Miniproject 2: Malloc and Free

CS 4410: Operating Systems

October 20, 2013

1 Introduction

For the second assignment, you will implement your own version of `malloc` and `free`. By compiling your implementation into a shared library, you can then link in the library, and run whichever C or C++ program you like with your `malloc` implementation, which is actually kind of cool.

We will provide a very basic memory request interface consisting of the function `void get_memory(unsigned num_bytes)`. We have also provided implementations of the related functions `calloc` and `realloc`. Please pay attention to the first line of the provided `realloc` implementation, which must be modified depending on your `malloc` implementation.

There are many different implementations of `malloc`, some available online, and others presented in textbooks; any of these can be used for reference, though implementations such as the GCC `malloc` are probably very confusing.

This assignment will need to be done on a Linux machine, since the `get_memory` function uses a system call called `sbrk` not provided on Windows. `sbrk` is provided on OSX, though the interface we give you has not been tested on any platform other than Linux, so YMMV.

2 Description

2.1 First Attempt

The interfaces you will complete are shown below. `malloc` allocates a set amount of memory, and `free` marks that block of memory available for re-use. Since we have provided a function to request new memory from the operating system, a sample “solution” could look like this:

```
void* malloc(size_t size) {
    return get_memory(size);
}
void free(void* ptr) {
    return;
}
```

Of course, this is hideously inefficient, and means that no memory can be re-used. Furthermore, a system using this implementation will very rapidly run out of memory, because `get_memory` will likely return a memory segment that is an integer number of pages, meaning an entire page is allocated for something like malloc(sizeof(char))[1]. There are significantly better ways of doing things.

2.2 A Better Way of Doing Things

Since you will want to keep track of what pointers have been allocated by your system, you will need to store information about each chunk of memory that has been allocated. A very common way of doing this is to define a header in front of each chunk of allocated memory with important information. A sketch of such an implementation would look something like this:

```
struct header {
    unsigned size;
};

void* malloc(size_t size) {
    struct header* ptr =
        (struct header*) get_memory(size + sizeof(struct header));
    ptr->size = size;
    return (void*)(ptr + 1);
}

void free(void* ptr) {
    // Do whatever you need to do.
}
```

This isn't a real implementation (it has all of the problems of the first solution), but it shows that you can set aside a section of memory *before* the pointer you return and store information there. This also illustrates the basics of pointer arithmetic: instead of returning `ptr`, `ptr + 1` is returned. Since ptr is a `struct header*`, adding one returns a pointer that is `sizeof(struct header)` bytes past ptr. You should become familiar with pointer arithmetic if you are not already.

2.3 Using Headers for a Linked List

A possible approach for this assignment is to use a circular linked list of available memory. `malloc` will search through the list until it finds a chunk that contains enough memory to satisfy the requested amount. If the chunk is exactly the right size, then the chunk is removed from the linked list and returned. Otherwise, the node is split into two nodes, one of which is removed and returned, the other one of which is kept in the linked list. An example chunk is shown below in figure 1.

[1] This depends on how a routine called `sbrk` is implemented on your system, but most implementations will act this way. You can determine the page size with the call `sysconf(SC PAGESIZE)`.

