CS 214 Spring 2023
Project I: My little `malloc()`

David Menendez

Due: February 23, 2021, at 11:59 PM (ET)

For this assignment, you will implement your own versions of the standard library functions `malloc()` and `free()`. Unlike the standard implementations, your versions will detect common usage errors and report them.

For this assignment, you will create

1. A library `mymalloc.c` with header `mymalloc.h`, containing the functions and macros described below.
2. A program `memgrind.c` that includes `mymalloc.h`.
3. Additional test programs that include `mymalloc.h`, along with the necessary Make-files for compiling these programs.
4. A test plan, describing:
   (a) What properties your library must have for you to consider it correct
   (b) How you intend to check that your code has these properties
   (c) The specific method(s) you will use to check each property

   See section 4 for more discussion.
5. A README file containing the name and netID of both partners, your test plan, descriptions of your test programs (including any arguments they may take), which of your design properties you were able to prove, and any design notes you think are worth pointing out. This should be a plain text file or a PDF.

Submit all files to Canvas in a single Tar archive.

1 Background

`malloc()` and `free()` are used to manage the heap: a program's primary location for dynamically allocated data. The versions in the C standard library obtain regions of memory from the operating system and then portion pieces of it as requested. To do so, they maintain a data structure indicating (1) how memory has been divided into chunks and (2) which chunks are currently in use (allocated)

or not in use (free). This *metadata* is itself dynamically sized, since its complexity depends on the number of chunks that memory has been divided into, so it is also stored in the heap.

`malloc()` is called with a single integer, indicating the requested number of bytes. It searches for a free chunk of memory containing at least that many bytes. If it finds a chunk that is large enough, it may divide the chunk into two chunks, the first large enough for the request, and returns a pointer to the first chunk. The second chunk remains free.

`free()` is called with a single pointer, which must be a pointer to a chunk created by `malloc()`. `free()` will mark the chunk free, making it available for subsequent calls to `malloc()`.

1.1 Memory fragmentation

Consider a case where we repeatedly call `malloc()` and request, say, 20-byte chunks until all of memory has been used. We then free all these chunks and request a 40-byte chunk. To fulfil this request, we must *coalesce* two adjacent 20-byte chunks into a single chunk that will have at least 40 bytes.

Coalescing can be done by `malloc()`, when it is unable to find space, but it is usually less error-prone to have `free()` automatically coalesce adjacent free chunks.

1.2 Your implementation

Your `mymalloc.c` will allocate memory from a global array declared like so:

```
static char memory[4096];
```

The use of `static` will prevent client code from accessing your storage directly. You are free to name the array something else, if you prefer. It is recommended that you use a macro to specify the array length and use that macro throughout, to allow for testing scenarios with larger or smaller memory banks.

Your `malloc()` and `free()` functions MUST use the storage array for all storage purposes. You may not declare other global variables or static local variables or use any dynamic memory features provided by the standard library.

In other words: both the chunks provided to client code *and* the metadata used to track the chunks must be stored in your memory array.

Do not be fooled by the type of the array: all data is made up of bytes, and an array of chars is just an array of bytes. The address of any location in the array can be freely converted to a pointer of any type.

The simplest structure to use for your metadata is a linked list. Each chunk will be associated with the client's data (the *payload*) and the metadata used to maintain the list (the *header*). Since the chunks are continuous in memory, it is enough for the header to contain (1) the size of the chunk and (2) whether the chunk is allocated or free. Given the location of one chunk, you can simply add its size to the location to get the next chunk. The first chunk will always start at the beginning of memory.

Note the pointer returned by `malloc()` must point to the payload, not the chunk header.

Note that your memory array, as a global variable, will be implicitly initialized to all zeros. Your `malloc()` and `free()` functions must be able to detect when they are running with uninitialized memory and set up the necessary data structures. You are *not* permitted to require clients to call an initialize function before calling `malloc()`.

2 Detectable errors

The standard `malloc()` and `free()` do no error detection, beyond returning NULL if `malloc()` cannot find a large enough free chunk to fulfil the request.

In addition, your library must detect and report these usage errors:

1. Calling `free()` with an address not obtained from `malloc()`. For example,

   ```
   int x;
   free(&x);
   ```

2. Calling `free()` with an address not at the start of a chunk.

   ```
   int *p = malloc(sizeof(int)*2);
   free(p + 1);
   ```

3. Calling `free()` a second time on the same pointer.

   ```
   int *p = malloc(sizeof(int)*100);
   int *q = p;
   free(p);
   free(q);
   ```

You *may* provide a function that can be called before a program exits to determine whether any memory chunks remain allocated (that is, to detect possible memory leaks).

