Malloc + Free

Sunny Chen and Michael Zhao

Program Description:

This program implements a dynamic memory allocator, malloc() and a deallocation function free() with error support, including preprocessor printf directives, for:

- Freeing addresses that are not pointers
- Freeing pointers not allocated by malloc()
- Redundant freeing of the same pointer
- Saturation of dynamic memory (the maximum memory we can allocate is 4094 bytes)

The included files are:

- `mymalloc.c`
- `mymalloc.h`
- `memgrind.c`
- `readme.pdf`
- `testcases.txt`
- `Makefile`

Usage

You can build and run the included workloads in memgrind.c using:

```
$ make && ./memgrind
```

You can clean the generated binary files with `make clean`.

All of our code for malloc() and free() is included in *mymalloc.c* and *mymalloc.h*. This library will automatically convert calls to malloc() and free() to use our custom functions instead. To build another program with these functions, include "mymalloc.h" and run:

```
$ gcc <source files> mymalloc.c -o <binary file>
```

mymalloc()

`void *mymalloc(size_t size, const char* file, int line)`

Description:

Allocates a memory block within the 4096 byte myblock char array if possible and prints an error message is unable to do so. Can allocate a maximum of 4094 bytes.

Return Values:

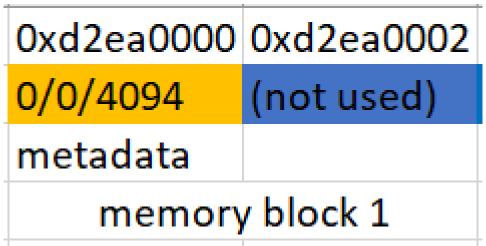
Returns a pointer to a memory block of the requested size if possible and NULL if unable to do so.

Operation:

We simulate main memory using a static char array of 4096 bytes, and all memory allocations through mymalloc use this array.

Our mymalloc function must efficiently allocate free memory blocks when called. We must keep track of which blocks are available and their sizes at any given time, without using any memory outside of our 4096 byte char array. To accomplish this, our program uses a 2 byte metadata header stored at the beginning of blocks to track the status of memory blocks in our array. This contains all necessary information about the block (size, used/unused, and number of hanging bytes). An in-depth description can be found below in our metadata header section.

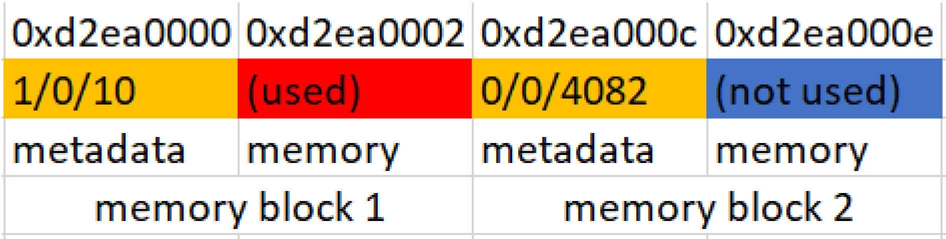
The first time mymalloc is called, we initialize the entire char array as a free block, using the first two bytes of myblock for metadata. This initializes our array so we can apply our normal allocation procedure, as demonstrated below. Note: the hex addresses point to the start of the block in question; the starting address for myblock is arbitrary; metadata is displayed in the format used/hanging/size.



0xd2ea0000	0xd2ea0002
0/0/4094	(not used)
metadata	
memory block 1	

Our memory blocks can be thought of as nodes in a linked list. However to save space, pointers to adjacent nodes aren't stored in the metadata (pointers can be 8 bytes, too large). Rather, we can reach the next memory block by incrementing our current address by the size of our metadata (2 bytes) and the size of our current block.

When searching for a freed block to use, mymalloc() uses the first free algorithm. It returns a pointer to the first free block large enough to accommodate the requested number of bytes. mymalloc() traverses our main memory block-by-block, checking if each one is unused and larger than the request size. If not, it continues to the next memory block, traversing the entire array if necessary and printing a "not enough memory" error if malloc cannot accommodate the request. If it finds a suitable block, it will modify the metadata to mark it as "used" and adjust the size to allocate the desired number of bytes, before allocating the remaining bytes as a new free block independent of the previous block. Continuing with previous example, if we call `mymalloc(10)`, our myblock array would now look like this:

