Next, previous, before and after

Larisa Cof

HT20

1 Introduction

The purpose of this assignment was to implement malloc, using a scheme similar to dlmalloc (Doug Lee's malloc). Given some initial code, one was to complete the code segments in order to make the program work and also write a benchmark program to evaluate the performance. Assignment 4.2 was chosen in order to improve the performance.

While working on the assignment, there were some problems that emerged along the way:

- Dalloc failed a couple of times and it was not possible to allocate a certain amount, which demanded some troubleshooting.
- In general, segmentation faults occurred that were probably related to wrongful pointer references. One had to be very mindful when writing certain parts of the code that handled a lot of references, for example in split, merge and insertion sort.

2 Method

The main parts of the code were based on the given code segments in the assignment and are thus very straight-forward. Hence, the basic operations on a block, the freelist (double linked list), allocate (dalloc), free (dfree) and merge were implemented. A benchmark program was written in order to evaluate the performance and the results are represented with figures in section 3.

2.1 Benchmark

A core part of this assignment was to write a benchmark program in order to evaluate the performance. In the benchmark program, the implementations were evaluated with regards to:

- The time performance when increasing the amount of allocations.
- The length of the freelist as the amount of allocations increase.

2.2 Coalescing blocks

In this part of the assignment, the merge operation was implemented. Before a block is marked as free, the merge operation will be called by dfree immediately and insert the block in the freelist. The operation will check if the block before and/or after the freed block is free and in such case, merge these blocks. The skeleton code that was provided in the assignment was used. After merging, the new merged block is added to the front of the list.

Figure 1 shows the length of the freelist when running the code with and without the merge operation. Figure 2 shows the time performances when using the merge operation and not using it. As can be seen in Figure 1, there is a noticeable difference in the length of the freelist that indeed shortens by a lot when using the merge operation. In terms of the time performance, the two implementations do not differ in a considerable way.

2.2.1 First-fit

Initially, first-fit was implemented, where the first block of memory that is large enough is allocated. There is a benefit regarding the execution time from using first-fit since it does not need to search through the whole freelist. However, the downside of using first-fit is that it could leave small blocks of memory in the beginning of the list.

2.3 Insertion sort (assignment 4.2)

In order to improve the implementation, the freelist was sorted with insertion sort. Improving the implementation by sorting the list means that one needs to make an assumption about the list being initially unsorted, otherwise it may not work. Figure 3 shows the code for insertion sort.

2.3.1 Best-fit

To implement best-fit, insertion sort sorted the freelist in ascending order with concern to the block sizes. The blocks allocated will be the ones that are considered to be the best fit regarding to size (i.e. the block is equal to or larger than the requested memory and does not have to split). A benefit from using best-fit is that it reduces the amount of wasted memory. A downside from using best-fit is that it requires more performance in time since one has to sort the freelist.

2.3.2 Worst-fit

To implement worst-fit, insertion sort sorted the freelist in descending order with concern to the block sizes. In this way, the biggest block available will be allocated for each allocation made. The benefits from using worst-fit is that it reduces the amount of small memory and thus leaves big pieces of memory free.

Naturally, a downside that follows is the cost of performance. This implementation takes away big blocks of memory which is memory that could be needed. Also, it requires more performance in time since the freelist has to be sorted.

2.3.3 Best-fit, worst-fit and first-fit

The performance in best-fit was compared to worst-fit and first-fit. The result is presented in Figure 4. As can be seen in Figure 4, the freelist becomes significantly shorter with best-fit compared to worst-fit and first-fit. Considering the time performance, there is a slight difference between the three implementations but not one that is noticeable enough. The results are presented in Figure 5.

