Report of Malloc Library Project

Runya Liu(NetID: rl235)

For this assignment, I implemented my own version of several memory allocation functions such as first-fit allocation and best-fit allocation. These functions are similar to those found in the C standard library's malloc() and free() functions. The main difference is that I created my own implementation from scratch, allowing me to better understand the inner workings of memory allocation and deallocation. Overall, this assignment was a challenging and enlightening experience that deepened my understanding of memory management and the C programming language.

1 Implementation description

1.1 Node of Freed Linked List

In my implementation, I used a linked list to add and remove nodes that are freed. Each node is a struct with the following fields: size, next, and previous. The 'size' field stores the size of freed data that the node has. The 'next' field is a pointer to the next free node in the list, and the 'previous' field is a pointer to the previous free node. This allows for easy traversal and manipulation of the list, as well as quick access to the size of available freed memory. The struct is defined as follows:

```
struct node {
  size_t size;
  struct node * next;
  struct node * prev;
};

typedef struct node Metadata;
```

Additionally, I also included a pointer to the head of the linked list, which makes it easy to access the first node in the list and start traversing from there. This approach allowed for efficient and effective memory management, as it made it easy to find the best-fit or first-fit block of memory for allocation. Overall, the use of a linked list in my implementation greatly improved the performance and functionality of the memory allocation functions.

1.2 First-Fit Allocation

My first-fit allocation algorithm traverses through the linked list, searching for the first node that has a size greater than or equal to the required memory size. The algorithm uses a pointer that starts at the head of the list and moves to the next node in the list at each iteration. The pointer stops at the first node that has a size greater than or equal to the required memory size and returns a pointer to that node. If the traversal pointer reaches the end of the linked list and no node with the required size is found, the algorithm returns a null pointer, indicating that there is no available block of memory that can meet the requirement.

1.2 Reuse_Block

The first-fit allocation algorithm returns a pointer, either null or the pointer that has size greater than the requirement. However, if the first case is returned, where no node with the required size is found, it may also be desirable to see if the block can be reused for future memory allocation if its size is significantly greater than the requirement. Therefore, in this case, not only do we need to remove the node that satisfies the requirement, but also add a new node to the freed linked list to save space.

1.3 Allocate New Block

Allocating new blocks happens when there’s no node in the freed linked list that satisfies the requirement of the size needed. This can occur either at the very beginning when the head of the linked list is null, meaning there’s nothing yet in the linked list, or at a later stage when the list is traversed through and yet no node satisfies the requirement. In these cases, the memory allocation function uses the sbrk() function call to request more memory from the operating system's heap.

The sbrk() function is a system call that allows a program to increase its data space by a specified number of bytes. It takes an argument of the number of bytes to be allocated and returns a pointer to the start of the newly allocated memory. In this way, the memory allocation function can request more memory from the heap when there is no available block in the freed linked list that meets the required size.

1.4 Add_to_ll and Remove_from_ll

These two methods, first-fit allocation and best-fit allocation, involve adding newly freed nodes to the linked list or removing existing nodes that are now used to store data from the linked list. Although these two methods are standard in any linked list implementation, there are many nuances to implementing these two methods in the linked list. This is because there are many corner cases that may easily lead to a segmentation fault, which is a common error that occurs when a program tries to access a memory location that it is not allowed to access.

1.5 Check_adjacent

This method, called "Coalescing", checks if the new node added to the linked list happens to be adjacent to any of its neighbors. In my implementation, whenever a new node is added, either freed or reused, it will be checked for its adjacency. This would greatly save up memory space and thus improve efficiency.

When a new node is added, the algorithm checks if the previous and next nodes in the linked list are also free. If they are, the algorithm combines the three nodes into one larger block of memory, effectively eliminating any gaps in the memory heap. This process is known as "coalescing" and it helps to reduce fragmentation in the heap and improve the overall performance of the memory allocation algorithm.

This method also helps to reduce the number of nodes in the linked list, making it easier to traverse and manage. It also reduces the number of calls to the system's memory allocation functions, which can help to improve the performance of the program.

1.6 Best-Fit Allocation

Similar to first-fit allocation, best-fit allocation also traverses through the linked list to find a node that meets the requirement of the size needed. However, it differs in that it not only needs to have a node with size greater than the requirement, but it also needs to guarantee that it's the best node in the linked list that satisfies the requirement, meaning the difference between its size and the required size should be as little as possible.

In order to accomplish this, the algorithm needs to traverse through the entire linked list and compare the difference between the size of each node and the required size. It then chooses the node that has the smallest difference between its size and the required size. This ensures that the memory block allocated is as close in size to the required size as possible and minimizes wastage of memory.

1.7 ff_free() and bf_free()

Both first-fit and best-fit allocation methods are implemented in a similar way. Both have to add the node of the freed block to the linked list and then check its adjacency with its neighboring nodes to ensure maximum efficiency.

2 Performance Result Presentation

	First-fit			Best-fit		
	equal	Smaller	larger	equal	smaller	larger
Time(s)	20.256582	18.727687	61.525749	20.194477	6.766720	69.634726
Fragmentation	0.45000	0.073883	0.093421	0.45000	0.026958	0.041318

3 Result Analysis

First, from the results above, the time and fragmentation for equal size is the same for both first-fit and best-fit methods. It is because the allocation size each time is the same (both 128), and thus both methods would work the exact same way. First-fit allocation would take the first free block of size 128, and so it best-fit allocation. The fragmentation is both 0.45, close to 0.5, because the fragmentation is recorded half way through.

In terms of smaller than case, the program works with allocations of random size, ranging from 128 - 512 bytes ((calculated by rand()%13 + 4) * 32). The results show that the best-fit method is faster than the first-fit method. While best-fit takes slightly longer to search for the appropriate free block, it utilizes the available free blocks in a more efficient manner. For example, let's say we have three free blocks with sizes of 8, 12, and 10 (suppose they aren't adjacent to each

other, otherwise they would have been coalesced). Suppose we are allocating blocks of size 6, 8, and 12. In first-fit, we would only be able to use the 8 and 12 size blocks, and allocate new memory for the size 12 using sbrk() function. However, in best-fit, we would be able to use all three free blocks and call sbrk() less. This is why the results show that under this situation, the runtime for calling sbrk() is more dominant than iterating and searching through the free blocks, resulting in best-fit being faster than first-fit.

For large_range_rand_allocs, the program works with allocations of random size, ranging from 32 - 62K bytes (((calculated by rand()%2048 + 1) * 32). The process is similar to the small_range_rand_allocs scenario, but with a wider range of block sizes. The results show that first-fit performs better than best-fit in this scenario. This is likely due to the fact that the sizes of blocks are more random, making it less likely to find a suitable free block. As a result, best-fit spends more time searching for the appropriate block, which results in slower performance. In contrast, first-fit tends to perform better in such situations, as it prioritizes speed of iteration over the efficiency of memory usage.

My recommendation is that when the size of allocation is small, it’s more preferrable to use best-fit for maximum memory storage efficiency. When the data size allocated is large, it’s preferrable to apply first-fit for time efficiency.