# CS3106: Practical 1 (Malloc) - Report

Matric Number: 170002815

13th October 2019

## 1 Overview

The objective of this practical was to implement a memory allocation library, which includes the ability for users to request variable sized regions of memory and free them after use - similar to C's `malloc(...)` and *free(...)* library calls.

The practical specification required the implementation of a given interface (*myalloc.h*), and also requested the use of a free list, as well as support for coalescing adjacent free regions of memory. My approach for these tasks is discussed in the *Design & Implementation* section of this report.

Testing of the program, and how completed extensions were approached are discussed in sections 3 and 4 respectively. The report is completed by an evaluation and conclusion, which reflect back on the learning objectives of the practical, and also evaluate how successful I have been in relation to the specification.

## 2 Assumptions

To limit the scope creep of this practical, and keep the work I am completing focused on the learning objectives, I created some practical-wide assumptions. These assumptions are influenced by the system I am developing my implementation on, and are as follows:

- **Assumption 1**: `myalloc(...)` and `myfree(...)` will always be called on a little endian system.
- **Assumption 2**: `myalloc(...)` and myfree(...) will always be called on a system using 64 bit (8 byte) words.
- **Assumption 3**: `myalloc(...)` and myfree(...) will always be called on a system using both word addressing and word alignment.

## 3 Design & Implementation

### 3.1 Linked-List Implementation

To begin this practical, I decided to create a basic linked-list implementation of the `myalloc(...)` and `myfree(...)` functions specified by `myalloc.h` header file.

This implementation involves excessive memory overhead, and does not implement a free list as mentioned in the specification, however it taught me a lot about the `mmap(...)` system call, and also how the practical can be approached in both a clean and effective manner.

The code for my linked-list implementation can be found in the `linked_list_implementation` directory, however it is not designed to be a fully functional implementation, and therefore does not pass all stacscheck tests.

## 3.2 Block Headers

During the execution of a program using `myalloc(...)` and `myfree(...)`, the memory allocated by the `mmap(...)` call is split up into different blocks. To keep track of a variety of information on the blocks (e.g. their size), it is common to use some memory of the block to store both a header and footer.

In my implementation of `myalloc(...)` and `myfree(...)`, I have decided to use only one header and no footer for each block. The structure of each header is optimised to the size of one word, and is influenced heavily by Wilson's et al. article on dynamic storage allocation [1]. The reasoning for the header structure is discussed in more depth in the *Extensions* section of this report, however a brief bullet point description of the structure is provided below, with a visualisation shown in Figure 1.

- Each header is the length of one word (8 bytes according to **Assumption 2** in *Section 2.2*).
- Each header is positioned at the start of a block, and contains information relating to that block.
- From the smallest bit to the largest bit, the header structure is as follows:
  - **Bit 0: Previous Block Free Tag**: Describes if the block before this block in the block list has been allocated to a program (0 for free, 1 for allocated). Bit is 0 if block is first in block list.
  - **Bit 1: This Block Free Tag**: Describes if this block has been allocated to a program (0 for free, 1 for allocated). Bit is 0 if block is last in block list.
  - **Bit 2: Final Block Tag**: Describes if this block is the final block in a block list (0 for not final, 1 for final).
  - **Bits 3-63: Block Size Bits**: Describe the size of this block, with the fourth bit (bit 3) representing $2^3$. Bits 0-3 are masked as 0 during the retrieval of the block size, so as not to interfere.

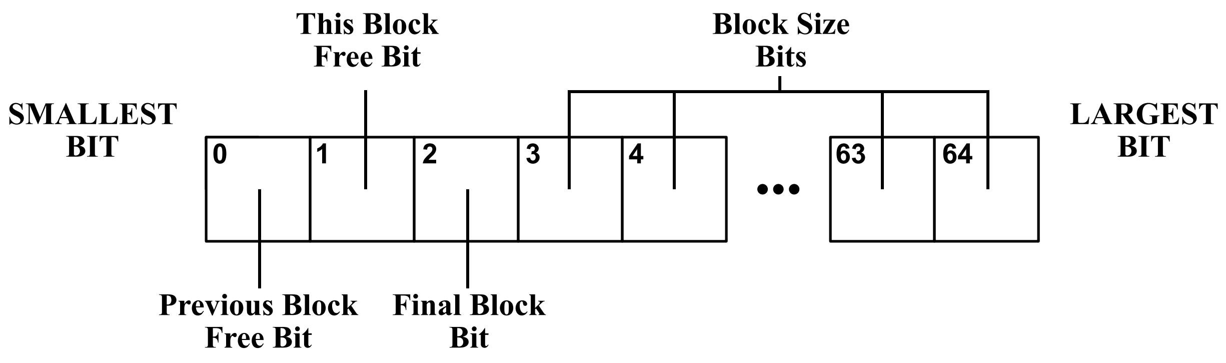


Figure 1: Block Header Structure

I created 4 word-long masks, each of which is used in a getter and setter designed for each piece of data in the header - this abstracts the complicated structure of the header, therefore making it easier to debug the program in the long run.

