instructions.md

CS377 Programming Assignment 4

Due April 16th at 11:55PM on Gradescope

Overview

In this assignment you will implement a basic malloc/free allocator. This lab needs to be completed in C using the provided skeleton framework. You must also make sure your code compiles and works under the programming environment we have specified for this course. The overall goal is to expose you to C programming at some of the very lowest levels and very near to the machine and to solidify your understanding of how memory allocators work.

Before you begin you should read or re-read Chapter 17 Free Space Management and keep this open as you complete this lab. This lab follows exactly their description of an allocator and will be crucial in your understanding of allocators as well as how to implement one!

Part 1: Getting Acquainted with the Code

There are five important files that you will need to become familiar with before starting this project:

- **`Makefile`**: This is the `Makefile` that will build your project. It will generate two executable files: `main` and `malloc_tests`. The `main` executable allows you to run your allocator and `malloc_tests` will run public tests locally before submitting your project.
- **`src/my_malloc.h`**: This file is the primary interface to your allocator. It includes the main interface for the two important allocator functions `my_malloc` and `my_free`. It also includes other functions that are primarily used for debugging and testing. In addition, it includes definitions that you will use in creating your allocator.
- **`src/my_malloc.cpp`**: This file will contain the implementation of your allocator. You are welcome to introduce additional functions if necessary in this file, however, you may not change the function signatures of the functions that you are to implement (i.e., those that contain a TODO).
- **`/src/main.cpp`**: This file contains a simple main function that runs your allocator.
- **`test/malloc_tests.cpp`**: This file contains public tests that test your allocator.

Part 2: Implementing the Allocator

There are five functions in the allocator that you must implement: `find_free`, `split`, `my_malloc`, `coalesce`, and `my_free`. We will discuss each of these functions below. Before we discuss the functions that you need to implement let us first introduce you to the important definitions that you will need before starting.

Definitions

- **`header_t`**: This structure is used as the header for allocated blocks of memory returned by the call to my_malloc. As described in the Chapter 17 Free Space Management, we use this structure to store information about an allocated block. In particular, the size of the block of memory allocated to the calling program (not including the number of bytes for this struct) and a magic number we can use to ensure that on a call to my_free we can reasonably be sure that we are given a pointer to a block of memory that was allocated by `my_malloc`. You will notice that we have provided a value for this magic number (`0xDEADBEEF`) that you will use to assign the value of magic.

- **`node_t`**: This structure is used as the header for blocks of memory on our free list. The free list (as described in the book) is a linked list of memory blocks that are available to allocate to the calling program. It contains a field for the size of the available memory (not including the number of bytes for this struct) and a next pointer to the next free node in the free list. You will use this structure for managing the free list.

  Other important definitions include helper functions used to implement `my_malloc` and `my_free` (we expose these for testing purposes), functions for debugging (e.g., `print_free_list`), functions for getting information about your implementation (e.g., `number_of_free_nodes`), and a definition for the maximum size of the heap (e.g., `HEAP_SIZE`).

Details and Support Functions

We have provided definitions that you will not need to implement. In particular, we have defined at the very top of the `my_malloc.cpp` a global pointer to the heap called `head`. The head pointer points to the first node on the free list. You will use this in your implementation of the free list.

- **`heap()`**: We also provide the `heap()` function that is used to allocate a large block of memory of size `HEAP_SIZE` from the operating system using the system call mmap. This function will allocate this large block of memory if one hasn't been allocated previously. It will also initialize that large block to be the first node on the free list. Note how we overlay our `node_t` structure on top of this large block of memory and set the size and next pointer appropriately. This function returns a pointer to the start of the free list.

- **`reset_heap()`**: This function is used to reallocate the heap. You will not need to use this function as it is used by the testing code to free the previous heap to the operating system and reallocate a new one. It may be useful if you define your own tests.

- **`free_list()`**: This function returns a pointer to the start of the free list.

- **`available_memory()`**: This function returns the total amount of free memory available for allocation.

- **`number_of_free_nodes()`**: This function returns the number of nodes on the free list.

- **`print_free_list()`**: This function prints the free list to standard output - helpful to see what your free list looks like.

You should study these functions first to give you some understanding of how we work with the free list. In particular, you should study the implementation of `heap()` to understand how to overlay/embed a C structure on a block of memory (it is as easy as a cast). Next, we talk about the functions you need to implement and some basic hints to get you started.

Functions You Need to Implement

You will implement a basic malloc/free allocator that uses the first-fit allocation scheme. That is, `my_malloc` will attempt to find the first node on the free list that has enough space to satisfy the call to `my_malloc` or return `NULL` to indicate that not enough memory is available. You allocator will only work with a heap of size of `HEAP_SIZE` and not attempt to allocate additional blocks of memory from the operating system. Although this limits the allocator in the amount of memory it can allocate, it is sufficient to give you an idea of the core functionality. Once you complete this lab it will be useful for you to think about how you might extend it to support more memory.

