation and deallocation running in O(1) time, it suffers from the drawback that it introduces high internal fragmentation relative to other possible heap management methods.

When we first allocate space for the heap, it is in our best interest not to just request what we need immediately but rather to get a sizeable amount of space, use a piece of it now, and keep the rest around in the freelist until we need it. This reduces the amount of times we need to call sbrk, the real version of which, as we discussed earlier, involves significant system overhead. So how do we know how much to allocate, how much to give to the user, and how much to keep?

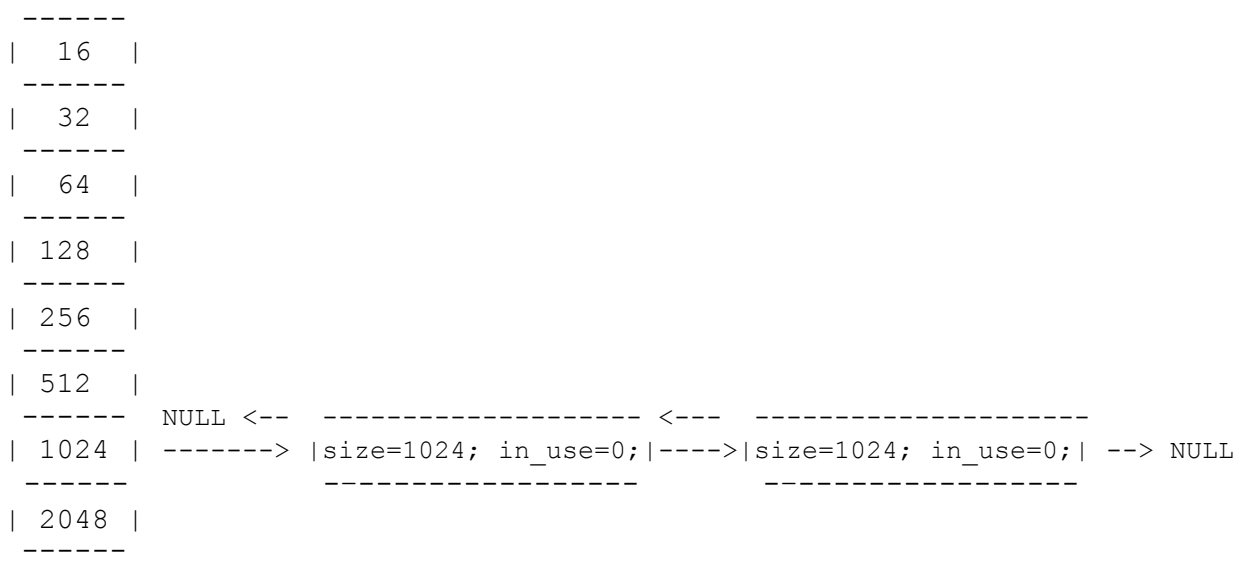
Our freelist for this assignment can hold blocks up to size 2048 bytes, so it would be appropriate when setting up the heap to ask for a block of this size. We don't want to waste space, though, so we want to give to the user the smallest size block in which his request would fit. For example, the user may request 256 bytes of space. It is tempting to give him a block of size 256, but remember we are also storing the metadata inside the block. If our metadata takes up 12 bytes, we need at least a 256+12=268 byte block. We like to align our blocks, so we require they be of the specific sizes that the freelist is designated to hold. The smallest block that can hold 268 bytes of space is of size 512 bytes, so that is what we will actually give to the user.

How do we get from one big free block of size 2048 to the block of size 512 we want to give to the user? This is where the buddy system comes in. If we can't find a free block in the freelist of the size we want, we look at the freelists of increasingly bigger sizes and once we find one with a free block in it, we split it into two equal "buddies" of the next smaller size. If these buddies aren't the size we want, we take one of them and split it again. We keep doing this until we get two buddies each of the size we want, then we take one of them for the user and leave the other in the freelist.

