CS 214 Spring 2024
Project I: My little `malloc()`

David Menendez

Due: February 20, 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 README file containing the name and netID of both partners, your test plan, descriptions of your test programs (including any arguments they may take), and any design notes you think are worth pointing out. This should be a plain text file.

Submit all files to Canvas in a single Tar archive. Place all files in a directory called "P1".

We should be able to run the following commands as written (aside from the name of the archive):

```
$ tar -xf p1.tar
$ cd P1
$ make
$ ./memgrind
```

1 Background

The *heap* is a region of memory managed by the run-time system through two functions, `malloc()` and `free()`, which allocate and deallocate portions of the heap for use by client code.

We will model the heap as a sequence of variably sized *chunks*, where a chunk is a contiguous sequence of bytes in the heap and all bytes in the heap belong to exactly one chunk. Each chunk has a size, which is the number of bytes it contains, and may be either allocated (in-use) or deallocated (free).

The number and sizes of the chunks is expected to vary over the run-time of a program. Large chunks can be divided into smaller chunks, and small chunks can coalesce into larger chunks.

We can further model a chunk as having two parts. The *header* contains information the run-time system needs to know about a chunk, such as its size and whether it is allocated. The *payload* contains the actual object that will be used by client code. We say that the payload contains data, and the header contains metadata.

Note that an object is itself a contiguous sequence of bytes, meaning that the run-time system cannot intermix data and metadata.

To ensure smooth operation, the run-time system and the client code must operate by certain rules:

1. On success, `malloc()` returns a pointer to the payload of an allocated chunk containing at least the requested number of bytes. The payload does not overlap any other allocated chunks.

2. The run-time system makes no assumptions about the data written to the payload of a chunk. In particular, it cannot assume that certain special values are not written to the payload. In other words, the run-time cannot extract any information by looking at the payload of an allocated chunk. Conversely, clients may write to any byte received from `malloc()` without causing problems for the run-time system.

3. The run-time system never writes to the payload of an allocated chunk. Client code may assume that data it writes to an object will remain, unchanged, until the object is explicitly freed.

4. The run-time system never moves or resizes an allocated chunk. [1]

5. The client never reads or writes outside the boundaries of the allocated payloads it receives. The run-time system can assume that any data it writes to chunk headers or to the payloads of unallocated chunks will be not be read or updated by client code.

`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 Coalescing free chunks

Consider a case where we repeatedly call `malloc()` and request, say, 24-byte chunks until all of memory has been used. We then free all these chunks and request a 48-byte chunk. To fulfil this request, we must *coalesce* two adjacent 24-byte chunks into a single chunk that will have at least 48 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] Functions like `realloc()` that explicitly move or resize chunks are permitted, but `malloc()` and `free()` by themselves must never change existing allocated chunks.

1.2 Alignment

C requires that pointers to data are properly *aligned*: pointers to 2-byte data must be divisible by 2, pointers to 4-byte data must be divisible by 4, and so forth. We will assume that the largest data type our programs will use has 8-byte alignment. To ensure that any object allocated by `malloc()` has 8-byte alignment, we will ensure that all offsets from the start of our heap are multiples of 8.

This means that allocations must also be multiples of 8: specifically, the smallest multiple of 8 greater than equal to the requested amount. Thus, a call to `malloc()` requesting 20 bytes must actually allocate 24 bytes.[2]

Note that each chunk must have a length that is a multiple of 8, and its header and payload must also have lengths that are multiples of 8. This means that the smallest possible chunk is 16 bytes.

1.3 Your implementation

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

```
#define MEMLENGTH 512
static double memory[MEMLENGTH];
```

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.

Note the use of `double`. This is purely to ensure that the array has 8-byte alignment and does not indicate that data should be stored as floating-point numbers. To perform byte-width pointer arithmetic, cast `memory` to a `char` pointer, e.g.,

```
char *heapstart = (char *) memory;
char *byte200 = heapstart + 200;
```

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 doubles 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.

[2]You will need a way to round up to the nearest multiple of 8. This can be done with a single addition followed by a bitwise and.

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).

Note that some errors, such as use after free, cannot be detected, as they do not involve `malloc()` or `free()` directly.

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).

Your `mymalloc.h` must contain these function prototypes and macros, exactly as defined, and no others:

```
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, but should include the name of the function and the file name and line number of the code that called the function. For example,

```
free: double free (test.c:24)
```

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 detected and 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 object, 120 times.
2. Use `malloc()` to get 120 1-byte objects, storing the pointers in an array, then use `free()` to deallocate the chunks.
3. Create an array of 120 pointers. Repeatedly make a random choice between allocating a 1-byte object and adding the pointer to the array and deallocating a previously allocated object (if any), until you have allocated 120 times. Deallocate any remaining objects.
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.

Note Depending on your header size, a chunk storing a 1-byte object will be at least 16 bytes long. If your header is such that the minimum chunk size exceeds 32 bytes, it will not be possible to allocate 120 objects. You may reduce the number of simultaneous allocations in such a case, but it would be better to reduce your payload size.

6 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