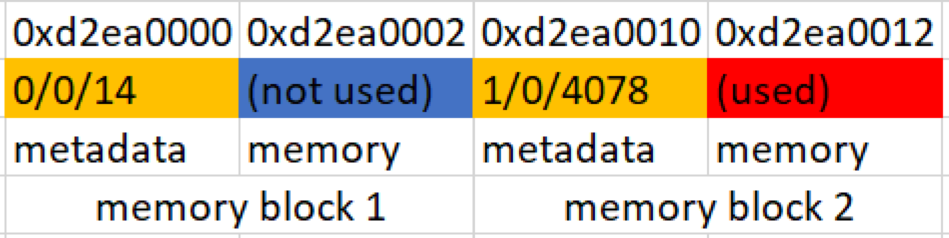


0xd2ea0000	0xd2ea0002	0xd2ea000c	0xd2ea000e
1/0/10	(used)	0/0/4082	(not used)
metadata	memory	metadata	memory
memory block 1		memory block 2	

Recall since curr points to the metadata header, in reality we return the memory address by returning `curr+META_SIZE`. In this example, curr would be 0xd2ea0000, and mymalloc returns 0xd2ea0002.

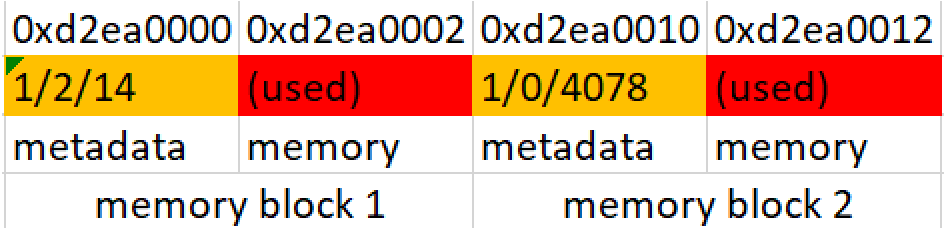
An edge case arises when the remaining number of bytes is insufficient to hold another separate block, since a free block must be at least 3 bytes large, 2 bytes for a metadata header and 1 byte of space (no point having a block of size 0). We call these "hanging bytes", and we just store them with the current block instead of putting them in a new block.

For example, assume we have the following blocks as shown:



0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012
0/0/14	(not used)	1/0/4078	(used)
metadata	memory	metadata	memory
memory block 1		memory block 2	

And we call `malloc(12)`. The resulting memory will now look like:



0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012
1/2/14	(used)	1/0/4078	(used)
metadata	memory	metadata	memory
memory block 1		memory block 2	

When we call `malloc(12)`, our first free algorithm would use the first available memory block to fulfill the request, which is the 14 byte block. We leave the size as 14 rather than 12 since we have 2 hanging bytes (there's not enough room to create a block to store 2 bytes). We update the metadata of memory block 1 to indicate that it contains 2 hanging bytes. This behavior will become useful in our myfree function.

myfree()

```
void myfree(const void *addr, const char *file, int line)
```

Description:

Frees the memory block at addr if it was allocated through `mymalloc()` and is currently being used. Prints error messages if any of those conditions aren't met.

Return Values:

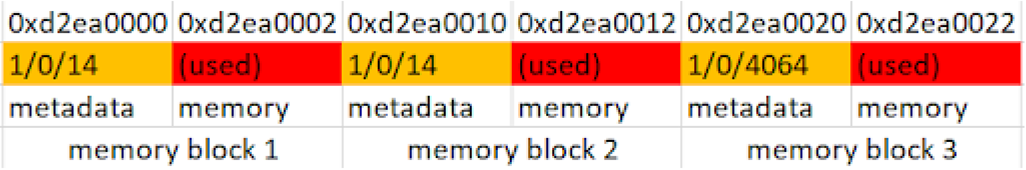
None

Operation:

Our `myfree()` function is responsible for freeing memory, preventing memory fragmentation, and gracefully handling errors (Error handling will be discussed in its own section).

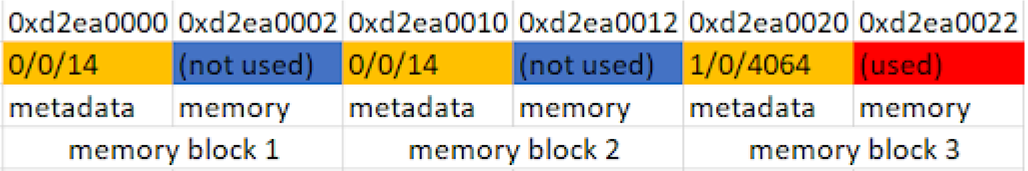
`myfree()` will search for the memory block it should free by traversing blocks using two pointers, `curr` and `prev`. `prev` will always point to the previous block while `curr` points to the memory block currently being checked. We keep traversing myblock while the `curr` pointer is less than the target pointer to be freed. Once the correct memory block is found, we free it by modifying its metadata: setting the used field to false and and setting hanging bytes to 0.

