ECE 650 My Malloc Library

Performance Study Report

Zhe Fan || zf70@duke.edu

January 25, 2023

1 Implementation Description

For this assignment, I implemented my own version of 2 memory allocation functions (malloc() and free()) from the C standard library.

For each space chunk, I define a struct using **Double LinkedList** to know the **size** of the chunk, whether or not it's **free**, and what the **next** and **previous** chunk.

```
  struct chunk_meta {
    size_t size;                //the size of this chunk
    int free;                   //it is free or not
    struct chunk_meta * next;
    struct chunk_meta * prev;
  };
typedef struct chunk_meta chunk;
```

I also define two variable to record the head and tail of free list:

```
  chunk * free_region_Start = NULL;
  chunk * free_region_End = NULL;
```

1.1 Malloc

For malloc, space should only be allocated when existing space is not enough. Thus given that we have this linked list structure, checking if we have a free chunk and return it. When we get a request of some size, we iterate through our linked list to see if there's a free chunk that's large enough. See Algorithm 1

In details, when the chunk is large enough, I also need to reuse free space efficiently if possible. To implementaion that, given that I have LinkedList, we can using splitting and merging in Algorithm 2:

If we don't find a free block, we'll have to request space using sbrk to allocate space.

Algorithm 1 : Malloc

```
Require: size_t size;
Ensure: size > 0
 1: if Space is clean then
 2:     Allocate space;
 3:     Check allocate sucess;
 4: else
 5:     Find free space;                        ▷ First Fit or Best Fit
 6:     if Space is not enough then
 7:         Allocate space;
 8:         Check allocate sucess;
 9:     else
10:         Try to reuse the chunk;
11:     end if
12: end if
13: return the chunk;
```

Algorithm 2 : Reuse chunk

```
Require: chunk * ptr, size_t size;
Ensure: ptr→size ≥ size
 1: if ptr→size is large enough to split then
 2:     Split chunk from ptr;
 3:     Update;             ▷ update head, tail and other related staff
 4: else
 5:     Update;             ▷ update head, tail and other related staff
 6: end if
 7: Update;                               ▷ update other staff in ptr
 8: return ptr chunk;
```

1.2 Free

For free(), first I upadate the size of the chunk and make the **free** = 1, then check the previous and next chunk if free or not, if free, then merge them.

2 Performance Result Presentation and Analysis

The performance results of my malloc functions are as below:

Table 2.1: Performance of First Fit and Best Fit.

Range Size	First Fit		Best Fit	
	Time(s)	Fragmentation	Time(s)	Fragmentation
Small	18.983	0.071	5.697	0.027
Equal	20.448	0.450	20.947	0.450
Large	61.294	0.093	65.364	0.040
*Data segment size	3768128		3595616	
*Data segment free space	268416		95904	

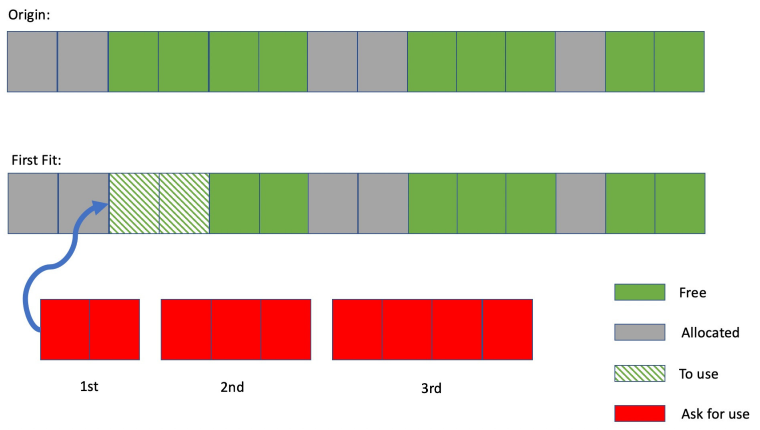

Figure 1: First Fit description.

From Table 2.1, we can see that for small range size, the execution time of Best Fit is less than that of First Fit. This is due to that in figure.1, if we first want to use 2 free chunks, finally the 3rd request will return that there is not enough space, which leading to call the sbrk() to request space from OS. However, calling sbrk() is more time-consuming than directly using free chunks in small range of size. While Best Fit will find the free chunk with minimum suitable space so that it will use all free chunks efficiently.

For equal range size, the two methods show that they have close execution time and fragmentation. This is because for equal range size, two methods run in the same way. Best Fit behaves in the same way as First Fit does namely.

For large range size, though Best Fit is faster than First Fit in small range size, iterating to find the free chunk with minimum suitable space in a large range will cost more time than sbrk(). Thus First Fit is faster than Best Fit in large range size.