Tuesday 16 February 2021

CECS 342-07 Spring 2021

Assignment 2

Overview

In this lab, you will implement your own dynamic memory allocator (heap manager) in C. We will use the simple (but somewhat inefficient) *free list* system of heap allocation. You will demonstrate how to use your allocator to allocate dynamic memory for a variety of C types.

Implementation

You must follow these implementation guidelines:

1. Define a struct to represent an allocation block:
```
struct Block {
    int block_size;           // # of bytes in the data section
    struct Block *next_block; // pointer to the next block
};
// The data section follows in memory after the above struct.
```
2. Determine the size of a Block value using **`sizeof(struct Block)`** and assign it to a global const variable. We'll refer to this value as the "overhead size".
3. Determine the size of a **`void*`** and save it in another const global.
4. Create a global pointer **`struct Block *free_head`**, which will always point to the first free block in the free list.
5. Create a function **`void my_initialize_heap(int size)`**, which uses **`malloc`** to initialize a buffer of the given **`size`** to use in your custom allocator. (This is the *only* time you can use **`malloc`** in the entire program.) Your global **`free_head`** should point to this buffer, and you should initialize the head with appropriate values for **`block_size`** and **`next_block`**.
6. Create a function **`void* my_malloc(int size)`**, which fills an allocation request of **`size`** bytes and returns a pointer to the data portion of the block used to satisfy the request.
   (a) Walk the free list starting at **`free_head`**, looking for a block with a large enough size to fit the request. If no blocks can be found, returns 0 (null). Use the **first fit** heuristic.
   (b) **`size`** can be any positive integer value, but any block you use must have a data size that is a multiple of your **`void* size`**. So if a **`void*`** is 4 bytes, and the function is told to allocate a 2 byte block, you would actually find a block with 4 bytes of data and use that, with 2 bytes being fragmentation.

(c) Once you have found a block to fit the data size, decide whether you need to split that block.
   i. If you do, then find the byte location of where the new block will start, based on the size of the block you are splitting and the size of the allocation request. Initialize a new block at that location by assigning its **`block_size`** and setting its **`next_block`** pointer to null. Reduce the size of the original block appropriately.
   ii. If you cannot split the block, then you need to redirect pointers **to** the block to point to the block that follows it, as if you are removing a node from a singly linked list. *WARNING:* The logic for a node in a linked list is **different** depending on whether or not the node is the **head** of the list. Draw it out and convince yourself why you need to account for this.
   iii. A block needs to be split if its data portion is large enough to fit the **`size`** being allocated, AND the excess space in the data portion is sufficient to fit another block with overhead and a minimum block size of **`sizeof(void*)`**.

(d) Return a pointer to the **data** region of the block, which is "overhead size" bytes past the start of the block. Use pointer arithmetic.

7. Create a function **`void my_free(void *data)`**, which deallocates a value that was allocated on the data heap. The pointer will be to the **data** portion of a block; move backwards in memory to find the block's overhead information, and then link it into the free list.

Testing Your Code

Test your code thoroughly by allocating values of various types. You should write (and turn in) your own testing main, which **at least** includes the following tests:

1. Allocate an **`int`**; print the address of the returned pointer. Free the **`int`**, then allocate another **`int`** and print its address. The addresses should be the same.
2. Allocate two individual **`int`** values and print their addresses; they should be exactly the size of your overhead plus the size of a **`void*`** apart.
3. Allocate three **`int`** values and print their addresses, then free the second of the three. Allocate a **`double`** and print its address; verify that the address is the same as the **`int`** that you freed.
4. Allocate one **`char`**, then allocate one **`int`**, and print their addresses. They should be exactly the same distance apart as in test #2.
5. Allocate space for a 100-element **`int`** array, then for one more **`int`** value. Verify that the address of the **`int`** value is **`100 * sizeof(int) +`** the size of your header after the array's address. Free the array. Verify that the **`int`**'s address and value has not changed.

Program

Write a program to use your tested allocator to solve a classic problem requiring dynamic storage:

1. Ask the user to enter a positive integer n.
2. Allocate space for an array of n integers.

3. Read n integers from standard in, into the array you allocated.
4. Calculate and print the standard deviation of the integers entered, using the formula

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

where x_i are the integers entered, and μ is the arithmetic mean of those numbers.

Deliverable

1. You can work on this assignment *in a group of up to 5 students.*
2. At the due date you will take a brief quiz to test your understanding of the assignment.
3. During the lab session on the due date each group will do
    1. A brief demonstration of the running applications.
    2. A presentation explaining the source code.
4. *Due date:* **Tuesday 9 March 2021** at the beginning of lecture.