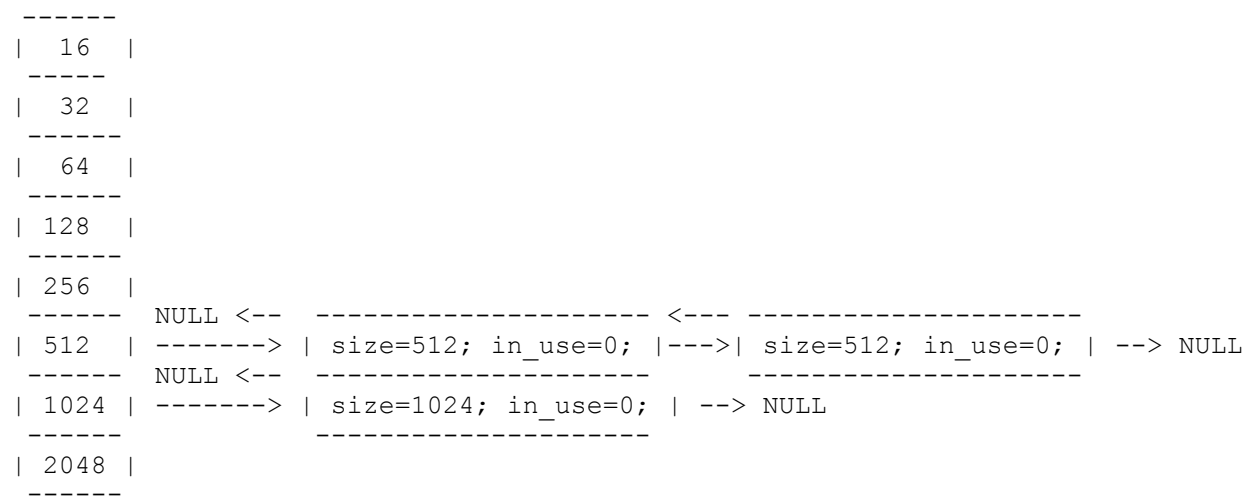
So we start with one big block at the top of the freelist:



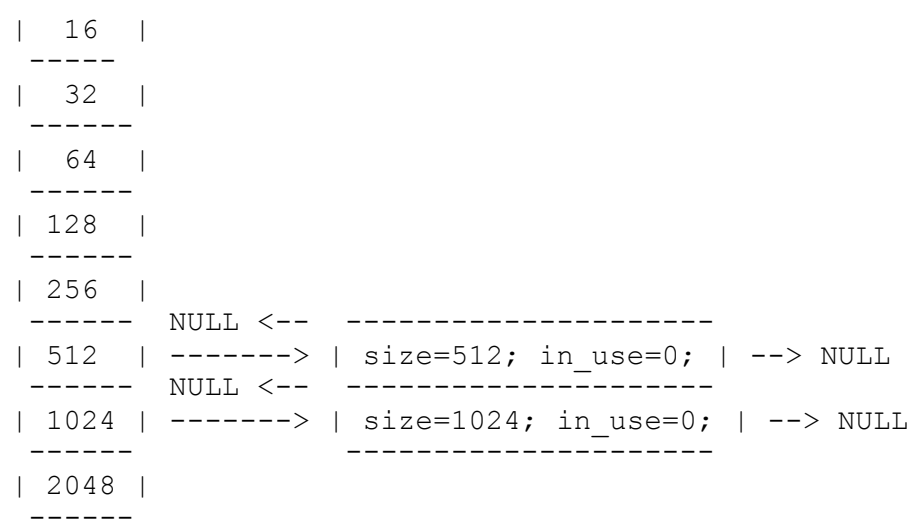
We want a block of size 512, so we look at the smallest size freelist that has a block that we can split, which is the 2048 block. The free list now looks like this after splitting the 2048 block:



However, 1024 isn't the size we want, so we take one of the 1024 blocks and split it in two as well:



Now that there are blocks in the 512 freelist, we can take one of them and give it to the user. The other one stays in the freelist.



The Buddy System – Deallocating

When we deallocate memory, we simply set the block's in_use variable to 0 and return the block to the appropriate free list. (Notice we don't clear out all the data. That really just takes too long when we're not supposed to care about what's in memory after we free it anyway. For all of you who were wondering why sometimes you can still access data in a dynamically allocated block even after you call free on its pointer, this is why!) We like the freelists to contain fairly large blocks so that large requests can be allocated quickly, so if the most recent buddy of a block we're freeing is also free, we can coalesce them, or join them into the bigger block of friendship that they were before we split them.

How do we know who a block's buddy is? Well, there is a reason that this allocator deals with sizes that are powers of 2. Let's look at some binary, and perhaps this will give some insight on the matter.

Assuming the addresses start at 0, lets look at the numbers if we increment each by 4

00000000 -> 0
00000100 -> 4
00001000 -> 8
00001100 -> 12
00010000 -> 16
00010100 -> 20
00011000 -> 24
00011100 -> 28

Looking at the binary above, we're going to see a pattern if we look closely at bit positions 2 and 3 (note these are 0-based positions starting from the right side). If you look at the second position, you will notice that moving between numbers the only thing that differs is that the bits alternate between '0' and '1'. Looking at the 3rd position, we can break up the sequence into pairs of '0's and '1's.

