Next, previous, before and after

Elvira Laurin

October 2023

1 Introduction

This document describes the implementation of a simple version of `malloc` in the C programming language, which was the focus of a seminar in the course ID1206 Operating Systems, originally held during the fall of 2021 at KTH Royal Institute of Technology.

2 Implementation

This section describes the implementation details of the solution to the seminar task. The code is available as a GitHub repository, at https://github.com/elviralaurin/dalloc.

2.1 Freelist

The implementation is based on the use of a freelist, which is a double linked list that keeps track of the free blocks of memory.

We create a C struct `Head`, which is the overhead that stores the information associated with each block. The `Head` struct stores the size (number of bytes) and status (1 if the block is free and 0 otherwise) of the block, as well as the size and status of the block before it in memory. The information about the block before is kept to simplify the process of merging blocks, which we will look into later. We also store two pointers in the `Head` - one that points to the next block in the freelist, and one that points to the one previous.

Note that there is a crucial difference between the order of blocks as they appear in memory, and the order in which they may appear in the freelist. Furthermore, not all blocks will appear in the freelist at all times, but they will of course always appear in memory. To avoid confusion, we use the words "next" and "previous" when referring to the freelist, and the words "before" and "after" when referring to the order of the blocks in memory (hence the title of the seminar). See the schematic in figure 1.

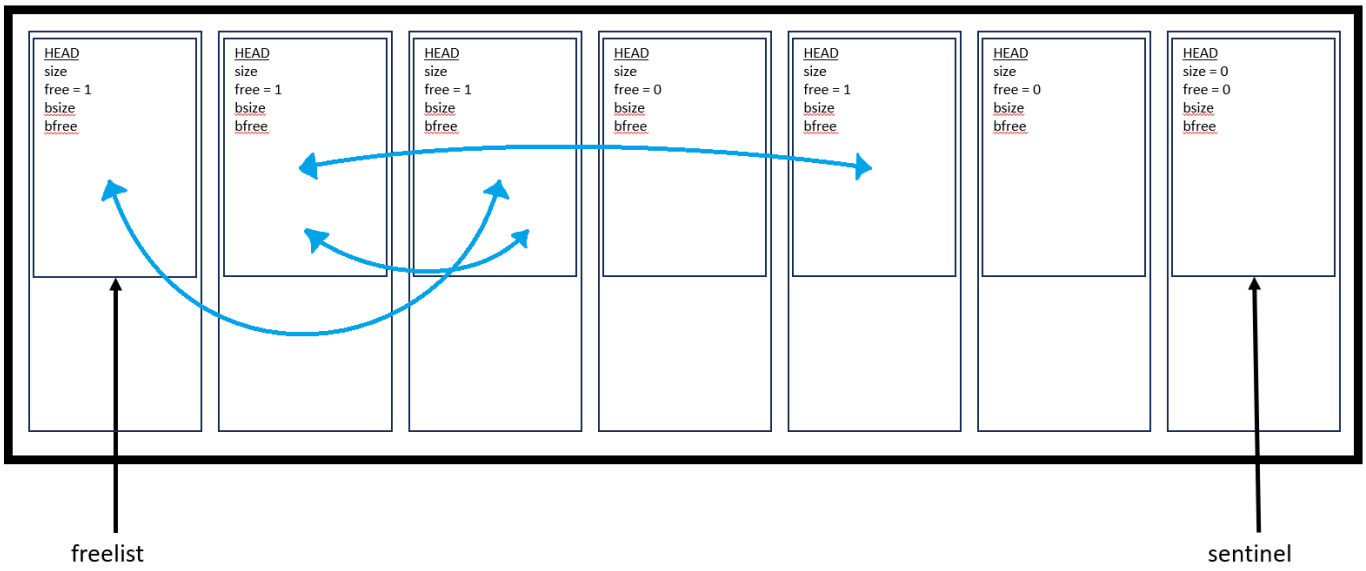

Figure 1: Schematic showing the relationship between the freelist and the layout of blocks in memory. The freelist is defined by the blue double links indicating the `next` and `prev` pointers, starting at the left-most block that has the `freelist` pointer. The `before` and `after` relationships are implemented as functions, since we are able to calculate the address of those given a block and its size. In the figure, the block `before` a given block is the one to its immediate left, and the block `after` it is the one to its immediate right.

2.2 Memory allocation

When the user calls the function `dalloc` - which is the name of the implementation of `malloc` described here - for the first time, memory is allocated through the Unix system call `mmap`[1]. We ask for a large chunk of memory, initially 64 KiB, which acts as our heap. The result of the `mmap` call is stored as a `Head` pointer, which we call `arena`. We set up the appropriate values for the size and status fields of the `arena` as well as for the `sentinel` block, which marks the end of the heap and prevents the user from being able to access memory outside of the arena. The freelist is also initiated, pointing at the beginning of the arena, and thus initially only containing a single block - the entire heap.

The algorithm used then works as follows, with `size` being the user-provided parameter, adjusted to be aligned to 8 bytes:

1. Iterate through the blocks in the freelist. As soon as a block bigger than or equally large as `size` is found, stop.
2. Check if the found block is big enough to be split. In other words, if the block were to be split according to the given `size`, would the remains be

big enough to support another block (at least 8 bytes, plus a header of 24 bytes)? If yes, split the block and put the remains back into the freelist.