To prevent code repetition, I created a function named `getNextHeader(...)`, which uses the block size getter to calculate the position of the next header after the one provided to the function. I also decided to create a struct for the header, which abstracts the individual word used for the data from type `long` into type `Header`, making the program easier for a reader to understand.

You can find the code for my header, the masks, and the getters & setters towards the beginning of `myalloc.c` (Approx. lines 10 - 150).

## 3.3 Free List

By completing my initial, linked-list implementation of `myalloc(...)` and `myfree(...)`, it was clear to see how inefficient memory allocation could be without using a free list. Avoiding iterating through taken memory blocks gives free-list implementations a huge advantage over linked-list implementations with respect to computational time, however designing my one-word headers to work as a free list proved a challenging concept.

Once again, taking influence from Wilson's et al. article [1], I decided to implement a doubly-linked free list by storing the pointers to the previous free header and the next free header in the first two words of the free memory. This is a neat solution, however there are some negatives which are discussed in the *Extensions* section of this report.

To keep the retrieval of the free list pointers similar to other block data, I created a getter and setter for each pointer. This also allowed me to zero out the data which was previously stored, making sure the pointers were not being interfered with by left over bits.

## 3.4 Constants

To define constants in my program, I decided to use C's preprocessor `define` command, since they can easily be accessed at the top of the file, and are completely safe from manipulation. The two constants I defined for my program are `MMAP_REQUEST_SIZE`, which represents the volume of memory requested with each `mmap(...)` system call, and `MIN_WORD_ALLOC`, which represents the minimum number of words each `myalloc(...)` call must be rounded up to - this is discussed further in the *Extensions* section of this report.

## 3.5 Global Variables

Global variables are usually considered a bad thing when programming in C, however I found them to be the best way to cleanly and effectively store pointers required in different functions. The variables I stored globally include `memToAlloc`, which points to the block of memory retrieved by the initial `mmap(...)` call, `firstHeader`, which points to the first block header of the block list, and `firstFreeHeader`, which points to the first header in the free list.

## 3.6 Setup: myallocSetup(...)

The first time `myalloc(...)` is called, memory needs to be requested from the OS and subsequently formatted correctly for my program. I have decided create an individual function (`myallocSetup(...)`) for this process, because the formatting of the memory takes up several lines of code, and isolating these lines from the `myalloc(...)` function makes it easier to follow and debug.

The first few lines of (`myallocSetup(...)`) call the `mmap(...)` system call with the `MAP_PRIVATE|MAP_ANONYMOUS` flags, creating a memory mapping which is not backed by a file - a pointer to the memory from this mapping is stored in the `memToAlloc` global variable.

Once the OS has provided the requested memory, it is formatted to include a header in the first word of available space, and a header in the last word of available space. The first header is marked as available with size equal to $RequestedBlockSize - 2 \times HeaderSize$, and is pointed to by the `firstHeader` and `firstFreeHeader` global variables. The second header is marked as the final header for the block, as well as being not available for allocation.

The code for `myallocSetup(...)` can be found in `myalloc.c`, approx. lines 200 - 250.

### 3.7 Memory Allocation: myalloc(...)

`myalloc(...)` is the memory allocating function associated with `myalloc.c`, and is therefore fairly complicated. I have attempted to combat this complexity by dividing the code up into several functions (`myalloc(...)`, `removeFromFreeList(...)`, `createNewFreeHeader(...)`), as well as using comments where appropriate.

The code for `myalloc(...)` and its associated functions can be found in `myalloc.c` (approx. lines 250 - 400), and a high-level overview of the actions completed by the function is given below.

1. Complete setup (`myallocSetup(...)`) if not already completed by another call - check by evaluating if the global variables have been assigned.
2. Adjust request size to minimum of words specified by `MIN_WORD_ALLOC`, and round up any larger request to a length of multiple words. The reasoning for this step is explained in the *Extensions* section of this report.
3. Iterate over the free list, locating the first block in the free list which is large enough for the request. If no block of sufficient size is found, return NULL, else complete the following steps.
4. If there is enough room for a new header to represent a new block, complete the steps below(a - d), otherwise allocate the found block and remove it from the free list.
   (a) Create a new header, leaving enough space for the request size in the found block.
   (b) Adjust both block sizes specified in the header to accommodate new block.
   (c) Replace found block with new block in free list.
   (d) Allocate found block.