*
00000000 -> 0 these two share a '0' in position 3
00000100 -> 4

*
00001000 -> 8 these two share a '1' in position 3
00001100 -> 12

*
00010000 -> 16 these two share a '0' in position 3
00010100 -> 20

*
00011000 -> 24 these two share a '1' in position 3
00011100 -> 28

Out of these pairs, it is possible to determine which one we are looking at by peeking into the 2nd bit position. Notice how the first block on the pair always has a '0' in this position, and the second block always has a '1'. This works just as nicely for blocks of a size that is any power of 2 - the pattern can be generalized as follows:

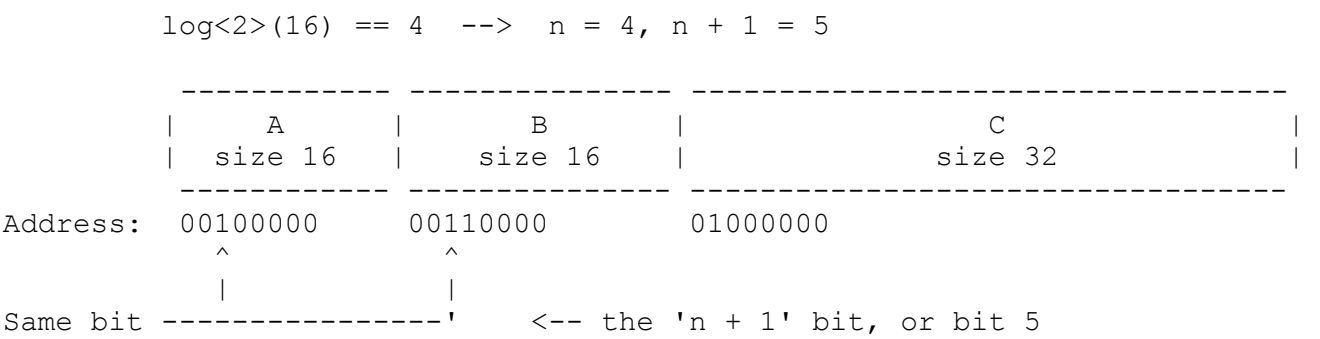
1) let 'n' be log base 2 of the block size (e.g. blocks of size 4 would have an 'n' of 2, as we just saw; blocks of size 16 would have an 'n' of 4, etc.)
2) to determine pairs, we can look at bit position 'n + 1' and make sure that the bits match up (e.g. in the case of #1, we would look at bit 'n + 1' == 5 to make sure they match up)
3) to determine which address in the pair we are looking at, we can look at bit position 'n'. if we have a '1', then we are looking at the second in the pair. if we have a '0', then we are looking at the first in the pair.

So in other words, these are the rules of how we determine who are "buddies", where a is the first in the pair and b is the second:

1) they are sequential in order. This means that: address(b) == address(a) + sizeof(a)
2) they are of the same size
3) they are paired correctly (e.g. they have the same 'n+1'st bit)

An example:

log<2> means "log base 2"



(*) - in the above picture we may safely determine that block A & B are buddies, since they satisfy all 3 requirements.

However, this relies on the assumption that the first address starts at 0. In order to do all of this buddy matching, we must make the addresses start at 0 for computation purposes. We can do this by subtracting the offset of the heap (returned by the first call to my_sbrk) from addresses. If the first call to my_sbrk returned 0x9bd0170, then we should subtract 0x9bd0170 from an address, do the bit manipulation, and then add 0x9bd0170 back to the result to get the appropriate buddy.

Since we determine buddies using addresses, this algorithm is remarkably simple! We don't need to search through the freelist to find the buddy – all we need to do is calculate the buddy's address based on the address we are freeing, see if the buddy is free, and coalesce the two if possible. Hence buddy system deallocation runs in O(1) time.

my_malloc

You are to write your own version of malloc that implements buddy system allocation. To jog your memory, this is what that entails:

1) Figure out what size block you need to satisfy the user's request. Remember that the 'size' field in our metadata structure includes the size of the metadata structure itself, so take the user size and sizeof(metadata_t) into account as well as the legal sizes a block can be.

2) Now that we have the size in units we care about, we need to index into our freelist array at the appropriate index to see if there is a free block in that list. If there is one, we need to remove it from the freelist, mark its in_use flag to 1, and return the block to the user. Note that the user should not be returned the pointer to our metadata structure, so the pointer we actually return to the user is 'metadata_pointer + sizeof(metadata_t)'.

