Fangting Ma (fm128)
650 HW1

Malloc Performance Study Report

Implementation description

I used a doubly linked list to store the list of free memory regions, which is ordered by their memory addresses. To store all the necessary header information of a memory block, I created a structure called metadata_t, which includes a flag variable “available” to track whether this block is free or occupied, “size” to stands for the usable size of this block to store data, and two pointers to track the information of the previous and the next block. A trick I used here is that I created two dummy nodes, which are allocated blocks with size 0, to represent the head and the tail node, so that I do not need to handle any special cases when searching the list or adding nodes to or removing nodes from the list.

Malloc and free are the two main parts of memory allocation. In malloc part, my function will search through the free list to check if there is any available block to store the requested size. If there is at least one available and fitted block, then this block will be allocated. Otherwise, it will use the sbrk() system call to expand the heap area. An important case that I took into consideration is that if the ideal free block is larger than required size, it will split the block into two separate blocks. The first part will be allocated, and the left part will become a new free block and replace the original free block into the free list. However, if the left part is not large enough to store the metadata of a block, then this free block will be directly allocated even if it has larger space than required.

When implementing malloc function, I used two different memory allocation policies, first fit and best fit. For first fit malloc, I searched the free list from the beginning, and once found a free block that has enough space to fit the required size, it will jump out of the loop and choose that free block so that it does not have to search the remaining list, which would be a waste of runtime. While for best fit malloc, I added two extra variables, one is a pointer to track the best fit block with the smallest size and the other is to store the available size of that block. Besides, it will iterate through all the free blocks and update these two variables if a smaller fitted block is found, since we need to pick the smallest size from all the fitted blocks. However, when the size of current block is equal to the required size, the function will directly jump out of the loop, and allocate that block.

In free part, my function will firstly find the appropriate location to add in the free list according to the starting address of blocks, then check if it needs to coalesce with the previous and/or the next block in the free list, and finally add the newly free block into the free list by adding or replacing according to different coalesce cases.

Results & Analysis

The performance results of my malloc and free implementations with respect to different allocation policies are shown as below. Two metrics are measured, runtime

and fragmentation ratio. Runtime can be used to stand for time efficiency while fragmentation can show the memory utilization efficiency. Lower fragmentation may reveal a better memory utilization since it shows the less space is free.

First fit:

	small	equal	large
Time (s)	6.941606	21.416572	39.842951
Fragmentation	0.060258	0.450011	0.093238

Best fit:

	small	equal	large
Time (s)	1.812324	21.943748	45.150834
Fragmentation	0.022104	0.450011	0.041815

We can see that when it works with allocations of random size with small range, best fit malloc will run faster and have a better space utilization than first fit malloc. While dealing with allocating large range size memory, best fit malloc runs slower but still maintains a lower fragmentation. Overall, it shows that best fit malloc performs better at space utilization efficiency for both cases, which may because it always chooses the fitted block with the smallest size, so the free blocks are utilized more efficiently compared to the first fit malloc. It may also improves the runtime when dealing with small range allocations so that the runtime of small range cases of best fit malloc is smaller than first fit. However, for large range allocations, iterating through the whole free list to find the best fit takes more time, so that the effect of better space utilization on runtime is offset. Therefore, best fit runs longer time than first fit malloc when allocates random size with large range.

It is obvious that both runtime and fragmentation of first fit and best fit policies for equal size allocations have quite similar values. It might reveal that they behave almost the same. Actually, they do behave basically the same when all the allocations are the same size, since for the best fit malloc, when it finds a free block with the same size to the required size, it will directly allocate that block and stop searching the left list.

Recommendations

Overall, I would recommend choosing the allocation policy according to your need. It seems that the best fit allocation policy behaves better at memory utilization efficiency for both small and large ranged-size allocations, so if you care more about memory utilization, you should choose the best fit malloc. However, when you care more about runtime, it would be better to choose first fit malloc if you need to work with large range size allocations, while to pick the best fit malloc when working with small range size data.