3. Return a pointer to the block to the user, making sure to increment it in order to hide the header.

2.3 Memory de-allocation

Just like `malloc` has `free`, we need a way to de-allocate memory allocated with `dalloc`. In this implementation, this is done in a function we call `dfree`, which takes as input a pointer to some location in memory.

In the naive version, this is simply done by finding the header of the given block by decrementing the given pointer. We then reinsert it in the beginning of the freelist, making sure to update all relevant size and status fields.

In listing 1 below, the output of a simple benchmarking program is shown. We see that the more calls to `dalloc` and `dfree` are made, the more fragmented the freelist gets. Blocks in the freelist get smaller and smaller over time since we only ever split blocks, which results in external fragmentation. The benchmark used here runs for ten iterations, each time calling `dalloc` with a random request size between one and ten bytes. After each iteration, all blocks are freed using `dfree`, after which a summary is printed showing the current total number of blocks in the freelist, as well as the current number of blocks in the freelist that has the smallest possible size of eight bytes.

Listing 1: The (shortened) output of a benchmark program, demonstrating the external fragmentation of the current implementation.

```
RUNNING BENCHMARK...

~~~* Iteration 1 *~~~

Number of blocks in freelist: 11
Number of blocks in freelist with smallest size: 9

~~~* Iteration 2 *~~~

Number of blocks in freelist: 18
Number of blocks in freelist with smallest size: 15

[...]

~~~* Iteration 10 *~~~

Number of blocks in freelist: 28
Number of blocks in freelist with smallest size: 22
```

```
BENCHMARK COMPLETED.
```

In order to remedy this issue, we modify the code for `dfree` so that whenever it is called, we first merge the given block with its immediate `before` and `after` neighbours, provided that they are also free. The same benchmark program used above now gives the output shown in listing 2 below.

Listing 2: The (shortened) output of a benchmark program run with the improved version of dfree that includes merging of neighbouring free blocks.

```
RUNNING BENCHMARK...

~~~* Iteration 1 *~~~

Number of blocks in freelist: 1
Number of blocks in freelist with smallest size: 0

~~~* Iteration 2 *~~~

Number of blocks in freelist: 1
Number of blocks in freelist with smallest size: 0

[...]

~~~* Iteration 10 *~~~

Number of blocks in freelist: 1
Number of blocks in freelist with smallest size: 0

BENCHMARK COMPLETED.
```

As can be seen in listing 2 above, we no longer have the external fragmentation issue.

3 Decreasing the size of the header

The size of the header is currently 24 bytes, which, considering that the smallest block we allow is eight bytes, isn't very efficient.

Consider figure 1. The blue arrows indicate the `next` and `prev` pointers, which are stored in the `Head` struct. However, we see that this information is redundant for the blocks not currently in the freelist, i.e. the fourth and sixth blocks, as well as the sentinel. It might not seem like much, but in larger examples, this means that there is an opportunity for significant improvement here.

We create a second struct called `Taken`, which includes the same properties as `Head`, except the `next` and `prev` pointers. This gives `Taken` a size of 8 bytes,

which is a significant improvement to the 24 bytes of `Head`. We update the code to use `Taken` instead of `Head` whenever referring to the total size of the header, as well as when hiding and revealing the header in the `dalloc` and `dfree`.

A small benchmark is now written using 10^4 iterations, which allocates 1000 blocks of memory, each of size 16 bytes. The purpose is to measure any difference in execution time between the original, naive version (from here on denoted `dlmall_naive`), and the new, improved version (from here on denoted `dlmall`) using the `Taken` struct. The execution time of the benchmark was measured using the Linux `time` command[2], and the results, averaged over ten runs, can be seen in figure 2 below.

	dlmall_naive	dlmall
Wall clock time (s)	2.306	2.193
	2.264	2.252
	2.229	2.246
	2.235	2.238
	2.222	2.232
	2.241	2.218
	2.191	2.308
	2.248	2.234
	2.239	2.23
	2.24	2.21
Average	2.2415	2.2361

Figure 2: The average execution time in seconds, taken over ten runs, of the naive and improved version of `dlmall`.

As can be seen in figure 2 above, the execution time barely differs between the naive and improved versions, which is reasonable considering that the size of the header doesn't affect any algorithmic aspects of the solution. A possible explanation for `dlmall_naive` still getting slightly faster execution times is that it takes a bit more time for the lower-level processes of the kernel to handle the extra size, and this extra time accumulates over larger time scales. However, this advantage of `dlmall` over `dlmall_naive` is, at large, negligible.

The real advantage of `dlmall` is, unsurprisingly, that the overhead is much smaller. Consider the following: a user calls `dalloc` and asks for the smallest block our implementation gives out: 8 bytes. In `dlmall_naive`, this block has a 24 byte header, which results in the block consuming a total of $8 + 24 = 32$ bytes. In `dlmall`, this block only has an 8 byte header, resulting in a total of $8 + 8 = 16$ bytes. We note that the naive version uses double the amount of memory that the improved version does.

To summarize, the improved version provides a significant benefit in terms of

lower memory usage, to no extra cost in terms of execution time.

References

[1] *POSIX: mmap - map pages of memory*, The Open Group Base Specifications Issue 7, 2018 edition ed., The Austin Group, 2018.

[2] *POSIX: time - get time*, The Open Group Base Specifications Issue 7, 2018 edition ed., The Austin Group, 2018.