**Next, previous, before and after.**

**Johan Montelius**

**HT2019**

# 1 Introduction

This is an assignment where you will implement your own malloc using a scheme similar to dlmalloc, Doug Lee's malloc. You should be familiar with the role of the allocator and how to implement and benchmark a simple version of it. In this presentation you will not be given all details on how to do the implementation, but we will go through a general strategy.

# 2 The implementation

The problem with a free/malloc implementation is how to handle the freelist of available blocks. You could implement a strategy that will be able to return and coalesce a block in constant time. This is achieved by keeping hidden information in all blocks so that one can quickly determine if neighboring blocks are free. If neighbors are free the algorithm will merge these blocks. We will start with only one freelist to keep things simple but this is something that you should change in the final implementation.

Your implementation does not need to follow these guidelines but you can take them as a starting point. Feel free to improve or implement a different strategy (if it is equally good, better or have some advantage).

## 2.1 the setting

We will keep all free blocks in a double linked list. When a block is requested a suitable block is found and removed from the list. If however, the block is much larger than what is requested we can divide the block in two, not necessarily equal, parts. The remaining part can, depending on how the freelist is organized, remain in its position or be inserted at another position.

We talk about the `next` and `previous` block and this is related to the double linked list of free blocks. In the beginning the freelist will be unordered but one can of course experiment with ordering to improve performance.

We will also refer to the block *before* and *after* a given block. This is referring to how they are ordered in memory. All blocks are of course in a sequence and given a block we can talk about the block immediately before it (lower address), or after it (higher address).

When we free a block we should be able to find the block before and after a block, and determine if they are free. I one or both are free then the blocks

should be merged. This is the tricky part but if you get the basic operations right it should be a walk in the park.

In this implementation we will start with one large 64 Ki byte block. This is also the largest block that we will be able to provide but you can extend this limit in your final implementation (it will be slightly smaller since we need some bytes for a header structure).

## 2.2 operations on a block

A block (also often called *chunk*) consists of a *head* and a byte sequence. It is the byte sequence that we will hand out to the requester. We need to be able to determine the size of a block, or rather the size of the data sequence, and if it is taken or free. We also want to know if the block before it is free and the size of it.

```
struct head {
  uint16_t bfree;  //  2 bytes, the status of block before
  uint16_t bsize;  //  2 bytes, the size of block before
  uint16_t free;   //  2 bytes, the status of the block
  uint16_t size;   //  2 bytes, the size (max 2^16 i.e. 64 Ki byte)
  struct head *next;   //  8 bytes pointer
  struct head *prev;   //  8 bytes pointer
};
```

When you extend the implementation you can play around with how the head structure is represented. The important thing is that it is aligned to 8 bytes i.e 8, 16 or 24 bytes. In the above declaration we have used the type `uint16_t` to make each size and status field to be two bytes wide.

It is of course important that the head is as small as possible, since it is an overhead. If an application is requesting 8 bytes we have an overhead of 24 bytes which is not very good but it will do for now.

### some numbers

We will need some numbers as we go forward and define some macros. The true and false values require no explanation but the `HEAD` value is very important. It is the size of the head structure, 24 bytes, and we could of course have used the expression explicitly in the code but we might want to change it as we go.

```
#define TRUE 1
#define FALSE 0

#define HEAD  (sizeof(struct head))
```

The minimum size, `MIN()` that we will hand out is 8 bytes, or rather this is the minimum size of bytes, apart from the header, that a block will consist

of. We will change this as we improve the implementation so for now just think of a block consisting of 24 bytes header and 8 bytes of data.

```
#define MIN(size) (((size)>(8))?(size):(8))
```

The limit is the size a block has to be larger than in order to split it. If we want to split a block to accommodate a block of 32 bytes it has to be 62 (8 + 24 + 32) or larger. The smallest block we will split is a block of size 40 that could be divided up into two 8 byte blocks (24 + 40 = 24 + 8 + 24 + 8).

```
#define LIMIT(size) (MIN(0) + HEAD + size)
```

We use the regular way of hiding and retrieving the header and this time we write it as a macro.

```
#define MAGIC(memory) ((struct head*)memory - 1)

#define HIDE(block)   (void*)((struct head* )block + 1)
```

All memory that is returned to the requesting process needs to be aligned by 8 bytes. On a 32-bit architecture this would have been 4 bytes.

```
#define ALIGN 8
```

