

ECE650 Malloc Library Project 1

Charlie Prior (cgp26)

Implementation

To implement malloc (for both algorithms), I used a block structure `block_t` that holds the length of the block including metadata, and pointers to the next and previous blocks if in the free list (NULL otherwise). I use a static pointer to `block_t` to hold the location of the head of the free list.

For the FF algorithm, we search through the free list to find the first block that has size (defined as `length - sizeof(block_t)`) greater than the requested size. For BF, we loop through once to find the first smallest block that has size greater than the requested size. We then pass the block to a function which splits the block if possible (if there will be enough leftover free space for the metadata of a second block), and updates the pointers of the free list.

If there is no block on the free list which is large enough to accommodate the requested size, then new space is allocated on the program heap using the `sbrk` function to extend the program break. The new block is initiated with its pointers to NULL.

To implement free, which is the same for both algorithms, we first calculate the address of the block from the data address passed in. Then we search through the free list to find the point at which to insert the now free block and update the pointers accordingly.

To implement the performance functions, we use a static pointer to store the address of the program break before any memory is allocated. This is updated once when the first malloc call is made. For `get_data_segment_size` this is compared to the current program break. For `get_data_segment_free_space_size` we loop through the free list adding the lengths of the blocks.

Performance

	BF Time (s)	FF Time (s)	BF Fragmentation	FF Fragmentation
Equal Size Allocs (100x10000)	83.62	0.22	45.0%	45.0%
Small Random Allocs (100x10000)	8.87	13.35	2.7%	7.4%
Large Random Allocs (50x10000)	56.03	40.03	4.1%	9.3%

Result Analysis

For the equal size allocs test, we would expect the fragmentation statistic to be the same for both algorithms since blocks are equal sizes and therefore there is nothing to optimize through the BF algorithm. The BF algorithm takes significantly longer in this case since we have to loop through the entire free list whereas the FF algorithm can take the first block.

For the random alloc tests, we expect the fragmentation statistic to be lower for the BF algorithm since we are more optimally able to fill the free list, without taking large blocks, splitting them and leaving a small leftover block. For the small allocs, the BF algorithm performs faster than the FF algorithm, probably because over time the free list is less fragmented and therefore has fewer small blocks the algorithm has to search through. On the other hand, for large allocs, FF still outperforms BF in terms of speed because there are always large enough blocks to allocate to.

In my approach, I used a doubly-linked list to implement the free list. One way that the speed efficiency of the BF algorithm could be improved would be to use a sorted tree data structure which would allow us to more efficiently find the smallest possible free block, without searching through the entire list. ($O(\log(n))$ vs $O(n)$.)

References

- Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX system programming handbook*. No Starch Press.
- Carvalho, A. (2017, June 14). *Implementing malloc and free*. Medium.
<https://medium.com/@andrestc/implementing-malloc-and-free-ba7e7704a473>