3 Figures

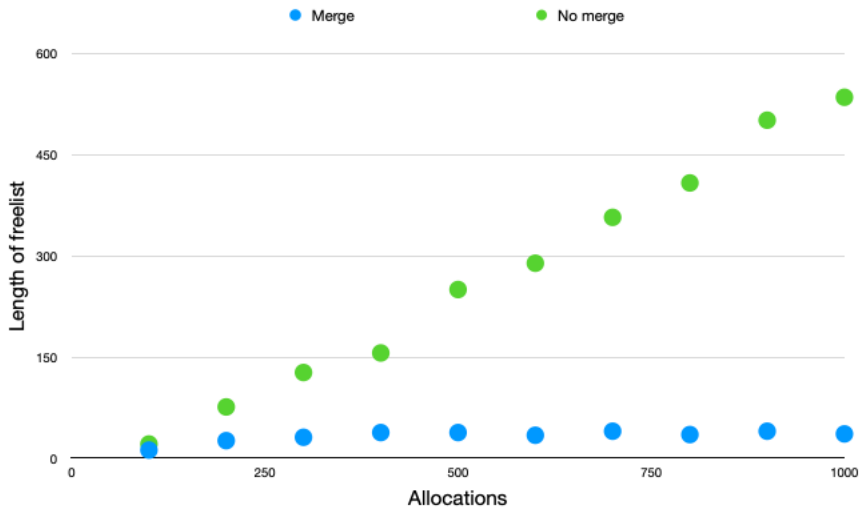

Figure 1: Merge vs. no merge

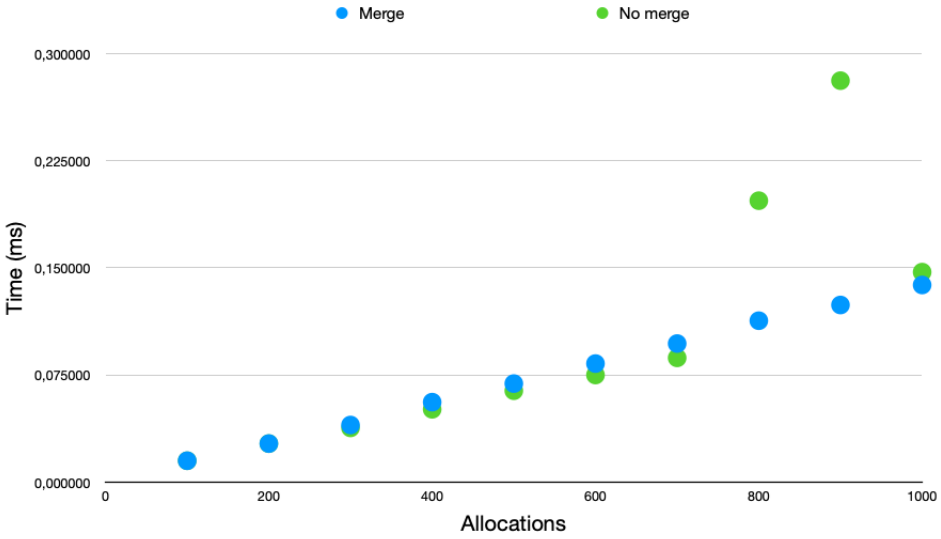

Figure 2: Time performance: Merge vs. no merge

```
void insertionsort(struct head *block){
  struct head *pointer = flist;
  if(pointer == NULL){
    block->prev = NULL;
    block->next = NULL;
    flist = block;
    return;
  }
  struct head *previousPointer = pointer->prev;
  while(pointer != NULL){
    if(block->size <= pointer->size){
      if(previousPointer != NULL){
        previousPointer->next = block;
      }
      block->prev = previousPointer;
      pointer->prev = block;
      block->next = pointer;

      if(pointer == flist){
        flist = block;
      }
      return;
    }
    previousPointer = pointer;
    pointer = pointer->next;
  }

  block->next = NULL;
  previousPointer->next = block;
  block->prev = previousPointer;
}
```

Figure 3: Insertion sort

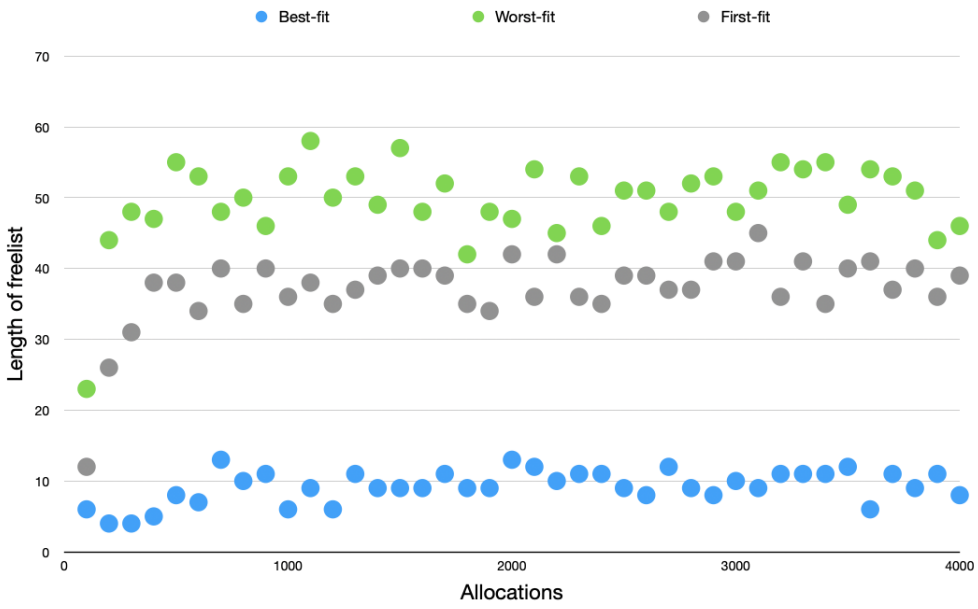

Figure 4: Best-fit, worst-fit and first-fit

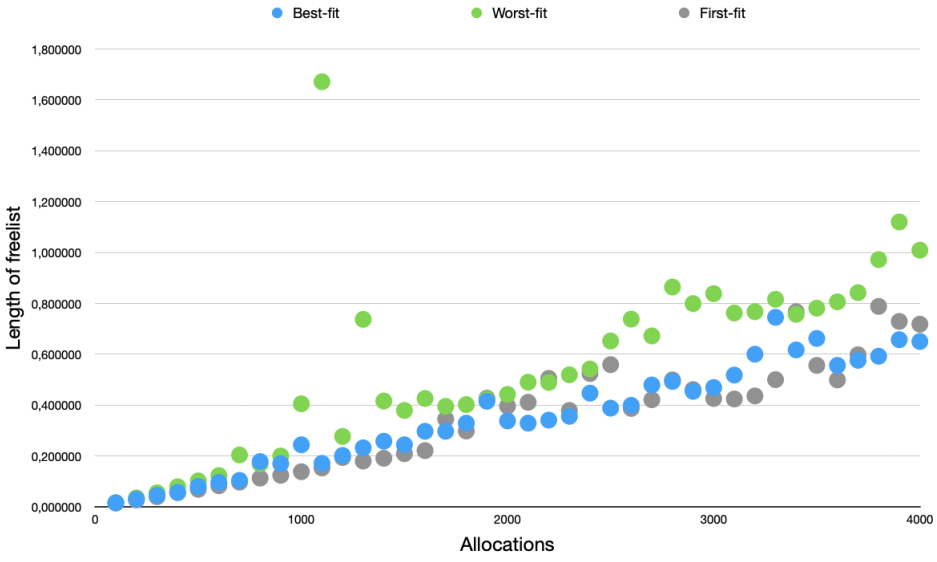

Figure 5: Time performance: Best-fit, worst-fit and first-fit