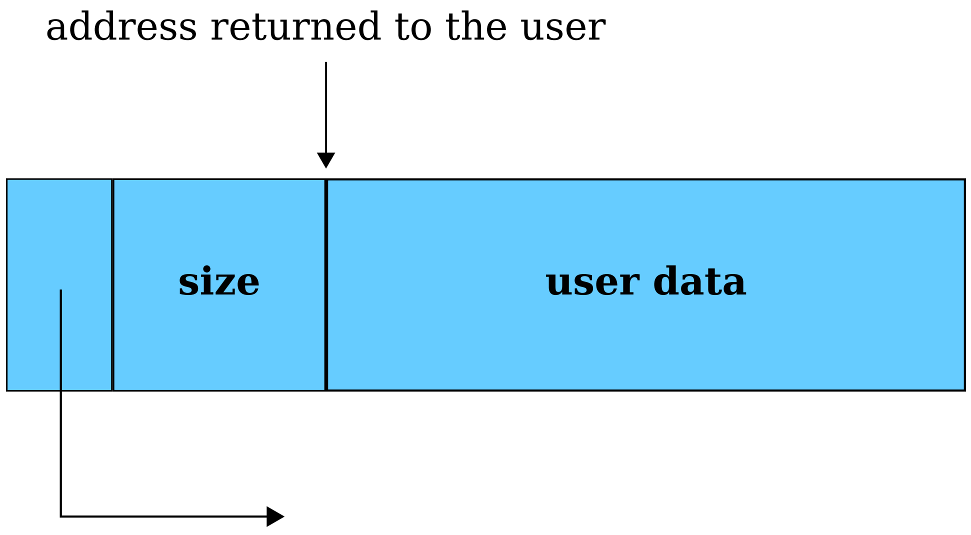

Figure 1: An example chunk in a linked list structure

When a chunk is freed, simple arithmetic can get the pointer to the header that is behind the pointer passed into `free`. Using this, you can insert this chunk into the linked list to be re-used.

```
void free(void* ptr) {
    struct header* header = ((struct header*) ptr) - 1;
    // Do whatever you need to do.
}
```

The implementation described above is very rudimentary. You will want to think about its many flaws, and see what you can do to mitigate them.

2.4 Things to Think About

1. How are you going to keep everything aligned in memory properly?
2. If you use the stupid description above, you will get seriously massive fragmentation; that is, in your circular linked list, you will have a lot of small chunks, and no large ones. As a result, every time you use `malloc` to request a very large chunk, you will have to request more memory from the system, even though you might have enough contiguous free memory in your linked list structure.
3. How will you keep your `malloc` and `free` efficient? Long searches through linked lists could be problematic. Is there a way to optimize so that long searches are less likely?
4. How much space are you wasting? If your header is 24 bytes of memory, and the user requests a chunk the size of a `char`, you are using a lot of your memory on internal storage. Try to keep your header as small as possible.

2.5 Reporting Errors

At some point, you may not be able to acquire more memory from the system when you need it. In this case, your `malloc` implementation should return `NULL`. Additionally, you must set a global variable called `errno` (declared in `errno.h`), which reports the reason for the failure. In this case, you would want to set it to `ENOMEM`.

3 How to Run Your Code

We have provided a makefile that will compile your code into a shared library. By setting the environment variable `LD_PRELOAD` to the path to the shared library, any program that uses `malloc` will instead use the `malloc` you've written. When testing your code, the best way to do this is to run `env LD_PRELOAD="/path/to/your/libmalloc.so" executable_name arguments`. This will run the executable with the arguments in a modified environment with `LD_PRELOAD` set. For more information, look at the `env` manpage (run `man env` in a terminal).

Alternatively, you can use `export` to run *every* program with your `malloc` implementation, as shown below:

```
$ export LD_PRELOAD="/path/to/your/libmalloc.so"
$ gcc input.c
$ ./a.out
```

If you want to stop using your `malloc` implementation and switch back to the standard library, then either close your terminal and open a new one, or run `export LD_PRELOAD=""`.

A very important thing to note is that `free(NULL)` is defined to do nothing at all! If you don't have an explicit check for this in your code, you will most likely get segfaults when trying to run more complicated, already-written programs, which will make you want to drop CS and take up bunny-raising. The author speaks from personal experience.

4 Grading

The most important criterion is correctness. Your `malloc` implementation should be able to allocate any amount of memory (limited by the system, not by any internal constraint). Freed memory should be re-usable, and there should be some scheme to correct for fragmentation. Bonus points will be awarded for more efficient implementations. For ideas and extra reading, look at [1] and [2].

Please attach a readme file describing your implementation, any design decisions, etc.

References

[1] Emery D. Berger and Kathryn S. McKinley and Robert D. Blumofe and Paul R. Wilson, *Hoard: A Scalable Memory Allocator for Multithreaded Applications*, ASPLOS, 2000, pg. 117-128, `http://doi.acm.org/10.1145/356989.357000`.

[2] Jeff Bonwick, *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, USENIX Summer, 1994, pg. 87-98, DBLP, `http://dblp.uni-trier.de`.