Programming Assignment #4

CS 6353 Section 1, Spring 2016

In this programming assignment, you will build a simple memory allocator for C programs, in C (NOT C++). In the next assignment, we will build a simple garbage collector.

As usual, I will grade your program by running it against a large test suite. You will be graded by the percentage of test cases you get correct.

The Due Date for this assignment is May 2nd at 8:00am.

1. Program Specification

In this program, you will not be writing a stand-alone program. You will be writing functions that I will call from my own C code. The functions you will implement are:

```
void   *allocate  (size_t request, size_t *info);
int     deallocate(void *ptr);
size_t  heap_max  ();
```

They are described below.

Your implementation will require you to create your own "heap" internally. For the sake of this assignment, you will create a heap of fixed size, which should be 2^{26} bytes (a mere 64MB – good enough for our purposes). This isn't hard, you can do it in one of two ways:

1. Create a global variable of the desired size (less desirable, but easy).
2. Use the `mmap()` system call to get virtual memory from the OS (use the `MAP_ANONYMOUS` flag). This is a little harder, and since you're not given an initialization routine, you would have to add logic to do this in `allocate()` if there is no heap.

You ARE expected to be able to re-use memory as I use your library to allocate and deallocate items. You must keep a "free list" of memory chunks that you can use to allocate from. Generally, at the start of a program, there will be one large free chunk on the free list. When someone requests memory from it, you'll break it up. As memory gets `deallocate()`'d, you will add it to a list of free chunks that you can also break up to service a request.

Additionally, you must prefer re-using the memory you have previously given to the user that they returned via `deallocate()`, instead of dipping into memory that you haven't

This could be done simply with a linked list of free items where you put newly deallocated chunks at the start of the list, but as you might expect, this won't perform well in many circumstances (for instance, we may have to traverse many small free chunks to get to one big enough to handle our allocation, which can be time consuming.

If you are adventurous, feel free to explore algorithms other than a simple linked list. B-trees are generally good for this kind of use. If you use anything other than a linked list, put a README file to explain your approach.

Here's a description of the semantics for each call:

void *allocate(size_t request, size_t *info);

The allocate function must return a pointer to memory from within your heap. The amount of memory available to the user must be a minimum of request bytes. The address you return to the user must be word-aligned to 64-bit words—meaning that the memory address must be divisible by 8 without a remainder.

Note that you will be servicing a request to `allocate()` by dedicating a little bit more memory than the user actually asked for. There are two reasons for this. First, it can help you with keeping word alignment. Second, you will need to do so to implement one of the requirements for `deallocate()`.

If you cannot return enough memory for the programmer to use, then return the `NULL` pointer, and write `0` into the info field.

Otherwise, write into the info field the ACTUAL length that you have reserved for this memory allocation, counting bytes from the pointer that you give me.

int deallocate(void *ptr);

This function returns a memory object to the free list. Before returning the memory to the free list, you must check the following:

1. If the pointer I give you does not point into your heap, return the value `-1`.
2. If the pointer I give you has explicitly been freed before (and not reallocated), return the value `-2`.
3. If the pointer I give you points into your heap, but is not a pointer that you returned from a call to allocate(), then return `-3`.
4. If the pointer I give you is valid, but in the course of using it, I did an out-of-bounds write, then return `-4`. In this case, you should still return the pointer to the free list.
5. Otherwise, return `0`.

For detecting an out-of-bounds write, I recommend that you place a unique value in memory immediately AFTER the valid end of memory. For instance, if I call `allocate(10, &my_info)`, you could write a special 64-bit value after the 10th byte that you then look at during `deallocate()`—if it's changed, then I overwrote the value!

There are no specific requirements on how to detect an out-of-bounds write. My test cases will do them sometimes, and if you catch them, you will get credit for a test case. If you do not catch them, you will not receive credit.

Similarly for the other error conditions. There are lots of ways you can do this, so I'll be expecting to see diversity in homeworks.

size_t heap_max();

This call should return the largest number of bytes I can request from your `allocate()` without getting an out-of-memory error.

2. Basic implementation approach

For each allocation you return to a user, you will need to reserve additional memory. Typically, you'll at least need to keep track of how big each allocation is. You may need to account for other things, depending on your approach.

Let's consider the size of the allocation, and assume that we're going to keep a linked list of free chunks. We might want to have a data structure to represent this information:

```
struct alloc_hdr {
  unsigned int      chunk_size;
  unsigned int      alloc_size;
  struct alloc_hdr *freelist_next;
};
```

The `chunk_size` field would indicate the total amount of memory we have reserved for this allocation. If the chunk in question is a free chunk, then `freelist_next` will point to the next free chunk's `alloc_hdr` (if there is another free chunk). If the chunk is not free, `alloc_size` would indicate how much memory the user actually requested.

The question is now how we tie this `alloc_hdr` data structure to the memory we give the user. We could keep a hash table that maps pointers to `alloc_hdr` data structures. But an easier and better approach is to stick this data immediately in memory BEFORE the memory you return to the user.

If you do that, when someone calls `deallocate()`, you can substract `sizeof(alloc_hdr)` to get a pointer to the `alloc_hdr` for that chunk of memory (assuming you are careful about alignment issues).

You are NOT expected to build a thread-safe memory allocation function. I will only test it using a single thread.

Also note that you are expected to have a "sane" approach to allocation. Particularly, if my first allocation is the entire heap, and I then free it, you must be willing to break it up to serve future requests. That means you can't have a strategy where, once you create a "chunk", you never change its size.

3. Other Requirements

You will receive a 0 on this if any of these requirements are not met!

1. The homework must be written entirely in C, with no third party libraries.
2. You must submit either a ZIP file containing only: .c files, .h files and a README, OR a single .c file. Generally, I would expect that you not need more than a single .c file for this assignment.
3. Your homework must NOT have a `main()` function. You're basically just delivering me a library.
4. I will compile your code using the command: `cc -c *.c`
5. I will then link test cases against your code, and run them using your memory allocator.
6. Your code must compile, run and work properly in the reference environment. Even if it works on your desktop, if it doesn't work in the reference environment, you will get a 0.
7. You must submit your homework before 8:00am on the due date.
8. You must submit the homework through the course website, unless otherwise pre-approved by the professor.

9. You may not give or receive any help from other people on this assignment.
10. You may NOT use code from any other program, no matter who authored it. This includes 3rd party libraries.
11. I have a detailed library of all publicly available `malloc()` implementations and will be running similarity testing to look for copying. It's best to avoid the temptation of cheating—there's a good chance I'll catch you.

4. Test Cases

No test cases this time—as C/C++ programmers, this should all be fairly clear-cut. Since I'm allowing a lot of flexibility on the internals, people should end up with very different behavior—for example, different students' libraries should run out of memory at different times. And they should run with different performance characteristics.

I'll take all that into account with the grading. I'll also be looking at this information across students to detect collaboration across students. Remember, you are not allowed to work together.

5. Reference Environment

The reference environment is once again kimota.net. Your same login credentials will continue to work.