3 Reporting errors

We will use features of the C pre-processor to allow `malloc()` and `free()` to report the source file and line number of the *call* that caused the error. To do this, your "true" functions will take additional parameters: the source file name (a string) and line number (an int).

Use these function prototypes and macros in your `mymalloc.h`:

```
void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);


#define malloc(s) mymalloc(s, __FILE__, __LINE__)
#define free(p) myfree(p, __FILE__, __LINE__)
```

The C pre-processor will replace the pseudo-macros `__FILE__` and `__LINE__` with appropriate string and integer literals, which will give your functions the source locations from which they were called.

Note that we are stealing the names of functions defined in the standard library. For this reason, make sure that `mymalloc.h` is included later than `stdlib.h`. For example,

```
#include <stdlib.h>
#include "mymalloc.h"
```

The content and format of your error messages is up to you. Describe your design in your project's README file.

4 Correctness Testing

You will need to determine that your design and code correctly implement the `malloc()` and `free()` functions. (Note: this determination is part of your coding process, and is distinct from detecting run-time errors in client code, as described in section 3.)

In addition to inspecting your code for bugs and logic errors, you will want to create one or more programs to test your library. A good way to organize your testing strategy is to (1) specify the requirements your library must satisfy, (2) describe how you could determine whether the requirements have been violated, and (3) write programs to check those conditions.

For example:

1. `malloc()` reserves unallocated memory.
2. When successful, `malloc()` returns a pointer to an object that does not overlap with any other allocated object.
3. Write a program that allocates several large objects. Once allocation is complete, it fills each object with a distinct byte pattern (e.g., the first object is filled with 1, the second with 2, etc.). Finally, it checks whether each object still contains the written pattern. (That is, writing to one object did not overwrite any other.)

Other properties you should test include:

- `free()` deallocates memory
- `malloc()` and `free()` arrange so that adjacent free blocks are coalesced
- The errors described in section 3 are reported

5 Performance Testing

Include a file `memgrind.c` that includes `mymalloc.h`. The program should perform the following tasks:

1. `malloc()` and immediately `free()` a 1-byte chunk, 120 times.
2. Use `malloc()` to get 120 1-byte chunks, storing the pointers in an array, then use `free()` to deallocate the chunks.
3. Randomly choose between
   - Allocating a 1-byte chunk and storing the pointer in an array
   - Deallocating one of the chunks in the array (if any)

   Repeat until you have called `malloc()` 120 times, then free all remaining allocated chunks.
4. Two more stress tests of your design. Document these in your README.

Your program should run each task 50 times, recording the amount of time needed for each task, and then reporting the average time needed for each task. You may use `gettimeofday()` or similar functions to obtain timing information.

6 Alignment

For the purposes of this assignment, you are not required to consider the alignment of the objects allocated by `malloc()`. Under the most strict reading of the C language standard, this limits your test code to using `char *` and `void *` pointers. In practice, we are using Intel hardware that allows for unaligned memory accesses, meaning that the use of other data types will work correctly.

In short, if you have not already completed 211 and are unfamiliar with alignment, you do not need to consider it for this assignment.

Challenge mode You may, as a personal challenge, write your code to respect object alignment. Assume that the largest possible alignment is 8 bytes. To guarantee 8-byte alignment, you will need to ensure that all objects begin at an address divisible by 8.

First, define your memory array using `double` instead of `char`, to ensure that the first byte of memory is divisible by 8:

```
static double memory[512];
```

Note that the length of the array has been divided by 8.

Second, ensure that your headers use a multiple of 8 bytes and that your allocations are rounded up to the smallest multiple of 8 at least as large as the requested space. (Implication: a 1-byte allocation will reserve at least 16 bytes in your array.)

Finally, add a requirement to your test plan that all pointers returned by `malloc()` are divisible by 8. (Note that `x` is divisible by 8 if and only if `x & 7 == 0`.)

7 Grading

Grading will be based on

- Correctness: whether your library operates as intended
- Design: the clarity and robustness of your code, including modularity, error checking, and documentation
- The thoroughness and quality of your test plan