### 3.8 Memory Deallocation: myfree(...)

`myfree(...)` is the memory freeing function associated with `myalloc.c`. The functions `joinFreeBlocks(...)` and `coalesce(...)` are called by `myfree(...)`, and isolate code to simplify the development process and also prevent repetitions.

The code for `myfree(...)` and its associated functions can be found towards the end of `myalloc.c` (approx. lines 360 - 500), and a high-level overview of the actions completed by the function is given below.

1. Find header from given memory pointer.
2. Flip bits in header and next header to represent the block being free.
3. If no free blocks already exist, then set the `firstFreeHeader` global variable equal to the pointer to this header, otherwise complete the steps below.
4. If no adjacent blocks are free, then position this header at the front of the free list, otherwise complete the step below to coalesce adjacent free blocks - this step represents the function `coalesce(...)`.
5. Iterate through the free list, checking for a free block either side of each node within the list. Since coalescing happens *every* time `myfree(...)` is called, we can be sure that if a free block is found adjacent to one in the free list, it is the block which has just been freed, and can therefore be joined to the block in the free list using the `joinFreeBlocks(...)` function.

## 4 Testing

### 4.1 Block Description Function: describeBlocks(...)

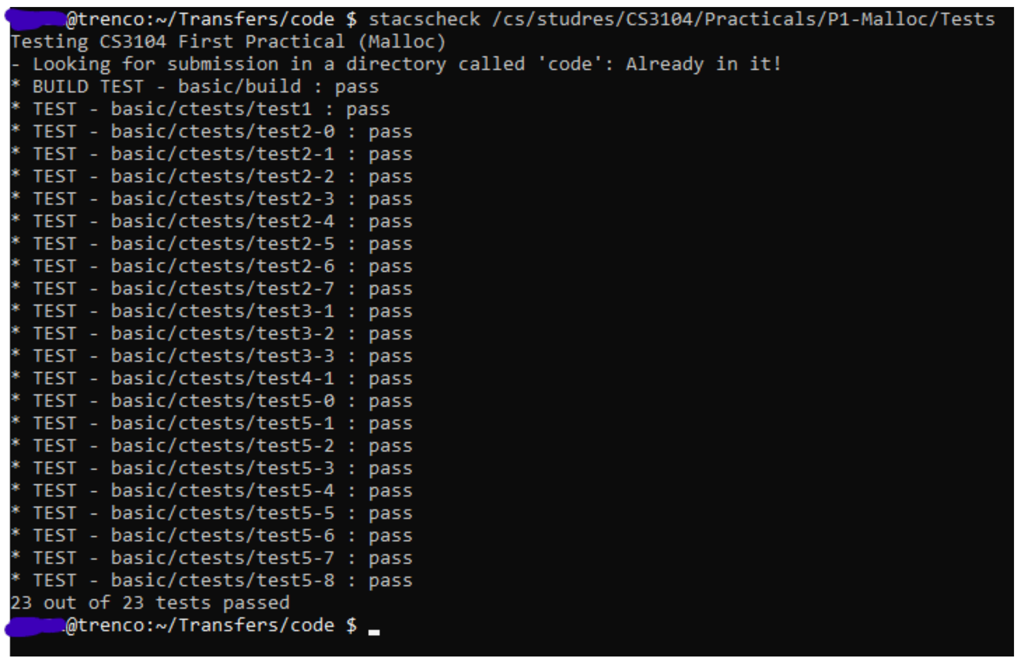
To help with both the testing and debugging of my system, I created a function, named `describeBlocks(...)` which printed out a short description of every block currently being handled by my program. The function works by using a while loop, and prints out the data by using the getters & setters discussed in *Section 3.2* and *Section 3.3*.

You can view the code for this function in `myalloc.c` (approx. lines 150 - 200), and can see an example output in *Figure 3*.

## 4.2 Stacscheck

During the initial stages of my work, small tests, such as checking the header getters & setters functioned properly, provided useful, however as my program became more complete, the provided stacscheck command became increasingly helpful.

