CS 214 Fall 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 file named AUTHOR containing the netID of both partners.
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), 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". You can create your archive using the `tar` command, like so:

```
$ tar -vzcf p1.tar P1
```

This command will create a compressed Tar archive named "p1.tar" containing all files in P1, including subdirectories. (Please clear out temporary files and executables before creating the archive.)

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] Functions like `realloc()` that explicitly move or resize chunks are permitted, but `malloc()` and `free()` by themselves must never change existing allocated chunks.

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.

1.2 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 4096
static union {
    char bytes[MEMLENGTH];
    double not_used;
} heap;
```

The union with a double ensures that `heap.bytes` begins at an address with 8-byte alignment. 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 other than the initialization flag discussed in section 1.4. 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 intimidated by the type of `heap`: the union is just there as a portable way to enforce alignment. You will never use `heap.not_used` for any purpose.

Do not be fooled by the type of `heap.bytes`: all data is made up of bytes, and an array of chars is just a convenient way to specify that. 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 conceptually a linked list. Each chunk will be associated with the client's data (the *payload*) and the metadata used to maintain the list (the

[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.

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. Thus, there is no need for the header to contain an explicit pointer.[3]

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

1.4 Initialization

Unlike the run-time system, your code will be unable to initialize your heap before `main()` is called. Instead, your implementations of `malloc()` and `free()` must check whether the heap has been initialized and call your initialization function if it has not.

To support this, you are permitted to create single static int variable that indicates whether the heap has been initialized.

Your library MUST NOT require clients to explicitly call an initialization function. Client code must continue to work as-is if the directive to include `mymalloc.h` is removed.

In addition to initializing the heap, your initialization function should register a leak detection function to run at exit, using `atexit()`.

2 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"
```

[3]Again, do not be fooled by names. A linked list can be any sequence where each element indicates how to find the next one. Structs with pointers are simply one way to implement linked lists.

2.1 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.

Your `malloc()` must also return NULL if it is unable to fulfil the request, but must also print a message to standard error with this format:

```
malloc: Unable to allocate 1234 bytes (source.c:1000)
```

replacing the number of bytes, the file name, and the source line appropriately.

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. For example,

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

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

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

In all three cases, your library should print an error message with this format:

```
free: Inappropriate pointer (source.c:1000)
```

At your discretion, you may make the error message more specific or provide additional information, such as the specific bad pointer value.

After printing the error message, `free()` should terminate the process with exit status 2, using the call `exit(2)`.

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

2.2 Leak detection

Your library must include a function that checks whether any chunks in the heap are allocated. If it detects any, it should print a message to standard error indicating the number of allocated chunks and their total size in bytes. For example,

```
mymalloc: 128 bytes leaked in 14 objects.
```

Use `atexit()` during initialization (see section 1.4) to register the leak detector to run during process termination.

Note: your leak detector MUST NOT call `exit()` itself.

3 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 2.)

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 2 are detected and reported
- Leaked objects are detected and reported

4 Performance Testing

Separate from the correctness testing described in section 3, you will want to determine the efficiency of your implementation. To do this, your program `memgrind` will perform a *workload* 50 times and report the average time required.

Include a file `memgrind.c` that includes `mymalloc.h`. This program will use `gettimeofday()` to get the starting time, execute the workload 50 times, get the ending time, and then report the average time for the workload (computed by dividing the elapsed time by the number of runs).

The workload consists of five tasks, of which the first three are:

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.

The remaining two are for you to design.

Note Your workload is not intended to check for programming errors in your library, nor should it include deliberate run-time errors. However, the amount of operations performed by `memgrind` may expose flaws in your implementation. Each task in the workload should ensure that all allocations are freed, as you may run out of memory otherwise.

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.

5 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