One problem we encountered early on was memory fragmentation. For example, consider we have the following memory:



0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012	0xd2ea0020	0xd2ea0022
1/0/14	(used)	1/0/14	(used)	1/0/4064	(used)
metadata	memory	metadata	memory	metadata	memory
memory block 1		memory block 2		memory block 3	

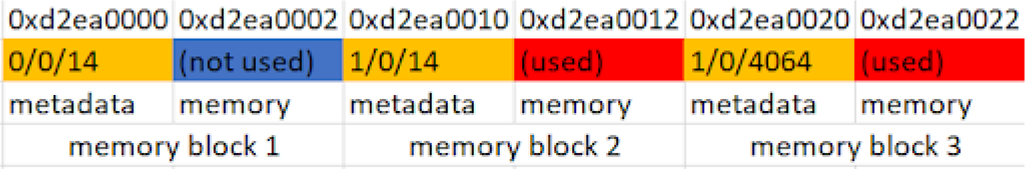
Calling `free(0xd2ea0002)` and `free(0xd2ea0012)` on the above would result in the following changes:



0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012	0xd2ea0020	0xd2ea0022
0/0/14	(not used)	0/0/14	(not used)	1/0/4064	(used)
metadata	memory	metadata	memory	metadata	memory
memory block 1		memory block 2		memory block 3	

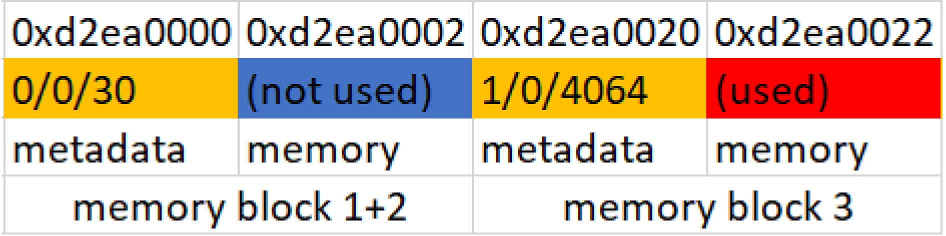
A `malloc(30)` would fail here, even though memory blocks 1 and 2 are both freed and have enough room to store 30 bytes together. This is known as memory fragmentation and can fix this by merging adjacent freed blocks. After we free a block, we check it's neighboring blocks behind and ahead to see if they are freed, and merge them if possible.

Using our initial example, first calling `free(0xd2ea0002)` would result in the following:



0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012	0xd2ea0020	0xd2ea0022
0/0/14	(not used)	1/0/14	(used)	1/0/4064	(used)
metadata	memory	metadata	memory	metadata	memory
memory block 1		memory block 2		memory block 3	

However, the next call `free(0xd2ea0012)` will free memory block 2 and also check if the previous block (mem block 1) is free. Since it is, memory blocks 1 and 2 will be merged:

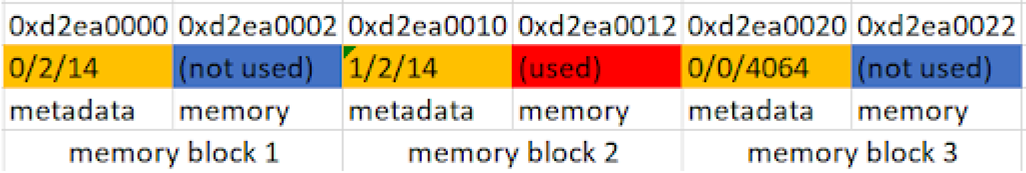


0xd2ea0000	0xd2ea0002	0xd2ea0020	0xd2ea0022
0/0/30	(not used)	1/0/4064	(used)
metadata	memory	metadata	memory
memory block 1+2		memory block 3	

This allows a future `malloc(30)` call to succeed.