The arena is large block that we allocate in the beginning i.e. the whole heap. The size of this is limited since we will not be able to handle larger blocks than 64 Ki bytes (we only have a 16 bit size field). The size of the heap will thus be rather small to start with but we will be able to change this later.

```
#define ARENA (64*1024)
```

### before and after

The size information for a block will allow us to determine where the block after it is located. Implement a function `after()` that given a pointer to a block returns a pointer to the block after. You will find the block if you take the current pointer, cast it to a character pointer and then add the size of the block plus the size of a header.

```
struct head *after(struct head *block) {
  return (struct head*)( ..... );
}
```

In almost the same way you will be able to locate the block before a given block since we have the `bsize` field in the header.

```
struct head *before(struct head *block) {
  return (struct head*)( ..... );
}
```

### split a block

We also need a procedure that given a (large enough) block and a size, splits the block in two giving us a pointer to the second block. We first calculate the remaining size (the size of the block minus the requested size and size of a header). Once we know the remaining size we can find the second part (`splt` is after the block). We initialize the new block and patch the size of the block after to leave the blocks in a consistent state.

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

  struct head *splt =  ...
  splt->bsize = ...
  splt->bfree = ...
  splt->size = ...
  splt->free = ...

  struct head *aft = ..
  aft->bsize = ...

  return splt;
}
```

### a new block

To begin with we have to create new block. We do this using the *mmap()* system call. This procedure will allocate new memory for our process and here we allocate an area as large as possible. In this first run we only allow one arena so if there is already one allocated we return a null pointer.

Look-up the man pages for *mmap()* and see what the arguments mean. When you extend the implementation to handle larger blocks you will have to change these parameters.

```
struct head *arena = NULL;

struct head *new() {

  if(arena != NULL) {
    printf("one arena already allocated \n");
    return NULL;
  }
```

```
  // using mmap, but could have used sbrk
  struct head *new = mmap(NULL, ARENA,
                          PROT_READ | PROT_WRITE,
                          MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if(new == MAP_FAILED) {
    printf("mmap failed: error %d\n", errno);
    return NULL;
  }

  /* make room for head and dummy */
  uint size = ARENA - 2*HEAD;

  new->bfree = ...
  new->bsize = ...
  new->free= ...
  new->size = ...

  struct head *sentinel = after(new);

  /* only touch the status fields */
  sentinel->bfree = ...
  sentinel->bsize = ...
  sentinel->free = ...
  sentinel->size = ...

  /* this is the only arena we have */
  arena = (struct head*)new;

  return new;
}
```

Before we return the new block we set it up to prevent our algorithm to fall outside of the arena. We set the `bfree` flag of the block to false to prevent anyone from trying to merge it with a non existing block before. We also set up a sentinel block in the end of the arena and mark the `free` flag to false. This will prevent anyone from trying to merge the block with the sentinel block.

### the free list

All free blocks will be linked in a double linked list. We will in the first implementation only have one list and will not order it in any way. The list is double linked since we want to be able to extract a block from the list without having to search for its position. This will be important when we want to free a block and realize that the block before or after is also free and

should be merged.

We define a procedures to `detach()` a block from the list and one to `insert()` a new block in the list. Since the list is so far unordered we will always insert a block to the front of the list.

```
srcut head *flist;

void detach(struct head *block) {

  if (block->next != NULL)
    :

  if (block->prev != NULL)
    :
  else
    :
}

void insert(struct head *block) {
  block->next = ...
  block->prev = ...
  if (flist != NULL)
     :
  flist = ...
}
```

I now think you have all the smaller pieces of the puzzle to make the rest of the implementation quite easy.

## 2.3 allocate and free

In order to continue you need to have a good understanding of the algorithm that we shall use. If you haven't done so yet you should pick up a pen and draw what the operations should actually do. I could have included drawings in this description but you will learn more if you do your own.

The goal is of course to provide a user with two functions: `malloc()` and `free()`. In order to make life easier for us we will call these functions `dalloc()` and `dfree`. When we are requested to allocate a new memory area we do the following:

- Determine a suitable size that is an even multiple of `ALIGN` and not smaller than the minimum size.
- Go through the freelist and find the first block that is large enough to meet our request and unlink it from the list. If the freelist is empty you need to create the arena (if not already created).

- If the found block is so large that we could split it in two then do so and insert the remaining block in the freelist.
- Mark the found block as taken and make sure to also update the block after the taken block.
- Return a pointer to the beginning of the data segment i.e hide the header.

That shouldn't be that hard to implement given that you have the code for the basic operations that we need. This is a beginning, implement `adjust()` and `find()` and you're done.

