ECE 650 Project #1 - Malloc Library

1. Project Overview

Project 1 was designed to implement several memory allocation functions (malloc() and free()) from the C standard library by using the C code.

In my implementation, a memory chunk contained two parts. The first part was a memory control block that saved basic information of the chunk, such as the byte length of the memory chunk, the availability of the chunk to save data and two pointers that pointed to the preceding and subsequent free memory chunks. I implemented the memory control block by using the structure in C. The second part was the memory block that actually saved user data.

The system maintained a double linked list that connected the adjacent free chunks. I set a pointer that pointed to the first free chunk and a pointer that pointed to the last free chunk. By doing that, I could easily locate a free chunk for allocating space. It was also convenient to remove and add a free chunk into the linked list.

Each memory chunk had two states, which could be the chunk that had been allocated for saving data and the free chunk. For the allocated one, I set its availability to 0 and its two pointers to NULL, which removed it from the linked list. For the unallocated one, I set its availability to 1 and added the chunk into the double linked list in a sorted way. A sorted way means the free chunk with a lower start address always comes before which with a higher start address in the linked list.

2. Implementation of Two Allocation Policies

First Fit and Best Fit are two popular memory allocation policies that have been implemented in project 1. I implemented two pairs of malloc() and free() functions.

For the malloc() function that applies the First Fit policy, the function took in the number of bytes for a memory allocation, added the number of bytes of the memory control block to calculate the full size, and used that value to find the first free chunk that can completely fit that number of bytes.

If there was no proper free chunk, a system call function called sbrk() was used to increase the total size of the heap. The sbrk() function returns the start address of that new chunk so that I can set its memory size to the size of memory control block plus

the size of user data, availability to 0 and two pointers to NULL.

If a free chunk with bigger size was found, I checked whether the remaining size of bytes could fit a size of memory control block in the future. If the remaining space was not sufficient to or can barely hold another memory control block, I allocated the whole chunk to the user and remove that from the linked list. Alternatively, if the remaining space was ample to hold a size of memory control block, then I split the chunk int two parts. The first part was used to the remaining free chunk, while the second part was the allocated chunk for user to save data. The reason why I chose the second part as the allocated chunk was that I could simply remove the allocated block from the linked list by modifying its memory size without changing its two pointers.

For the malloc() function that applies the Best Fit policy, the same strategy applied. The only difference was that, instead of finding the first suitable free chunk, I traversed the whole linked list to find the free chunk that had the smallest number of bytes greater than or equal to the requested allocation size.

The return address of the malloc() function was the start address of the chunk that saved the user data. As the pointer that pointed to the allocated chunk was a type in structure (struct *), I changed it into the character type (char *) so that adding the size of memory control block only added the bytes of it as the size of a character was one byte, which gave the correct address.

The free() function took in a pointer that pointed to the start of the user data, found the appointed chunk to free, set its availability to 1, added it into the linked list, and merged any two chunks that are adjacent. The adding mechanism was implemented by traversing the linked list and finding the first free chunk that had a larger start address than the start address of current chunk. If no such free chunk was found, current chunk was added at the end of the linked list.

Because the linked list was ordered in a sorted way, the merging mechanism was implemented by simply checking whether the last address of the former free chunk is equal to the start address of the current free chunk. If they were equal, I merged the two chunks by increasing the memory size of the former free chunk and deleted current free chunk from the linked list. Meanwhile, I checked whether the last address of current free chunk is equal to the start address of the later free chunk. If they were equal, I merged the two chunks by increasing the memory size of the current free chunk and deleted the later free chunk from the linked list.

To sum up, the adding mechanism took a time complexity of O(n), while the merging mechanism and deleting mechanism only took a time complexity of O(1) as it was operating a double linked list.

There was no difference between the free() function applying the First Fit policy and which applying the Best Fit policy.

3. Experimental Results

The results were obtained under the Linux system and recorded in the following table. Each memory control block occupies 32 bytes. Each result was calculated based on the average of ten testcases.

Table 1. Experimental Results of Project 1

Memory Size	Allocation Policy			
	First Fit		Best Fit	
	Execution Time (s)	Fragmentation	Execution Time (s)	Fragmentation
Equal	5.31	0.45	5.38	0.45
Small	2.26	0.06	0.65	0.02
Large	45.01	0.09	53.12	0.04

4. Analysis of the Results

The equal-size-allocations used the same 128 bytes in all its malloc calls, malloc'ed the same number of chunks, and free'ed the chunk in the same order as they were malloc'ed. According to Table 1, it can be observed that two allocation policies almost take the same execution time. The reason is that there is no difference between these two allocation policies if the program doesn't reallocate the chunk that has been free'ed before. The distributions of free chunks in these two allocation policies should always be the same. Additionally, the fragmentations of two allocation policy are the same as they are recorded halfway during the operation of the program, which proves the truth that equal-size-allocations malloc'ed the same size of chunks regardless of the allocation policy.

The small-range-random-allocations malloc'ed a large number of random regions (from 128 – 512 bytes), randomly free'ed 50 of them, and malloc'ed 50 more regions with random sizes (from 128 – 512 bytes). For the small-range-random-allocations, we can see that Best Fit works better than First Fit, with lower execution time and less

fragmentation. The primary cause is that First Fit may invoke sbrk() function to increase space in heap more frequently than Best Fit. For example, the double linked list contains two free chunks. The first chunk is much larger than the second one. After a malloc request of small size is made, First Fit tends to split the first chunk and provide a free chunk with medium size, while Best Fit will skip that chunk and use the second one to hold data. If, afterwards, a malloc request of large size is made, First Fit may need to do a system call to request a heap increment, while Best Fit can use the first free chunk to hold that data. The system call can be time-consuming, which delays the execution time of First Fit. Moreover, First Fit is poor in reusing free chunks based on the situation above, which leads to a high fragmentation.

The large-range-random-allocations malloc'ed a large number of random regions (from 32 – 64K bytes), randomly free'ed 50 of them, and malloc'ed 50 more regions with random sizes (from 32 – 64K bytes). For the large-range-random-allocations, Best Fit takes more time but lower fragmentation than First Fit. I think the reason is that Best Fit consumes more time in searching the most proper free chunk than simply finding the first available chunk in First Fit. However, First Fit wastes many free memory chunks as it has a high fragmentation.

As a result, different allocation policies have different advantages. If we have less memory space and pay more attention on using memory efficiently, we better choose the Best Fit policy. If we are dealing with large-scale data and take serious of the execution time, we'd better choose the First Fit policy.