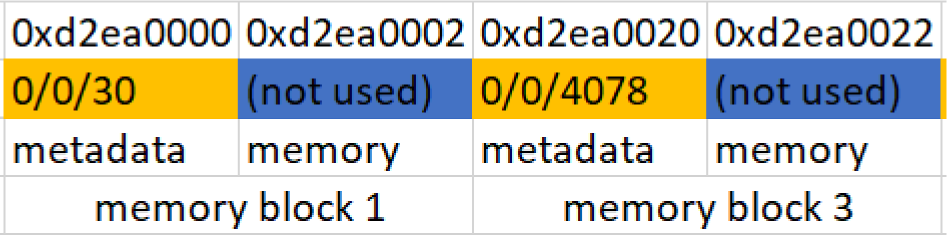
Additional Example:

myfree is also able to free a used block sandwiched between two unused blocks, as seen below:



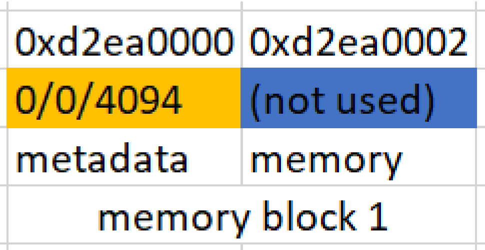
0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012	0xd2ea0020	0xd2ea0022
0/2/14	(not used)	1/2/14	(used)	0/0/4064	(not used)
metadata	memory	metadata	memory	metadata	memory
memory block 1		memory block 2		memory block 3	

In this case, `free(0xd2ea0012)` would first check if prev is unused. Since it is, memory blocks 1 and 2 would be merged (`block1.size = block2.size + metadata_size`) to give the following:



0xd2ea0000	0xd2ea0002	0xd2ea0020	0xd2ea0022
0/0/30	(not used)	0/0/4078	(not used)
metadata	memory	metadata	memory
memory block 1		memory block 3	

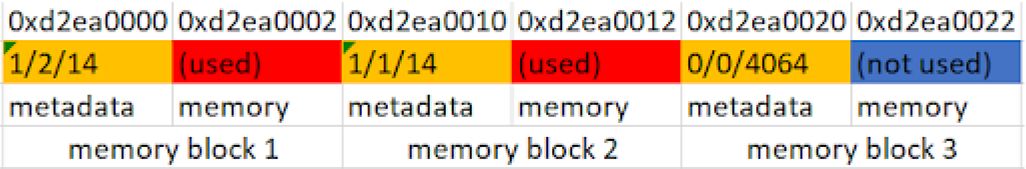
We also check if the next block is unused. Since it is, memory blocks 1 and 3 are also merged. At the end of our `myfree()` call, our memory will look like this:



0xd2ea0000	0xd2ea0002
0/0/4094	(not used)
metadata	memory
memory block 1	

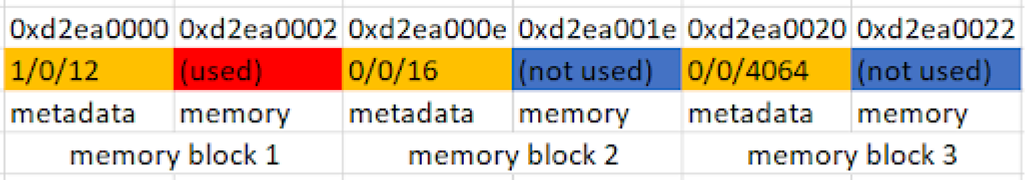
Handling Hanging Bytes:

Lastly, we'll discuss the reason we keep track of hanging bytes. Consider the following situation.



0xd2ea0000	0xd2ea0002	0xd2ea0010	0xd2ea0012	0xd2ea0020	0xd2ea0022
1/2/14	(used)	1/1/14	(used)	0/0/4064	(not used)
metadata	memory	metadata	memory	metadata	memory
memory block 1		memory block 2		memory block 3	

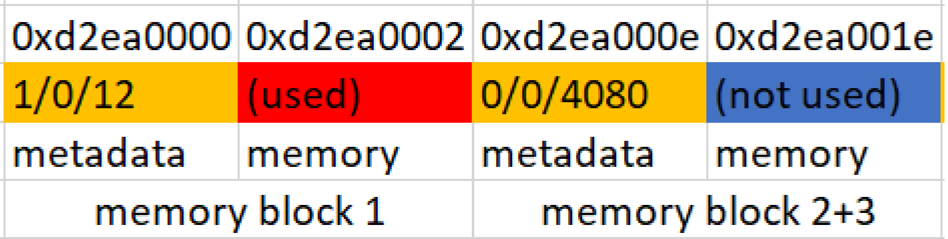
Memory blocks 1 and 2 both have hanging bytes, 2 and 1 respectively. When we `free(0xd2ea0012)`, we check the previous memory block for hanging bytes, resulting in the changes below:



0xd2ea0000	0xd2ea0002	0xd2ea000e	0xd2ea001e	0xd2ea0020	0xd2ea0022
1/0/12	(used)	0/0/16	(not used)	0/0/4064	(not used)
metadata	memory	metadata	memory	metadata	memory
memory block 1		memory block 2		memory block 3	

Since there are two hanging bytes in block 1, we shift the metadata of block 2 left two bytes so that memory block 2 now has 2 extra bytes of free memory (its size grew from 14 to 16). Notice that the hanging bytes and size of memory block 1 have both been decreased by 2, indicating that it has been resized. Resizing a block this way is safe, since those 2 bytes were never used by the user in the first place (mem block 1 was created with the call `malloc(12)`, leaving 2 hanging bytes).

