My-Little-Malloc

This project implements a simulation of the malloc and free functions included in `<stdlib.h>`. These implementations of malloc and free can be used as replacements to the actual malloc and free defined in `<stdlib.h>`. Please refer to the PDF for more details on the directions of this assignment.

PLEASE NOTE: The professor said that using MarkDown was an acceptable format. I had directly confirmed with him.

FURTHERMORE: To have the best reading experience, we HIGHLY recommend you to use a MarkDown editor as that will provide the best reading experience.

Pre-Requisites

This library (and it's corresponding programs) were only intended to be executed on `x86` and `x86_64` architectures. This library and programs assume that the hardware will be consistent with that of `x86` and `x86_64` architecture. This requires the hardware to use 2s complement and be consistent with Little Endian endianness.

Project Structure

- There are two directories list: `src` and `tests`
  - `src` includes all `memgrind.c` and `mymalloc`'s necessary files. This is where our implementation of `malloc` lies. `memgrind.c` includes all the tests that we were instructed to write in the `pa1.pdf` write-up.
    - To run `memgrind`, execute the following in the `src` directory:
      - ```
        make
        ./memgrind
        ```
  - `tests` has all the additional custom tests we wrote that are including our implementation of `malloc`.
    - Inside the `tests` directory, there are five separate `test#` directories. Navigate to any of the directories and you'll find a `Makefile` that will compile execute the specified `test#.c` file.
    - To run any of these test, navigate to the desired directory and run the following commands.
      - ```
        make
        ./test#
        ```
    - **PLEASE NOTE**: Replace `#` with the number with the test number you wish to run.

Test Plan

Our test plan includes a series of stress tests that will trigger almost all the error checking we did in our program. `errors.h` contains function names for the errors we are checking for. They are listed below.

Errors

```
void doubleFree(char* file, int line);
void wrongPointer(char* file, int line);
void tooMuchMem(int MEMSIZE, char* file, int line);
void noMoreMem(char* file, int line);
void mallocZeroError(char *file, int line);
void nullPointerPassed(char *file, int line);
```

- `void doubleFree(char* file, int line);`
  - This error is called in the event that the user is trying to free a pointer they have already freed or if they are trying to free something that may not have been allocated yet and is marked as `AVAILABLE` in the metaData, this error will be triggered with an error message on the screen.
- `void wrongPointer(char* file, int line);`
  - This error is called in the event that the user passes any pointer other than the original one was that given to malloc. In this event, this error will be called and free will fail.
- `void tooMuchMem(int MEMSIZE, char* file, int line);`
  - This error will be called in the event the user tries to allocate too much memory. Say that `MEMSIZE` is 4096. If the user tries to allocate more than 4080 bytes (because malloc needs to store two instances of metaData), then this error will be called indicating that too much memory was requested. `NULL` will be returned, along with an error message.
- `void noMoreMem(char* file, int line);`
  - Contrary to `tooMuchMem`, this function will be called in the event that malloc cannot find a big enough block of space to request the data requested by the user. In this case, this error message will be printed out and `NULL` will be returned.
- `void mallocZeroError(char *file, int line);`
  - Contrary to the actual `malloc` defined in `<stdlib.h>`, in our version of malloc, the user cannot allocate zero bytes (for safety reasons). This dramatically improves the safety of the client's program by preventing them from allocating zero bytes. In such an event, `NULL` will be returned.
- `void nullPointerPassed(char *file, int line);`
  - This means that a `NULL` pointer was passed into the `free` function. This is illegal and the program will crash in this case with a return status of `EXIT_FAILURE`.

Properties Our Library Must Have To Be Correct

In order for the library to be correct, we concluded that the library must have the ability to `malloc`, `free`, and coalesce blocks. The library must also have the ability to call out errors were detailed in the errors section here

How to Check That Our Code Has These Properties

In order to check that our code has these properties, we intend to use various forms of testing. In the `memgrind.c` file, we intend to place the three stress tests from the write-up that was given and add two more stress tests of our own. In addition, located in the separate `tests` directory, we have added five more custom tests that use `malloc`/`free` normally and some tests that intentionally cause errors. Later on, we intend to use the library we implemented on these test cases to check whether our code has these properties. Furthermore, as we test, we are using a function that prints out our memory array so, we can manually check whether our library is correct.

