Malloc

27 november 2020

Philip Salqvist

phisal@kth.se
CINTE
1992-04-17

Innehåll

1 Introduction **2**

2 Merge **3**
2.1 Regular version 3
2.2 Optimized version 3

3 Benchmarks **4**
3.1 Benchmark 1 4
3.2 Benchmark 2 5

4 Result **6**

1 Introduction

The goal of the assignment was to implement the functions malloc and free instead of the ones defined in the standard library. The functions are supposed to be constructed in a way similar to Doug Lee's malloc. There are a multitude of challenges concerning this implementation. Starting with the free operation, one of the challenges concerned how to implement and handle the doubly linked list containing the memory areas that are freed. Furthermore, after freeing a block the block should merge with the ones before and after in memory.

The allocation function should be able to find a block of a suitable size, that is larger than the minimum size and a multiple of 8 bytes. If a different size is requested the size is supposed to be adjusted to the nearest larger multiple of eight. If a suitable block is found, we need a function that determines whether the block is large enough to split in two. A block should only be split if the block is sufficiently large to create a new block with the requested size and a header of 24 bytes, while the remaining block is large enough to pass the limit of minimum size. Additionally to this, one should try to optimize one or more functions and create appropriate benchmarks to determine whether an performance improvement in execution time or memory allocation was successful or not. In this report, the merge function was analyzed with the intent of improving execution time.

Two versions of the merging functions where created to be able to test the difference in speed. One that we will refer to as the regular version, and another that will be referred to as the optimized version in the continuation of this report. Finally, two different benchmarks was created with the intent of producing one execution where the number of merges increased. The result showed that we gain some time with the optimized version in both of the benchmarks. As it turns out, we gain even more time using the optimized version when more merges are made rather than fewer.

2 Merge

When the free function is called with a specific block as input, this block should be added to our doubly linked freelist. Before doing this, another function called merge should check whether adjacent blocks in memory could be merged with the requested block to be freed. In the merge operation, there are four scenarios that can occur. Either both of the adjacent blocks are free, only the one before is free, only the one after is free or none of the adjacent blocks are free. In the fourth case, the function simply returns the requested block as it originally was, so for our purposes we really only care about the first three scenarios. We will call these scenarios merge1, merge2, and merge3 in the continuation of the report.

2.1 Regular version

If a type of merge2 occurs in the regular version of merge, we will execute three important operations. First we will detach the block before the current block from the freelist, after this we will update the size of this block by adding the size of current block and it's header, and third we will update the after block's before size to match the size of our new block.

```
detach(bef);
bef->size = block->bsize + block->size + HEAD;
after(block)->bsize=block->size;
```

If a type of merge3 occurs we will simply detach the block after current block, update the size of the current block, and finally update the before size of the block after our new block in memory.

```
detach(aft);
block->size = block->size + aft->size + HEAD;
after(block)->bsize = block->size;
```

Finally, if a type of merge1 occurs we will simply run all of the operations in both merge2 and merge3. In this case, we will run six quite costly operations, and this is first and foremost the number of operations we will try to decrease in the optimized version.

2.2 Optimized version

In the optimized version, we have three types of explicitly defined cases matching the scenarios described above. Since merge1 is the one that we mainly want to

optimize, we will only focus on this implementation and leave the other two cases aside. If a type of merge1 occurs in the optimized version we will first update the size of the block before the current one in memory by adding the current block size, the after block size and the size of two headers. After this, we will simply detach the after block and finally update the before size of the block after our new merged block in memory.

```
bef->size = block->bsize + block->size + aft->size + 2*HEAD;
detach(aft);
after(block)->bsize = block->size;
```

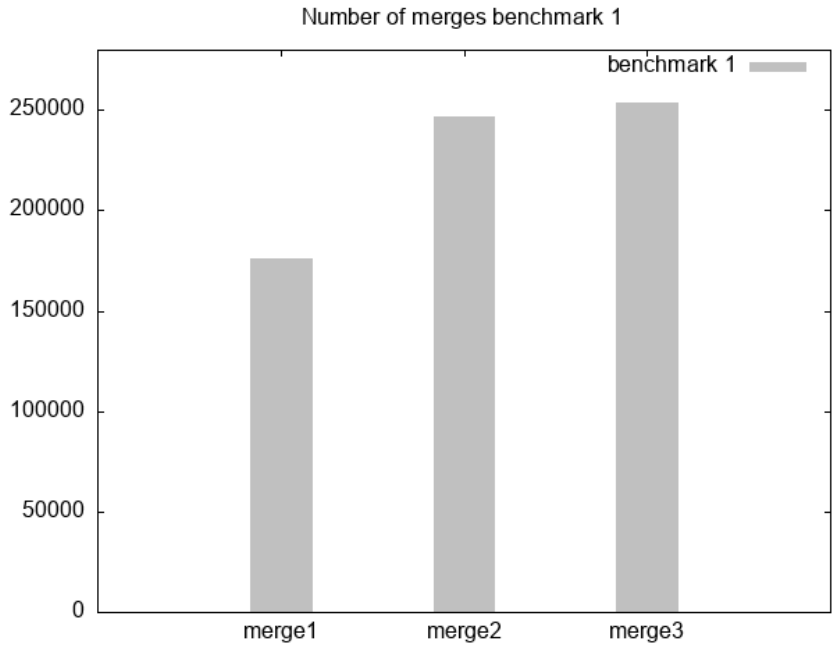
It is obvious that we have decreased the number of operation for merge1 quite substantially. In the regular version we had to perform a total of six operation, while the optimized version only performs three. In conclusion, we have reduced the number of operation in this scenario by a factor of two.