Another note: remember that you want the address you return to be 'sizeof(metadata_t)' *bytes* away from the metadata pointer. Pointer arithmetic can be unforgiving - if our pointer was of type int* (which it's not, this is just an example), adding 4 to it would result in an address that is 4 ints away from the original address. ints are not byte sized (ha), so maybe a cast should be involved?

3) If the list we indexed into in the above step is empty, then we need to start iterating the lists of larger size in order. Once we find a list that has a block which is free, we then need to begin splitting the block in half, placing both halves into the freelist preceeding its own. We continue this process until we arrive back at the list which can satisfy the users request. We then return one of the halves, and leave the other half alone.

HINT: Think recursively!!! This is one of those algorithms that can be written much more easily using recursion.

A couple of other things that are worth noting for the my_malloc function.

1) The first call to malloc should call my_sbrk to set up the heap.

2) If the user request exceeds 2048 bytes (note that this number is what is calculated *AFTER* my_malloc adds on the obligatory sizeof(metadata_t)), then return Null and set the error code.

3) In the event that there are no blocks currently available in the freelist that satisfy the users request (note that you must attempt the procedure described above before determining this), then you should issue a call to my_sbrk to expand the heap size by SBRK_SIZE bytes. Failure to use this macro to expand the heap may result in a lower grade than you think you deserve. Please make *SURE* to use this macro when calling my_sbrk to avoid any possible problems. Also note that in the event that my_sbrk returns failure, you should return NULL and set the error code.

my_free()

You are also to write your own version of free that implements buddy system deallocation. This means:

1) Calculate the proper address of the block to be freed, keeping in mind that the pointer passed to any call of my_free is the pointer that the user was working with, not the pointer to the block's metadata.

2) Attempt to merge the block with its buddy if its buddy is free. If this merge is successful then your algorithm should attempt to merge the newly created block with its buddy. This process of merging should continue until either you have just merged a block into the largest possible block size, or an attempt to merge buddies fails (this will occur if the block's buddy is not free). You must also remove any newly merged blocks from whatever linked lists they were in. Remember the case where the user tries to free memory already in the free list. This should set the error code.

my_memmove()

This function takes in two pointers, a source and a destination as well as a size_t that indicates the number of bytes to be copied. The source and destination could overlap, so it is important to copy as if you copied all the number of bytes to be copied from the source to a temporary buffer and then copied to the destination. For example, if memory looks something like this:

0xf000	f001	f002	f003	f004	f005
1	2	3	4	5	6

The above is byte-adressed so each memory location holds 1 byte. Now say we have a pointer src pointing at 0xf000 and a pointer dest pointing at 0xf002 and we specify that we want to copy 3 bytes, that is, at the end of the memmove, ther array should looke like this:

0xf000	f001	f002	f003	f004	f005
1	2	1	2	3	6

The 1,2,3 in f000, f001, f002 respectively got copied over, though at the end of the process, part of the source changed.

If you, in your code, do a memcpy instead of memmove, it will probably look ike this:

0xf000	f001	f002	f003	f004	f005
1	2	1	2	1	6

This behavior is wrong, it discards the 3!

It is highly recommended that you go and gain a thorough understanding of memmove by testing out the actual memmove function in C.

my_calloc()

This function is used to dynamically allocate entire arrays at a time. It takes in the number of elements to be allocated in the array as well as the size of each element. It will use malloc to allocate a stretch of dynamic
memory appropriately sized for the user. Notet that calloc also initializes this stretch of memory to 0.

Should the malloc call fail, it should return NULL and set the proper error code.

Error codes

For this assignment, you will also need to handle cases where users of your malloc do improper things with their code. For instance, if a user asks for 12 gigabytes of memory, this will clearly be too much for your 8kb heap. It is important to let the user know what he or she is doing wrong. This is where the enum in the my_malloc.h comes into play. You will see the four types of error codes for this assignment listed inside of it. They are as follows:

NO_ERROR: set whenever my_calloc, my_malloc, my_free, or my_memmove complete successfully.
OUT_OF_MEMORY: set whenever the user request cannot be met b/c there's not enough heap space
SINGLE_REQUEST_TOO_LARGE: set whenever the user's request is beyond our biggest block size
DOUBLE_FREE_DETECTED: set whenever the user attempts to free memory already freed.