- `void printMemory(int bytes);`
    - This function will print `bytes` number of `bytes` from the memory array. Each address in the array will be printed, along with the value that is stored in that specific element of the memory array. Please note, since this program is only intended to be ran on `x86` or `x86_64` architecture, the bytes will be printed using 2s compliment using Little Endian endianness.

Specific Methods to Check Each Property

In order to test whether an individual property is working we have specific methods to check each one.

- `malloc` and `free`: There are specific test cases in `memgrind.c` and specific test cases in the `tests` directory that check whether `malloc` and `free` are working properly. In `memgrind.c,` all the tests `malloc` data in different ways and `free` it at some point during the test. We have confirmed to make sure all these test cases behave as expected.

- Coalesce: Technically coalesce should be working for the test cases mentioned previously, however in order to check it properly we made a specific test case in `memgrind` that randomly allocates memory and randomly frees it. This will allow us to check whether our free blocks are combining correctly and allowing malloc to happen even after we free multiple times.

- Error-Checking: In the `tests` directory, there are multiple tests that intentionally use malloc and free improperly in order to check whether our errors in the library are called when required.

Test Programs

Memgrind.c Test Cases

In this program, we run each stress tests 50 times.

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:

    1. Allocating a 1-byte chunk and storing the pointer in an array
    2. 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. Use `malloc()` to allocate enough memory for an integer array of 120 size. Fill the array with elements, which will be 1 to n. Finally `free()` the dynamic array.

5. Similar to test case 1 and test case 2, in this test, we are creating a 2D array of size 10 * 12. Then, we fill the array up with integers between 0 and `RAND_MAX`. We are using `malloc` and `free` to create and deallocate this array respectively.

PLEASE NOTE: ALL THESE TEST CASES ARE REPEATED 50 TIMES AS SPECIFIED BY THE DIRECTIONS. THE AVERAGE TIME IT TOOK FOR EACH TEST TO RUN IS PRINTED WHEN `memgrind.c` IS EXECUTED USING THE MAKEFILE PROVIDED IN `src`.

`tests` Directory Test Cases

Please Note: Inside the `tests` directory, there is a directory for each test case. Inside, there is a `Makefile` which will execute each test.

1. Test1 will get the number of elements from `stdinput` and print out the elements of the array, which will be 1 to number of elements in array.
   - Expected arguments:

     > Integer from the command line that determines how large the array will be

   - Expected output:

     > Enter number of items:#
     > Number of elements:#
     > Array: {1,...,#}
     > e.g. if number of items entered is 4 the output would be
     >
     > Enter number of items:4
     > Number of elements: 4
     > Array: {1, 2, 3, 4}

2. Test2 will get size of identity matrix from user, make identity matrix, and print it out
   - Expected arguments:

     > Integer from the command line that determines how large the identity matrix will be

   - Expected output:

     > Enter size of matrix: #
     > Enter size of matrix: #
     > 1 0 ... 0
     > . .
     > . .
     > . .
     > 0 0 ... 1
     >
     > e.g. if user inputs 5 as the size of identity matrix output will look like
     > Enter size of matrix: 5
     > 1 0 0 0 0
     > 0 1 0 0 0

> 0 0 1 0 0
> 0 0 0 1 0
> 0 0 0 0 1

3. Similar to test case 1, however, instead of freeing correctly we are freeing twice. We are expecting this to give the `doubleFree` error in our library.
   - Expected arguments:
     > Integer from the command line that determines how large the array will be
   - Expected output:
     > `doubleFree` error
4. Similar to test case 1, however, instead of freeing correctly we are freeing the wrong pointer because we added 1 to the pointer. We are expecting to get a `wrongPointer` error.
   - Expected arguments:
     > Integer from the command line that determines how large the array will be
   - Expected output:
     > `wrongPointer` error
5. Similar to test case 1, we "accidentally" multiply by zero in our malloc call. This triggers a the `mallocZeroError`.
   - Expected arguments:
     > Integer from the command line that determines how large the array will be
   - Expected output:
     > `mallocZero` error

Design Properties

Our library includes several unique and interesting design properties. All design properties for `memgrind.c` and the `tests` directory were referenced above in [Test Programs](). This section will strictly focus on `mymalloc.c` and how we actually implemented our versions of `malloc` and `free`.

