ID1206 Operating Systems : Seminar 2

Amiran Dugiev

December 2, 2021

1 Intro

This report goes through some of the challenges of implementing a malloc and free function, most of the assignment was filling in blanks which seems mundane to discuss so instead this report will go through the parts of implementation that presented difficulties, it will also discuss optimization and benchmarking.

2 Splitting a Block

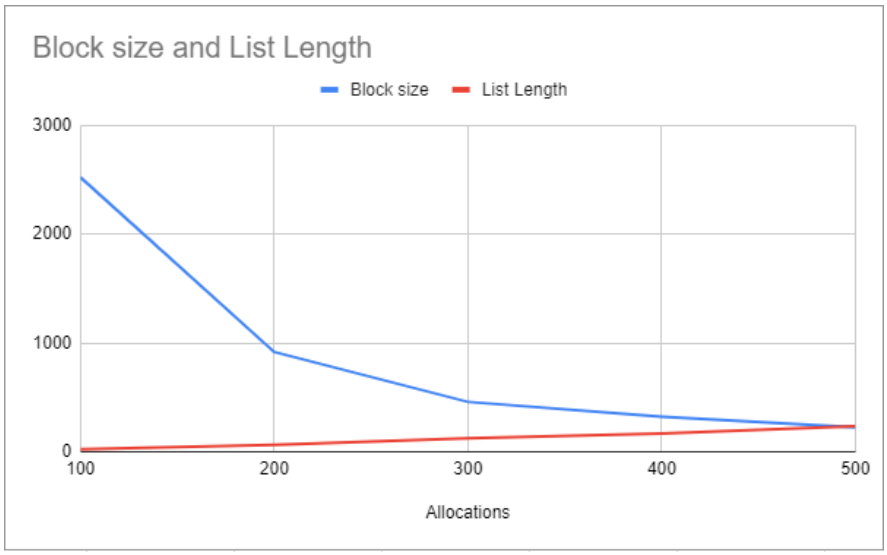
The first part of the seminar that presented difficulties was the function for splitting a block. This wasn't especially hard to implement in hindsight but because I didn't quite understand how the blocks were meant to used yet i was stuck for a while and had to draw the process on sheets of paper to understand and finally implement a solution.

```
struct head *split(struct head *block, int size)
{
    int rsize = block->size - size - HEAD;
    block->size = rsize;

    struct head *splt = after(block);
    splt->bsize = block->size;
    splt->bfree = block->free;
    splt->size = size;
    splt->free = FALSE;

    struct head *aft = after(splt);
    aft->bsize = splt->size;

    return splt;
}
```

After drawing the process I understood that I should remove memory from the block parameter equal to the size parameter and give the rest to a new split, the implementation of splitting reduces internal fragmentation.

3 First Implementation

The rest of the implementation went smoothly and it was time to benchmark my first version, I wanted to investigate the length and block sizes in the free list as the amount of allocations increase so a graph was created using sample data.



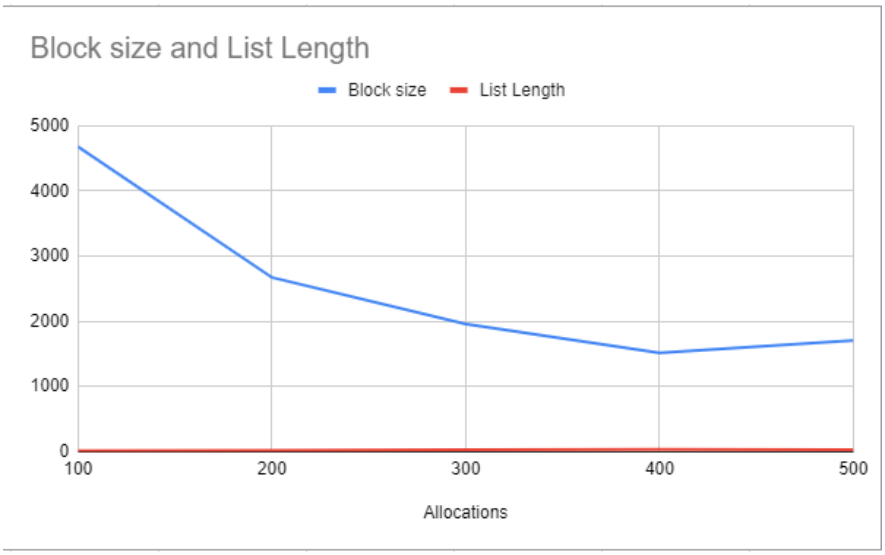
As the graph illustrates the length of the list seems to grow almost linearly while the size of the blocks seem to decrease exponentially