```
void *dalloc(size_t request) {
  if( request <= 0 ){
    return ...
  }
  int size = adjust(request);
  struct head *taken = find(size);
  if (taken == NULL)
    return NULL;
  else
    return ....
}
```

To free a block is slightly more complicated but we will fake it in the beginning. We will simply insert a block into the freelist but we will not merge blocks. This will of course make life easy but we will of course loose a lot of memory in external fragmentation.

```
void dfree(void *memory) {

  if(memory != NULL) {
    struct head *block = ...

    struct head *aft = ...
    block->free = ...
    aft->bfree = ...
     :
  }
  return;
}
```

You should be done in five minutes, there is no hidden catch here.

**before you continue**

Implement the above functions in a file called *dlmall.c*. Also add a file *dlmall.h* that holds a declaration of the `dalloc()` and `dfree()` procedures. Then, in a file *test.c* write a *main()* procedure makes some call to `dalloc()` and `dfree()` - does it work?

You could also, do it it will help you as you start to make changes, in the file `dlmall.c` implement a procedure `sanity()` that checks if the freelist and arena looks ok. This procedure could for example check that all blocks in the freelist have the correct previous pointer and that they are all marked as free. The size should also be a multiple of ALIGN bytes and not less than our minimum.

You can also traverse all blocks starting from the `arena` pointer. Each block is found using the `after()` procedure and the `bfree` and `bsize` fields aft that block should match the fields of the current block. The blocks could of course be either free or taken (the free blocks should be in the freelist) but could come in any order. When we have our merge procedure up and running two consecutive blocks can not both be free (if so they should have been merged).

# 3 Your first implementation

When your dlmall allocator works as expected it's time to do some benchmarks. The things we want to know is:

- What is happening with the length of the freelist as we do more malloc/free operations.
- What is the sized of the blocks in the freelist?

You will need a benchmark program that can vary the requests given a min and max size. You should also be able to change the number of blocks requested and the number of blocks the application has allocated at a given time. The sequence of operations should preferably match a real program i.e. probably not be a sequence of identical requests.

Set up some benchmarks and try to describe the properties of your implementation.

## 3.1 coalescing blocks

You should now implement the merge operation. The merger operation will be called by `dfree()` immediately before we mark the block as free and insert it in the freelist. The merge procedure will check if the block immediate before and/or after the freed block is free and then merge these blocks. Here is some skeleton code that will get you going.

```
struct head *merge(struct head *block) {

  struct head *aft = after(block);

  if (block->bfree) {
    /* unlink the block before */

    /* calculate and set the total size of the merged blocks */

    /* update the block after the merged blocks*/

    /* continue with the merged block */
    block = ....
  }

  if(aft->free) {
    /* unlink the block  */

    /* calculate and set the total size of merged blocks */

    /* update the block after the merged block */
  }
  return block;
}
```

I strongly advice you to do some drawings to better understand what is going on. Which blocks need to be updated when we merge two blocks? What is happening if we merge three blocks? Could it be the case that we should merge more blocks?

### 3.2 do we gain anything?

Run the benchmarks again, what does it now look like? Have we improved the system, how and why?

## 4 Improving performance

There are some things that we could improve. Choose one, two, three or all of the suggestions below and give it a try. You report should include at least one of the improvements suggested below, it's solution and benchmarks that shows the difference.

### 4.1 the size of the head

The size of the head structure is one thing you can take a look at. If you have implemented it as above it is 24 bytes (2+2+2+2+8+8) which we might improve. Since we want to hand out segments aligned by 8 bytes it's not possible to improve it by a byte or two, we have to do something radical.

One idea is that the head structure only needs to hold its two pointers if it is a free block. Only then are the pointers needed for linking. A taken block only need the status flags and the sizes. The head would then only be 8 bytes which makes a big difference if the block should only hold a few bytes of user data.

The smallest block that we could hand out would now be 24 bytes consisting of a 8 byte header and 16 bytes of user data. We can not hand out anything smaller since when the block is returned it needs to be large enough to house a full header of 24 bytes.

To implement this you need to do some small changes. First define a structure `taken` that only holds the size and flag fields.