`mymalloc` includes several functions that will allow the user to call `malloc()` and `free()` seamlessly, just as if they had used the `<stdlib.h>` definition of `malloc` and `free`.

Our library also has a cool feature where it can `freeAll` the entire array. This will prevent memory leaks and will provide a hard reset on the entire memory array in case something goes wrong. C does not have such a feature built into any of its standard libraries.

Documentation and Design Properties of `MyMalloc.h`

`mymalloc.h` provides function prototypes for these functions that are later defined in `mymalloc.c` These prototypes can be seen below:

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

`mymalloc.c` also defines a function `void *initializeMemory(size_t size)` and `void coalesceBlocks()` which will be elaborated on later.

- `void *initializeMemory(size_t size)`
    - This function will initialize the publicly defined `memory` array with two sets of metaData. One for information about the requested size from `mymalloc` and another that will contain information about the remaining memory in the array (this excludes space for `metaData`, consistent with the methodology in `mymalloc`). That is, the first metaData will contain information about the requested size allocation from `mymalloc`. Then, to the right of the allocated data, a new, `available` `metaData` is stored. This will contain information about how much memory is left over. This metaData will ensure space that it subtracts the space `metaData` will take on the next call to `malloc`. This means, it stores the amount of memory that can actually be allocated by the user. It subtracts space for all the `metaData` overhead. This means, the information stored to the right of the metaData will be the amount of memory that is actually left over to be allocated!
- `void *mymalloc(size_t size, char *file, int line);`
    - This function will replace all calls in the client code to `malloc` assuming they have `mymalloc.h` included in their files.
    - This function will first check to if the requested memory is greater than `MEMSIZE`. By default, `MEMSIZE` is 4096.
        - Please note: The requested size must be less than or equal to `MEMSIZE - 16`. This is because of `mymalloc` to function correctly it requires that there be enough space to store one `metaData` that will store information about the requested data, and there be enough space to store a second `metaData` AFTER the allocated space to store information about how much space is left over. This `metaData` is set to be available.
        - Also, there may be a case where the user deallocates all their memory (and then by default, all the blocks will be coalesced). In this case, the metaData that will be left over will contain information about how much space is left, accounting for one more `metaData` allocation and the current `metaData` allocation. This means, that in this case, where all the blocks have been deallocated and coalesced, the memory that will be available to the user will still be `MEMSIZE - 16`.
    - Contrary to the `malloc` defined in `<stdlib.h>`, this version of `malloc` will not allow the user to `malloc` 0 bytes. In this case, `malloc` will fail and return `mallocZeroError`. This will provide more safety and security compared to actual `malloc` defined in `<stdlib>`.
    - If the memory has not been initialized, `mymalloc` will initialize it. Please see the `initializeMemory` section for more information.

    - Then, we iterate through the memory array. If we find a block with the right size, we allocate that block. We then check to see if there `metaData` stored to the right of the block we just allocated. If there is no metaData there, we check to see if we are in the bounds of the `memory` array, and if we are, then create a new metaData to the right of the block we previously allocated. This means we are at the right most part of the array where there is no `metaData` yet allocated. We allocate this new `metaData` setting it to `available` by default. We also ensure that we store how much memory is left to be called by the client code. This ensures that we have enough space for the next metaData to be stored, hence why we subtract the sizes of the `metaData` storing from the amount of memory left available to allocate. THIS MEANS that the amount data size left is stored in this `metaData`. In other words, the amount that is stored in this `metaData` is how much the user has left to allocate, accounting for the `metaData` overhead.
        - If there is not enough space for the second metaData stored to the right of the allocated block, then we return `noMoreMem` and return `NULL`.
    - Assuming the above executed without running into any of the described edge cases, we return the address of the allocated chunk back to the client.
        - Note: If we were not able to find an available block in the entire memory array while iterating through each `metaData`, then we return `noMoreMem` and return `NULL`

- `void myfree(void *ptr, char *file, int line);`

    - This function will take a pointer to the BEGINNING of an allocated chunk of memory that was returned by `mymalloc`.
        - If `*ptr` is `NULL`, then we print a `nullPointerPassed` error and `EXIT_FAILURE`
        - If the above does not happen, then we take the pointer given and do pointer arithmetic to get the metaData that *should* be stored right behind the pointer that the client passed.
            - If the `metaData` stored at this address is set to `available`, then this means that the user is either trying to double free this pointer OR the user is trying to free a block that has set to `available` already.
        - If the above edge cases fail, then this means the pointer that was passed is valid. In this case, we set that blocks `metaData` to `available` and then we call `coalesceBlocks()`.