4 Coalescing

The next section of the seminar involved optimizing our first implementation by adding a function that merges blocks.

```
struct head *merge(struct head *block)
{
    struct head *aft = after(block);
    if (block->bfree)
    {
        struct head *bef = before(block);
        detach(bef);
        bef->size = bef->size + block->size + HEAD;
        aft->bsize = bef->size;
        block = bef;
    }

    if (aft->free)
    {
        detach(aft);
        block->size = block->size + aft->size + HEAD;
        aft = after(block);
        aft->bsize = block->size;
    }
    return block;
}
```

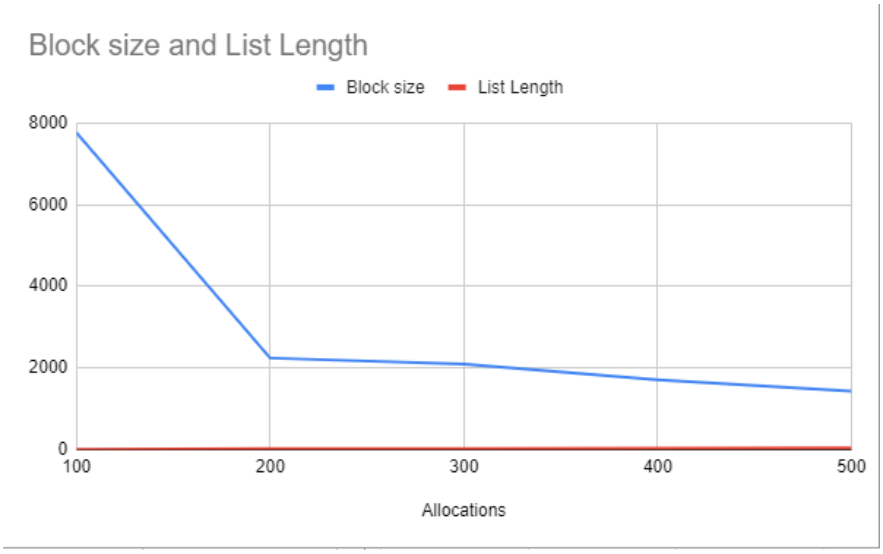
In this implementation the function checks if the free block before the current one is free and if it is detach the previous block, add the memory of the current one into it and make it the current one, effectively merging the previous block into the current one. It then computes a similar process with the block after but instead adds the memory of the block after to the current one.



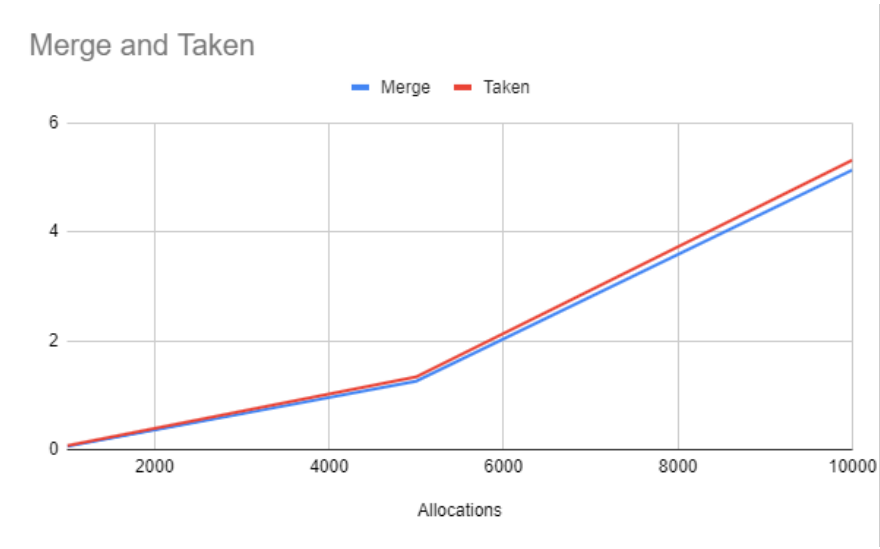
The implementation of merging has vastly decreased the free list length for example from 235 to 34 with 500 allocations but has made the block sizes bigger and made them decrease less drastically, even increasing at higher amounts of allocation while the free list decreases. This means that merging blocks reduces external fragmentation.

5 Optimizing Head Size

Further optimization is done by removing pointers from taken heads, thereby saving some memory during requests, we benchmark and graph this new implementation.



Finally we compare the execution times of each implementation so far.



Unfortunately the first implementation breaks above around 2000 allocations because the blocks become too small due to a lack of merging and external fragmentation makes us run out of usable free spaces. Taken runs slightly slower the merge only implementation however it taken up less space overall due to not wasting it on pointers, the difference is very marginal however.