Finally, we'll also check mem block 3 when freeing.



0xd2ea0000	0xd2ea0002	0xd2ea000e	0xd2ea001e
1/0/12	(used)	0/0/4080	(not used)
metadata	memory	metadata	memory
memory block 1		memory block 2+3	

Being able to reclaim these “hanging bytes” allows us more memory to work with.

Error Handling

Our program is able to catch common errors and terminate gracefully, while printing an error message. The following is a list of all possible error messages and an explanation for each.

mymalloc():

- size must be non-zero - mymalloc() does not support a requested size of 0. If it receives this request, it will return NULL and print an error message informing the user that 0 bytes cannot be allocated.
- not enough memory - this error occurs when mymalloc() is asked to allocate too much memory. mymalloc() will always iterate through all memory blocks until it finds one large enough to support the request. If it reaches the end of the array without finding such a block, mymalloc will return a NULL pointer and print an error message.

myfree():

- nothing allocated - myfree() will check if mymalloc() has been called by checking if the first block has a size of 0 (which should never happen if malloc has been called). If mymalloc hasn’t been called yet, there are no memory block to be freed.

- invalid pointer - myfree() will also check for a NULL pointer argument or pointers outside the range of addresses for the array. In both of these cases, the user is attempting to free an address that is not a pointer to our memory.

- block already freed - the user attempts to free a block that has already been freed, which shouldn’t happen.

- pointer doesn’t point to start of block - if myfree() iterates through the entire memory without finding the pointer in question, it must have been misaligned (not pointing to the start of a memory block). This falls under the case of a pointer not allocated by malloc.

Metadata Design

Our metadata is stored at the beginning of each block. We prioritized decreasing metadata size to maximize the amount of memory available to mymalloc(). Our metadata consists of 3 pieces of information:

- Block is used/unused
- Block size
- Number of hanging bytes (explained later)

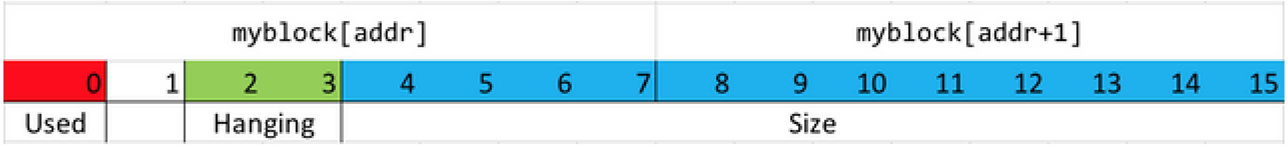
Block usage (1 bit) - only requires 1 bit of information, a 0 represents a free block while a 1 means it's in use.

Block size (12 bits) - requires at most 12 bits of info. The maximum block size possible is 4096 (size of our memory) and $\log_2 4096 = 12$.

Number of hanging bytes (2 bits) - is either 2, 1, or 0, so we only need 2 bits to store this.

Data Representation

We only need 1+12+2=14 bits to store our metadata, it's possible to fit this within 2 bytes of our main memory. We accomplish this using the following format



myblock[addr]								myblock[addr+1]							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Used		Hanging		Size											

For example, let's take a look at sample metadata stored across two bytes at `myblock[addr]` and `myblock[addr+1]`:

```
myblock[addr] = 0b01011100
myblock[addr+1] = 0b11011001

used = 0b0 = 0 (free)
hanging = 0b01 = 1 byte
size = 0b1100 11011001 = 3289 bytes
```

Storing and reading this information from memory requires bit manipulation. To make this easier, we used getter and setter functions for each of the 3 types of meta info. All setter functions make sure not to modify the metadata bit outside their specified range.

`get_used()` and `set_used()`

`get_hanging()` and `set_hanging()`

`get_size()` and `set_size()`

NOTE: All addresses given to getter and setter functions **MUST** point to start of metadata, **NOT** start of blocks.