```
struct taken {
   :
   :
};
```

Now change the code that hides the header so that it only jumps a `take` structure forward. Also change the magic trick so you retrieve the block by jumping a `taken` structure backwards.

The only thing that is left is to change the macros that describes the size of the head and how the minimum allocation is calculated.

When you have it up and running you should do a benchmark to see if it means anything in reality. Of course we know that we save some memory since a 16 byte request in the original implementation would require a block of in total 40 bytes (24 + 16) whereas our improved implementation only requires 24 bytes (8 + 16 which is also the smallest block possible). Try a benchmark where you first allocate a thousand blocks of 16 bytes each and then run a loop where you write to all of these blocks a couple of thousand times - is there a difference in execution time, why?

If you actually want to implement even larger blocks (I don't think so but the idea is relevant for a 32-bit system and is what Doug Lee did in his first implementation) you could do another trick. The idea is that the size of the block before need only be known if it is free. If it is free it does not use the room allocated for the user data. Assume that we know that there is always room for eight bytes of user data, then we can shrink our own header to only our own size and two flags. If we see that the block before the current block is free than we know that the size is written in the last eight bytes of the user data i.e. immediately before our header.

If you know what you're doing - do we need 64 bits to encode a next or previous pointer? What if the pointers were offsets given the start of the arena? Given that an arena is not very large a 32 bit offset would be more than we need. This would mean that the two pointers could be encoded in 8 bytes which mean that the whole header is only 16 bytes (out of which 8 can be used by the user). Run the same benchmarks as before but now try using 8 byte user data.

### 4.2 order, order, Ooorrrdeeerrr!

One slight improvement that you can give a try is to do less work when we free a block. In the implementation as I have outlined we always remove a block from the freelist if it should be merged with another block. This might be avoided and I should not give you the solution since figuring out what to do is more fun than actually doing the implementation. Take a pen and paper and draw what is happening when we free a block and discover that the block before it is free.

This improvement relies on the fact that the lists is not ordered so we can have a block of any size anywhere in the list. If we change this assumption the improvement might not work.

Assume on the other hand, that the list is ordered with smaller blocks first, what would you do then? Try it and see if you can keep the list ordered without having to start an insertion from the beginning of the list every time.

Once you have an ordered list up and running one will of course think about selecting the best possible block when a request should be handled. The best is probably one that we do not have to split. Implement best-fit and see if the length of the list is reduced or if we make things worse when too many small blocks are created. Try worst-fit, does it make a difference?

If you know what your doing you could keep a circular list of free blocks. You would then have a `current position` which is the next block to consider when we're looking for a new block. You do not keep the list ordered in any way so the optimized merging described above will work fine. The first-fit approach should now give quite good performance. Note - make sure that you don't run around in circles looking for something that will not be found.

### 4.3 why only one list

Your implementation now keeps all the free blocks in one list. As this list grows we will of-course have to spend more time searching for suitable block. How about keeping a number of lists, one list for each possible size up to some threshold might be an idea. Hmm, 16, 24, 32... 128, that is 15 lists, and then one more list for anything else ...

There are several ways to order these lists, come up with a strategy and run some benchmarks, does it make a difference?

### 4.4 more arenas

We still have a very small heap. The 16 bits used for the size information could of course changed to 30 bits leaving four four bits to encode if a block is free or not. This would solve the problem but it would also mean that we would need to allocate a huge heap already from the beginning. A better strategy is to work with several arena where each arena is still limited to 64 Ki bytes.

The thing we have called the arena is only the consecutive memory that we have allocated and the freelist pointer has been treated as a global variable. What if we say that an arena is a data structure that holds: a memory segment divided into blocks and a freelist pointer. Each block now needs to have identifier of the arena to whom it belongs.

We could encode things in as few bits as possible but we now focus on how to make things work. Add another field to the header which is a pointer to an arena structure. This means that when we free a block we know which arena it belongs to and thus also which freelist it should be inserted in. Merging of blocks is not a problem since all blocks that could be merged are in the same arena.

Now when we can not find a block that is large enough we simply allocate a new arena. This is done by changing the `new()` procedure and the arena structure is of course taken from the first segment that is allocated by `mapp()`.

To keep track of all arenas we could link them together in a single linked list. We can then update the `sanity()` procedure to go through all arenas looking for inconsistencies.

If you have everything up and running you could try to run a benchmark using multiple threads. If we now have one arena list per thread, and this could be implemented using thread local storage, then each thread would allocate from its own arena. Most of the time it will free blocks that belong to one of its own arenas. Each arena will have a lock and to do any changes to the arena you need to take the lock. If most operations can be done without any lock contention the implementation should be quite efficient.

## 5 Summary

If you have completed the allocator and done some benchmarking you should have a better understanding of how *malloc()* and *free()* works - or rather could work; the regular malloc used by Linux is a bit more advance but not much.