- `void coalesceBlocks()`

    - This function loops through the entire memory array looking for two adjacent blocks and combines them into one block (while maintaining the metaData overhead property explained in the `mymalloc` function documentation).
    - We store the first `metaData` in the array, and we also find the location of the next metaData in the memory array.
    - If the next `metaData` is within the bounds of the memory array, we can continue. If not, we return because we have reached the end of the array.
    - If the first `metaData` that we stored, and the second `metaData` that we stored are both set to be `available`, then we set the first `metaData` to have enough space for the block that it previously allocated, the size of the second `metaData`'s allocation, and the size of that `metaData`.
        - This means that if the first `metaData` had a four byte allocation, and the second `metaData` also had a 4 byte allocation, then, the coalesced `metaData` (the first one),

will store the following size that is allowed to be allocated once again my `malloc`: `firstMetaDataAllocationSize + secondMetaDataAllocationSize + sizeof(secondMetaDataSize)`.
- Please see the `coalesceBlocks()` function in `mymalloc.c` for more information.
- If we were not able to find two adjacent free blocks, we iterate over to the next metaData.

- `void freeAll();`
    - This function loop through the entire array, looking for any allocated blocks and calling `free` on those blocks manually.
    - After looping through the entire `memory` array, it will call one more `coalesceBlocks` to make sure that everything is entire deallocated and coalesced.

- `void printMemory(int bytes);`
    - This function was defined and explained HERE. Please refer there.

What We Are Able to Prove

We are able to prove all these design properties because of our `printMemory` array and our wide range of test cases and stress tests. Because we are testing and verifying each test case and stress test with our `printMemory` function, we have been able to view the entire memory array and verify that each of the Design Properties are behaving as expected and this can be proved with our `printMemory function.`

Furthermore, we are also able to prove that after the first call to malloc, there will always be a `metaData` in the array indicating how much memory there is left to call (with it taking into account that there will be `metaData` overhead. If you are not sure what the `metaData` overhead is, please read the `mymalloc` function documentation in Design Properties). Thus, at any one time, there will always be a metaData in the array that indicates how much memory there is left to allocate.

In addition to this, based on our timings in `memgrind`, we are able to conclude that the method we used is similar to that used in `<stdlib.h>` in terms of performance. Both, our `malloc` implementation and the `malloc` implementation defined in `<stdlib.h>` have similar average times to run each test: both of them taking about several hundred microseconds, on average to complete.

Design Notes

There are several design notes we believe are worth pointing out. Additionally, we took some extra measures to improve the functionality of our `malloc` compared to the one defined in `<stdlib.h>`. Additionally, here is where we explain what `gcc` flags we are using in the compilation process and why we are using them. Here is where we will just explain some of the decisions we made and why we happened to make them.

- First off, in our `mymalloc()` implementation, we are always storing two instances of `metaData` in the memory array.
    - The first `metaData` marks the block allocated next to it as being `used`. This first `metaData` stores information about the size of the block that the client has allocated.

