Project 1 Report

Name: Zheyou Guo Duke ID: zg69

1. Abstract

This project is to implement malloc and free operation in C language. It calls system function sbrk to increase memory on heap and involves splitting and merging functions to utilize memory more reasonably. In addition, there are two different fitting policies in this project: first-fit and best-fit policy. Three testcases with different range of bytes are used to test their performance. Result shows that best-fit policy boasts an overall better performance than first-fit policy.

2. Design

The main design idea is to create a free-List which only tracks freed blocks. It connects freed blocks one by one in the order of time-series, namely the time when they are freed. In this way, it's easier and faster to search suitable freed block and malloc new memory on it, achieving to a maximization of utilization. As illustrated in figure1, this project creates a data structure called "linkedSpace" to track both allocated and freed blocks. It has metadata of the space allocated/freed, including size, availability, next freed block and previous freed block. For allocated blocks, their next freed block and previous freed block are set to NULL. Overall, the program contains 7 functions which have their own functionality. The whole design ideas can be clearer after working through their functionalities.

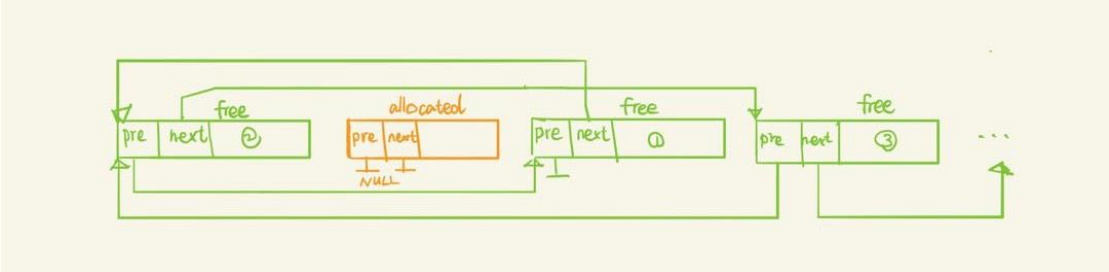

Fig 1. freeList

1. allocNew function: This function is to request more memory from system when there is no suitable freed block in freeList. It calls sbrk() and assigns four attributes(size, available, preFree, nextFree) to the new block.

2. split function: When the searched freed block is too large to be occupied, it will split the freed block into two pieces. To simplify the process of updating and connecting, it sets the first piece as freed block and the second piece as allocated block, as shown in figure 2. Therefore, it does not need to do anything to the freed block but just

changes its size.

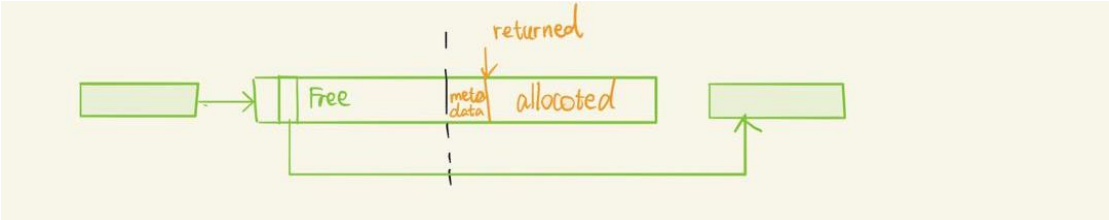

Fig 2. Split block

3. connectNew function: When the size of searched freed block is suitable to be used, the whole chunk will be directly allocated. In this situation, the freed block needs to be removed from the freeList, which requests a series of operations. It updates the preFree of the nextFree block and the nextFree of the preFree block. At the same time, preFree and nextFree blocks of itself are set to NULL, etc.

4. merge function: This function is to merge two adjacent freed blocks to avoid fragmentization. Because the freeList only tracks freed blocks, this function can only track the next adjacent block of the current freed block by adding its address to its size and size of metadata to get physical address of its next adjacent block. If this block is available(freed), it replaces this next block with current freed block and increases current freed block's size.

5. free function: There are two global variables called initialFree and currentFree which represents the head and tail of freeList respectively. The main job of this function is to connect newly freed block into freeList and update head and tail if needed.

6. ff_malloc function: This function works in first-fit policy. It starts searching from the head of freeList and breaks as long as it finds a freed block whose size is larger than size requested. Then it will decides whether to allocate the whole chunk by calling connectNew function or split the block by calling split function. If after searching from head to tail, there is no suitable freed block, it will request to allocate more memory on heap by calling allocNew function. In addition, the return value is

7. bf_malloc function: This function works for best-fit policy. The difference from ff_malloc is that it will search the freeList from start to end every time. It compares, updates and stores the most fitted freed block constantly. Then, it needs the second time of traversal to find the fitted freed block and then operates as ff_malloc does.

