Assignment 1: Memory management

TINBES02-2

Bastiaan Teeuwen

Hogeschool Rotterdam

May 8, 2019

1 man pages

I recommend you always use the `man` command (on Linux and macOS) when you can before looking something up online. Manual pages contains a bunch of information and can be a huge help, especially when figuring out how certain functions work in C.

As I won't describe every new function or command you learn in this document, I will denote them like this: **`name(n)`**. Where `name` is the name of the function or command and `n` is the section of the manual page. You can then look up the manual page on the command line like this:

```
$ man [n] [function]
```

Let's look up the manual page for `man` itself, which would be noted as **`man(1)`**:

```
$ man 1 man
```

If you're on Windows, use this website.

2 Memory management

The C library on your system abstracts the **`sbrk(2)`** system call for us using various functions to make life easier. You will recognize these from previous courses or C projects perhaps: **`malloc(3)`**, **`free(3)`**, **`calloc(3)`** and **`realloc(3)`**.

It is up to the user of our memory manager to keep track of the starting address of memory allocated by `malloc()`, but it's up to us to keep track of its other properties.

3 The assignment

Open the "`alloc.c`" and `alloc.h` file. Don't be overwhelmed, we will go though the program step by step.

We'll be tracking the memory we allocate with blocks. A block has several properties, which are already defined in a struct:

• `void *addr`	is the starting address of the allocated memory.
• `size_t size`	is the size of the allocated memory in bytes. This is further described below.
• `bool used`	defines whether the block is in use, or has been freed by `free()`.
• `struct block *next`	is a pointer to the next block. This is further described below.

`void *` is used to store any address, as a pointer is always the size of an address, regardless of its type. We use `void` as it allows us to dereference it to any other type without the need of a cast. It is also the type that `malloc()` and its siblings return.

`size_t` is a special type used to store the size of any object in memory. You can use it just like you'd use an `int` for example.

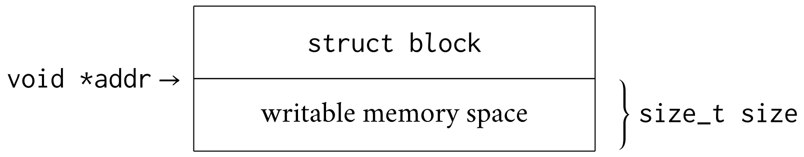
3.1 Allocating new blocks

We connect multiple block structures using a relation called a *linked list*. `struct block *head` is the first allocated block, the head of the list. The `next` pointer in the structure points to the subsequent block. The `next` pointer in the last allocated block in the list points to `NULL`, which also marks the tail of the linked list.

`uintptr_t kern_end` points to the end of the kernel, all memory past the address in this variables is free for use.

Several functions are also already predefined. We'll start by writing the `block_alloc()` function body. This function should look for the end of the linked list, fill in the properties of the block and return the starting address of the new writable memory space.

We also need a place to store the block structure itself. This will be stored at the start of the new memory space. In the structure we'll have the `addr` pointer point to the first writable address. The `size` variable will store the size of the writable memory space. In memory this should look something like this:



Now it's time to set the properties of the new memory space in our block structure. The values of a `struct *` can be accessed with `->`, like this for example:

```
struct block *bptr;
bptr->size = 4;
```

You can get the size of the structure like this:

```
sizeof(struct block)
```

⟶ Make the `bptr` pointer point to the start of the new memory space. If this is the first block, use `kern_end` as starting point. Then, set `addr` and `size` referring to the figure above. Also set `used` accordingly.

⟶ In the `block_alloc()` function, add the address of the block stucture and the `n` parameter together. This is the start of the writable memory space.

⟶ Return the value of starting address of the writable memory space.

3.2 Using a new block

Let's move to `malloc()`, the main user interface for your memory manager. This function should return the starting address of a writable memory space of at least `size` bytes to the user as a `void *`.

Fortunately, all the complex functionality for memory allocation is already in the `block_alloc()` function you've just written.

⟶ All that's left to do is to call `block_alloc()` and return the starting address of the writable memory space. Also check if the size is not 0.

⟶ Again, beware that `block_alloc()` may also return `NULL`, in which case `malloc()` should return `NULL` as well.

Right now, this function doesn't do much except for calling `block_alloc()`. We are not finished however and will continue in section 3.4.

Refer to section 3.5 if you want to test your memory manager at this point.

3.3 Freeing a block

This is where the information we have stored about blocks comes in handy. As `free()` only takes a pointer to the starting address of a memory space, we have to retrieve its size and used status from the block structure ourselves.

⟶ `free()` should call `block_free` immediately, unless `ptr` is `NULL`. `block_free()` takes a pointer to a block structure as parameter, so be sure to subtract the size of the structure before passing it along.

⟶ In the `block_free()` function, set the `used` flag of the block to `false` so we can use it again later. We'll do this is the next section where `block_get()` will be implemented.

Refer to section 3.5 if you want to test your memory manager at this point.

3.4 Reusing an unused block

If a block is still in the linked list and has its `used` flag set to false, we have to option to use it again.

⟶ In the `block_get()` function, use a loop to walk through the linked list (with `next`). Check the used flag and if the size of the block is equal or larger than `size`. Then, set the `used` flag accordingly and return the starting address.

⟶ In the `malloc()` function, first do a check for unused blocks before allocating a new one.

3.5 Testing your memory manager

You should now have a relatively simple but working memory manager! Testing the final result can be done by calling `malloc()` and `free()` from the `main()` function.

To verify correct behavior, call `malloc()` twice or more, `free()` once or more in another order the memory was allocated. Then `malloc()` again and if everything is alright, this shouldn't give any "segmentation fault" errors during runtime.

Also try to store something in the writable memory space allocated by your memory manager. You can copy a constant string to a memory location using **`strcpy(3)`** for example. This shouldn't give any "segmentation fault" errors either.

3.6 Bonus points

You can get bonus points if you extend functionality even further, some ideas:

- Implement `calloc()` or `realloc()`.
- Split unused blocks into multiple blocks if its size is larger than requested.