  - Then, after the first block is allocated, we add a second `metaData` (assuming there is not one there already. If there is, we don't do anything. (Please read the `mymalloc` function documentation in Design Properties for more information.))
  - This second `metaData` is marked as `free` and ready to be allocated. This `metaData` stores how much more memory can actually be allocated by the user. In other words, this is the left-over limit that malloc is allowed to call and nothing else. This means that the leftover data that is allowed to be allocated accounts for the current metaData that it is stored in, and the creation of a new metaData.
    - In our implementation, there **ALWAYS** must be a `metaData` stored to the right of any allocated block. Even when the entire memory has been freed (and is currently initialized), there will still always be a metaData left telling the user how much more memory they are allowed to allocate.
  - Thus, when we are allocating the second `metaData` to store how much data is left over to be allocated, we are storing how much memory is left, minus the size of the current `metaData`, minus the size of the next `metaData` that would be allocated in the event another `malloc` call is made. Thus, this is why the `metaData` containing the information about how much memory is left may be less than `MEMSIZE - (what's already allocated)`. It is because there is a `metaData` overhead that we need to consider for first. Thus, anywhere in this document where we mention `metaData` overhead, this is what we mean. We also explain this in our documentation of `mymalloc` here.
- Secondly, we included a `freeAll()` function in `mymalloc.h` and in `mymalloc.c`. This function will free all allocated blocks of memory and coalesce them all together. The `malloc` function defined in `<stdlib.h>` does not include such a function. This includes a "HARD RESET" button on all allocated blocks of memory in case something goes wrong in the client's code. This is meant to be a debugging tool for the client, but it can also be a "lazy way" to deallocate memory if needed.
  - Additionally, this function provides more safety to the client code by having a way to essentially avoid any and all memory leaks they may cause (as long as they are not messing up the `metaData`). This is a key advantage over the `<stdlib.h>`'s implementation of memory management.
- Thirdly, we give the client's code access read-only access to `x` bytes of the array up to `MEMSIZE` in case they want low-level access to their allocations. This is meant for debugging purposes, but `<stdlib.h>`'s implementation of `malloc` does not account for this or provide such access.
- Fourthly, alignment. Because we wanted to not let any memory go to waste, and we wanted to provide the most memory possible (since we were told that this code would only ever run on Intel `x86` or Intel `x86_64` architectures), we decided not to include any type of alignment to allow for the most amount of memory to be used. Thus, this means that this code is only able to work on any kind of `x86` or `x86_64` architecture. However, based on our code's modularity, this functionality would be easy to add for any kind of data alignment.
  - That is, it is very easy to make sure on our iterations that we are always iterating to `n`-byte aligned addresses to guarantee that our memory is aligned.
  - However, since our use case with this code given the architecture that is being used, we decided to maximize space rather than alignment as that would provide better results in this niche use case.

C Flags That Are Used Throughout This Project

Throughout this project, we use different C Flags when compiling. The custom `tests` that we wrote uses their own C Flags (because some of them) trigger and error on purpose. `memgrind` is compiled using its flags. Here, we explain why we used each one.

- `memgrind` compile flags:
    - `-g`: provides debugging information in case the use for a debugger like `gdb` is needed. It can provide more debugging information.
    - `-std=c99`: This is C standard that we are required to use for this class.
    - `-Wall`: Enable all major warnings.
    - `-Wvla`: Warn if variable length arrays are used. This is to avoid tampering with the `memory` array or any strange behaviors in the client code.
    - `-Werror`: Treat all warnings as errors.
    - `-Wundef`: Warn if any undefined behavior occurs.
    - `-Wpointer-arith`: Warn if any invalid pointer or risky arithmetic occurs.
    - `-O2`: Enable second level optimizations. This will provide a good balance between performance and compile time.
    - `-fsanitize=address`: We include this because this program involves a lot of pointer and address arithmetic. We include this to make sure nothing goes wrong at compile time.
- `tests` compile flags (all the tests in the `tests` directory use the same flags):
    - `-g`: provides debugging information in case the use for a debugger like `gdb` is needed. It can provide more debugging information.
    - `-std=c99`: This is C standard that we are required to use for this class.
    - `-Wall`: Enable all major warnings.
    - `-Wvla`: Warn if variable length arrays are used. This is to avoid tampering with the `memory` array or any strange behaviors in the `tests` should any occur.
    - `-Werror`: Treat all warnings as errors.
    - `-Wundef`: Warn if any undefined behavior occurs.
    - `-Wpointer-arith`: Warn if any invalid pointer or risky arithmetic occurs.
    - `-fsanitize=address,undefined`: We include this because this program involves a lot of pointer and address arithmetic. We include this to make sure nothing goes wrong at compile time. We also include `undefined` sanitizer because we want to ensure that we are not doing anything undefined will occur that will mess with the `memory` array.