Inside the .h file, you will see a variable of type my_malloc_err called ERRNO. Whenever any of the cases above occur, you are to set this variable to the appropriate type of error. You may be wondering what happens if a single request is too large AND it causes malloc to run out of memory. In this case, we will let the SINGLE_REQUEST_TOO_LARGE take precedence over OUT_OF_MEMORY. So in the case of a request of 9kb, which is clearly beyond our biggest block and total heap size, we set ERRNO to SINGLE_REQUEST_TOO_LARGE.

Code Documentation

YOU MUST COMMENT YOUR CODE for this assignment! If you do not have any comments in your code, then you will lose points for it. We don't expect every line of code to have comments, but we expect that you will have some comments so that we can figure out what you are trying to do in your code.

An example:

```
void* my_malloc(size_t size) {

      /* Declaration of variables*/
      int a,b;
      int *c;

      /* Looks for buddy */
      ...your code here...

      /* Merges 2 blocks back together */
      ... your code for de-allocation here...
}
```

The Assignment

1) You must use the buddy system exactly as described in this file.

2) You may submit multiple files. The functionality for my_malloc, my_calloc, my_free, and my_memmov should reside in the provided file 'my_malloc.c' file.

3) You must also implement a basic linked list for ints that uses your my_malloc code for memory allocation. Thie provided 'list.h' and 'list.c' files are included for doing this.

4) We have provided you with a blank tester file (test.c). In your last homework, all of your tester code was supplied to you. This time you will not be so lucky. Naturally, we encorage you to test your code, but we also want you to learn the skills required to make that testing possible. The supplied Makefile assumes you will use test.c to test your code. If you choose to use a different file you will have to modify your Makefile to do so. Note that we are asking you to submit a library so no file with a main function should be submitted (No need to submit test.c). But more importantly, make sure there is no main file in any of your other code (ex:my_malloc.c). If any linking s are encountered while attempting to grade, including 'multiple definition of main', you *WILL*
receive a ZERO.

5) You may NOT core dump (cause a segmentation fault).

6) You may NOT maintain the heap in such a way that normal usage of your library would cause a core dump (ie, returning a block of size 4 when a block of size 32 was requested, returning a block which isnt properly aligned, etc.)

7) You may not call any C library functions which are in any way related to memory management (ie malloc, calloc, etc.). You may, however, use the math library which is included for you with the -lm flag.

8) Your code must compile cleanly the flags provided.
gcc -ansi -pedantic -Wall -O2 -Werror <files>
If your code does not compile with these flags, or if it cannot be compiled using the Makefile, you WILL receive a ZERO.

9) You should turn in all of your files. Above we told you there was no need to turn in test.c, but if you are having trouble with the assignment it would not hurt for the grader to see your test code (however, this is not a requirement).

10) Test as much as you want - just make sure no tester output comes out when we run your library files. Bon courage!

11) Be sure to comment your code thoroughly! You will lose points if it is not commented!

12) Deliverables
- Makefile
- my_malloc.c
- my_malloc.h
- my_sbrk.c
- list.c
- list.h

FAQ

1. Does the metadata_t associated with ptr have a size of 18 + sizeof(metadata_t)? Or does it just store 32, the entire size of the block?
   ie: void* ptr = my_malloc(18);

   The total size or capacity of block, will be a power of 2. It has nothing to do with how much the user allocated. So in the above example, the block would be of size
   32 even though the user requested 18.

2. Why am I getting a segfault in....
   Use GDB

3 The hw assignment says to just call my_sbrk again. But wont this mean we then have 2 heaps?

   Not exactly, it will expand the heap by another 2KB. You don't get two heaps. Once it has been expaned to 8KB, calls to my_sbrk will return -1

4 When splitting a block into two half-sized blocks, do we use malloc to create a metadata for the second block?

   The use of the already coded malloc in C is expressly forbidden! *If you made a call to malloc in order to do this then on the first allocation you'd set up the heap right?*

   Now assuming you'd do that the size requested will most likely be a block BIGGER than what they've requested so you'd call split right?

   *If your split calls malloc in order to perform the splitting then you will call malloc again. Here you pass in the size of the metadata and you'd want a block that is the smallest block that fits the effective size of 2 * size of the metadata However the only block available is the 2048 block so now you have to call split to split that 2048 block. To split a block you need to find the "buddy" of the block.*
   Meaning you simply need to find a pointer to halfway into the memory of the block.

5. Can we build our freelists with list heads/dummy nodes?
   No. No dummy nodes. The autograder checks the state of the freelist and if you have dummy nodes it will throw it off.

6. Should we first initialize the freelist to NULL?
   No, it is static and is therefore already NULL