`find_free`

```
find_free(size_t size, node_t **found, node_t **previous)
```

The first function you should implement is the `find_free` function. This function makes use of double pointers to allow it to manipulate what the provided pointers point to so it can return to the caller (by assigning these to these pointers, i.e., `*found` and `*previous`). The goal of the function is given a size of bytes to allocate it will find a node on the free list that can "fit" the requested number of bytes as well as the previous node. We will need a pointer to the previous node in order to perform the "splitting" discussed next.

The basic idea is to iterate over the free list to find a block that will fit the requested size number of bytes. It is important to note that you must consider the size of the block with respect to the size of the `node_t` struct that takes up space and the `header_t` struct that also takes up space. That is, the actual size of the block on the free list is the size field of the `node_t` struct and the `node_t` struct itself which must be enough to accommodate size bytes (provided as a parameter) plus the size of `header_t`.

Once a node on the free list has been found a pointer to it must be returned using found (second parameter) and the previous node (if any) must be returned using previous (third parameter).

`split`

```
void split(size_t size, node_t **previous, node_t **free_block, header_t **allocated)
```

This function has two goals: to split a free node if it has more space than is required by the allocation request and to return a pointer to the allocated block. We use double pointers here as well to allow us to return multiple values. The `size` parameter is the number of bytes we are allocating, the `previous` parameter is the previous node on the free list to the one we are allocating from, the `free_block` parameter is the node of the free list we are going to allocate into, and `allocated` is the block we are allocating and eventually returning back to the caller of `my_malloc`.

The basic approach is this:

1. Adjust the free block pointer by the number of bytes we need to allocate. You must not forget that you must also accommodate the size of the `node_t` struct when doing this. Note, you will need a pointer to where the free block was originally before to assign allocated properly. After you adjust the free block pointer forward you will need to update its size (which will be the original size minus the size of the requested number of bytes plus the "size of" `node_t`).
2. Adjust the previous pointer to point to the new location of the free block - this means you need to assign to the next pointer of the previous node to the new location of the free block. You need to pay attention to if previous is `NULL`. If it is, it means we are allocating from the first node in the free list which means we need to assign `head` (the pointer to the start of the free list) to the new free block.
3. Lastly, using the pointer you saved to the original free block in step 1 you must overlay/embed a `header_t` to the start of this piece of memory. This is easy, you simply assign to `*allocated` the pointer to the start of the original free block (performing the proper cast). After you do that, update the `size` field which will be the `size` requested in the parameter and assign the magic number.

Hints: remember, when you are getting the size of "things" in C you should use the `sizeof` operation. Also remember that sizeof returns the number of bytes. Because you are adjusting pointers by some number of bytes you must make sure to cast a pointer to a `node_t` to a `char*` to make sure you are adding bytes and not adding by `node_t` size. For example, here is how we adjusted the pointer to the free block:

```
size_t actual_size = size + sizeof(header_t);
*free_block = (node_t *)(((char *)*free_block) + actual_size);
```

`my_malloc`

```
void *my_malloc(size_t size)
```

The goal of this function is relatively straightforward since it primarily relies on the two helper functions `find_free` and `split`. First, you should define some local pointers to pass to these functions. We called them `found` and `previous` which are both `node_t*` types and allocated which is a `header_t*` type. Next, you need to make the call to `find_free` passing in the size and the `found` and `previous` pointers. After this call make sure you check to see if you found a free node. If not, you need to return `NULL`. After this you should call split passing in the appropriate arguments. Lastly, return a pointer to the allocated block. Note, you do not want to return the address to the start of the allocated block

- you must adjust this pointer to be just after the allocated block's header (`header_t`). This will require a little pointer arithmetic.

`coalesce`

```
void coalesce(node_t *free_block)
```

The goal of this function is to perform coalescing between adjacent nodes on the free list. This function will be called by `my_free` after freeing memory. The basic idea is to iterate over the free list starting at the provided `free_block` (given parameter) and check to see if a free block is adjacent to another free block. To do this you will need to compare the address of the current free block to the next free block to see if they are the same. If they are the same then they can be coalesced. You will need to figure out how to merge two adjacent free blocks into one. This is as easy as adjusting pointers like you would do in any linked list implementation. Note, do make sure you update the size of the merged node to reflect the merged number of bytes and the size of the `node_t` at the start of the free block.

Hints: When checking the addresses, make sure you remember to include the `node_t` size as part of the block size of the free node. If not, you will be off by `sizeof(node_t)` bytes which means you will never identify adjacent blocks. Here is what we did:

```
size_t block_size = p->size + sizeof(node_t);
...
if (((char *)curr_address) + block_size == next_address) { … }
```

`my_free`

```
void my_free(void *allocated)
```

This function is relatively easy with the help of the `coalesce` function above. First, cast the allocated parameter to a `header_t*` type. Remember, you need to adjust the pointer that is given by `sizeof(header_t)` to point to the actual start of the allocated block. After you do that you should use assert to ensure that the magic field is indeed equal to `MAGIC`. Next, you simply need to cast the `header_t*` to a `node_t*` and set the size to the size of bytes we are freeing. Lastly, link in the freed node into the free list by making this newly freed node the start of the heap - don't forget to assign its next pointer to the previous head!