3. Result

Fitting Policy	Small_range_rand		Equal_range_rand		Large_range_rand	
	Time(s)	Frag.	Time(s)	Frag.	Time(s)	Frag.
First fit	5.455	0.109	0.019	0.450	27.974	0.157
Best fit	0.227	0.012	0.027	0.450	0.580	0.026

FF:

```
data_segment_size = 6700768, data_segment_free_space = 730560
Execution Time = 5.454633 seconds
Fragmentation  = 0.109026
zg69@ece551:~/project1-kit/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 0.019332 seconds
Fragmentation  = 0.450000
zg69@ece551:~/project1-kit/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 27.973583 seconds
Fragmentation  = 0.157169
```

BF:

```
zg69@ece551:~/project1-kit/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3787296, data_segment_free_space = 46656
Execution Time = 0.227325 seconds
Fragmentation  = 0.012319
zg69@ece551:~/project1-kit/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 0.027357 seconds
Fragmentation  = 0.450000
zg69@ece551:~/project1-kit/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 0.579624 seconds
Fragmentation  = 0.025815
```

From the result above, we can see that for both policies, the execution time for equal-range-rand testcase is very short and fragmentations are both 0.45. As for small-range-rand and large-range-rand, the execution time of large-range-rand is generally longer than small-range-rand. Although the fragmentation of these two testcase for both first-fit and best-fit is near 0, it can be noticed that small-range-rand has a smaller value than large-range-rand testcase.

There is also noticeable difference between fist-fit and best-fit policy. The execution time of best-fit policy for small-range-rand and large-range-rand is significantly shorter than that of first-fit policy. In addition, the fragmentation of best-fit is also smaller than first-fit policy.

4. Analysis

For phenomenon observed above, corresponding discussion and explanation are listed below.

The reason why execution time for both best-fit and first-fit is short is that in the code, there is a judgement condition in the process of searching suitable freed block to allocate that, if the size of current freed block is exactly equal to size we want, the searching process will terminate and directly allocate the whole chunk since it must be the best suitable block. Also, the number of freed blocks in freeList is small, which accelerates its searching time.

Because in the equal-range-rand testcase, the fragmentation is calculated in the middle, when half space is allocated and half is freed. Since it can always find the suitable freed block in freeLIist and re-occupy it, there won't be growth of space on the heap. The testcase firstly malloc some space and free half of the space, then it is just repetitively freeing one block and malloc it. Therefore, the fragmentation will always be approximately 1/2.

The fundamental reason why large-range-rand testcase costs a longer time is that splitting process happen more often in large range. When a large block, like 64KB is freed, it means this block can be split thousands of times for smaller blocks of size like 32B, so the time of splitting operation causes a longer run time. Also, the number of freed blocks in freeList becomes larger, which is a heavy burden for searching process, which also contributes to its longer run time. And due to the fact that large-range-rand has a much larger range of memory bytes, it is harder to fit into a suitable-size freed block, ending up with either occupying a whole chunk which is larger than we want or size is so small that it has to allocate a new piece of memory on heap.

Delving into root of small-range-rand and large-range-rand testcases, in ideal circumstances, the fragmentation should be zero. Because the testcase firstly allocates a certain space, and then mallocs a few and frees a few alternatively. Since the number of malloc and free are same, most of the newly malloced blocks should occupy previously freed blocks. But they have different randomly generated size, so the ideal circumstances will never happen, ending up with a value slightly larger than zero.

It is understandable that best-fit policy boasts a smaller fragmentation since it can always find the most suitable block to allocate. However, first-fit policy will always match to a block with larger remaining size. But why the runtime of best-fit is also more ideal than first-fit? It is contradictory because best-fit needs to traverse through freeList every time which is supposed to cost a longer time. One reasonable explanation is that best-fit method splits blocks less. It is more like for best-fit policy to find a freed block with size fit to size we want, so it dose not need to split the freed block and add it to freeList, instead, the whole block is directly allocated for use and is removed from freeList. In this way, the number of blocks in freeList is reduced, which relieves burden for searching and merging. To validate this assumption, I counted the number of blocks in freeList of best-fit policy and first-fit policy respectively. It shows that during process of malloc and free, the number of bocks of best-fit is 1000+ while that of first-fit is up to 7000+. So even if best-fit searches from start to end, the time cost is still smaller than first-fit. Not to mention its number of blocks keeps dropping at the end.