3 Benchmarks

To examine the performance of the optimized version in comparison with the regular version, we need appropriate benchmarks. We do not only want to determine whether the optimized version is better in general. We want to find out if the difference in execution time between the regular and optimized version increases if the number of merges increases. Especially, we want to analyze what happens if we increase the number of merge1 made while keeping the number of merge2 and merge3 reasonably stable between the two benchmarks. In general we want to make a high number of allocation and free operations to make the execution time difference between the two functions more apparent. In both of the benchmarks we execute in a loop that runs a million iterations. When executing the two benchmarks, we create counters to keep track of the number of times the different kind of merges occur. We thus count number of executions for merge1, merge2 and merge3 to compare the two benchmarks.

3.1 Benchmark 1

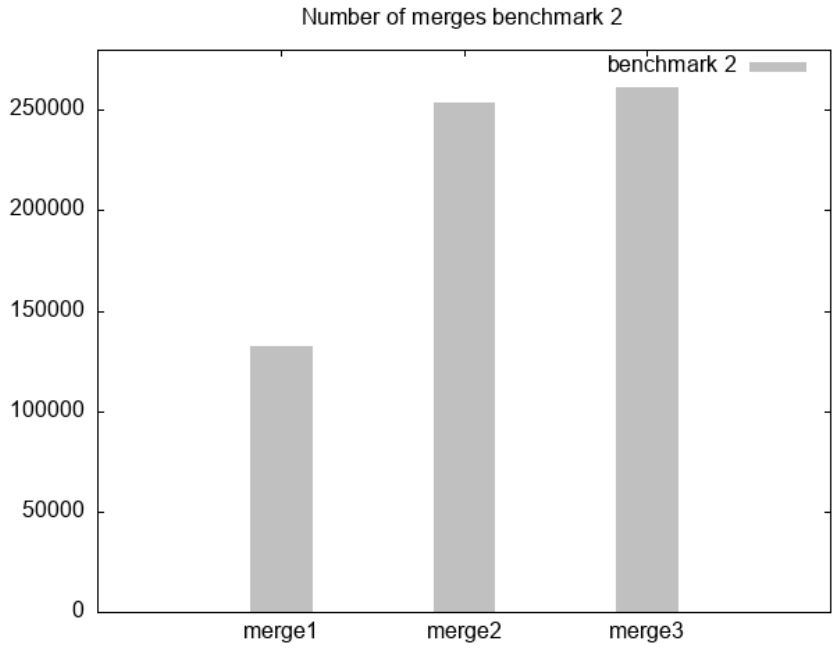
In benchmark 1, we want to keep the number of merges made by marge1 quite high. We randomly select the size when allocating, but not using a normal distribution. Instead, we make the distribution decrease exponentially between a minimum size of 8 and a maximum size of a 1000. The number of merges made by merge1, merge2 and merge3 in benchmark 1 can be viewed in Figure 1.



Figur 1: Number of merges made by marge1, merge2 and merge3 in benchmark 1.

3.2 Benchmark 2

In benchmark 2, we want to decrease the number of merges made by merge1 in relation to benchmark 2. We randomly select the size when allocating using a normal distribution between the minimum size of 8 and maximum size of 300. The number of merges made by merge1, merge2 and merge3 in benchmark 2 is visualized in Figure 2. We can see in the histogram that merge2 and merge3 is quite stable between the benchmarks, while merge 1 is substantially lower in benchmark 2.



Figur 2: Number of merges made by marge1, merge2 and merge3 in benchmark 2.

4 Result

The results from the two benchmarks is visualized in Table 1. We can see that we gain a moderate amount in execution time between the two versions when using benchmark 1, although no more than 0,044 seconds. When using benchmark 2, we can see that the execution time difference increases. In this case, we gain as much as 0,153 seconds. There could of course be other factors in play that plays a role in determining this change in execution time. Although it is fairly safe to say that we gain execution time when using the optimized version if the amount of merges made by merge1 increases.

Benchmark	Merge	Execution time
Benchmark 1	Regular version.	2,439 seconds.
Benchmark 1	Optimized version	2,395 seconds.
	Execution time difference:	0,044 seconds.
Benchmark 2	Regular version	2,369 seconds.
Benchmark 2	Optimized version	2,216 seconds.
	Execution time difference:	0,153 seconds.

Tabell 1: Execution time results comparing benchmarks and merge versions.