Initially starting at 16/23 passes on my first run of stacscheck, I worked though each failure, correcting errors in my program to earn me an extra pass, working my way up to 23/23, as seen in *Figure 2*. To pass test 4-1, I used an initial `mmap(...)` request size of 16MB, which is quite large for a `malloc(...)` implementation, and would only offer a machine with 4GB of RAM a maximum of 250 processes - ways to overcome this "hack" are discussed in the *Evaluation & Conclusion* section of this report.



```
[redacted]@trenco:~/Transfers/code $ stacscheck /cs/studres/CS3104/Practicals/P1-Malloc/Tests
Testing CS3104 First Practical (Malloc)
- Looking for submission in a directory called 'code': Already in it!
* BUILD TEST - basic/build : pass
* TEST - basic/ctests/test1 : pass
* TEST - basic/ctests/test2-0 : pass
* TEST - basic/ctests/test2-1 : pass
* TEST - basic/ctests/test2-2 : pass
* TEST - basic/ctests/test2-3 : pass
* TEST - basic/ctests/test2-4 : pass
* TEST - basic/ctests/test2-5 : pass
* TEST - basic/ctests/test2-6 : pass
* TEST - basic/ctests/test2-7 : pass
* TEST - basic/ctests/test3-1 : pass
* TEST - basic/ctests/test3-2 : pass
* TEST - basic/ctests/test3-3 : pass
* TEST - basic/ctests/test4-1 : pass
* TEST - basic/ctests/test5-0 : pass
* TEST - basic/ctests/test5-1 : pass
* TEST - basic/ctests/test5-2 : pass
* TEST - basic/ctests/test5-3 : pass
* TEST - basic/ctests/test5-4 : pass
* TEST - basic/ctests/test5-5 : pass
* TEST - basic/ctests/test5-6 : pass
* TEST - basic/ctests/test5-7 : pass
* TEST - basic/ctests/test5-8 : pass
23 out of 23 tests passed
[redacted]@trenco:~/Transfers/code $
```

Figure 2: All stacscheck Tests Passed.

## 4.3 Personal Tests

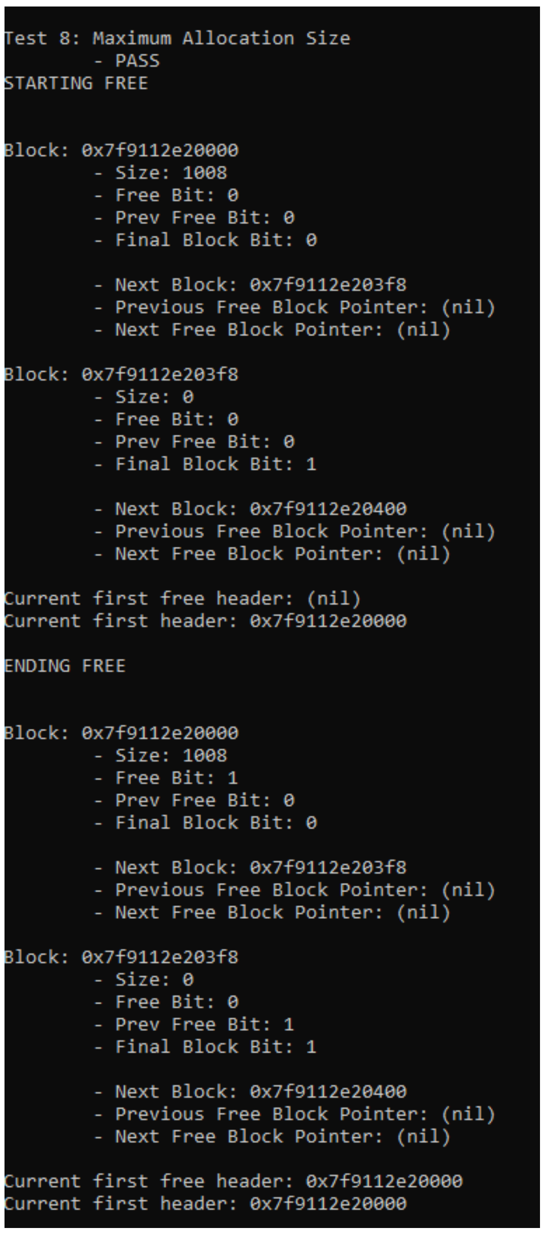
Once I had completed all 23 tests provided by stacscheck, I decided to create some of my own tests to further the verification of my program. These tests are written in the `mytests.c` file in the `code` directory of the project, and the `make` command will make an executable named `mytests` in the `code` directory. Each test is briefly described below, and the terminal output of `mytests.c` is shown in *Figure 4*.

- **Test 1 - Simple Malloc**: This test is designed check that `myalloc(...)` works without any errors with an expected input.
- **Test 2 - Simple Free**: This test is designed check that `myfree(...)` works without any errors with an expected input.
- **Test 3 - Allocate, Assign & Free**: This test is designed to check both the memory allocation and memory freeing functions work without any errors in an expected scenario.
- **Test 4 - Allocate, Multiple Assigns & Free**: Same as *Test 4*, except multiple values are assigned into the allocated memory to check all bytes are usable.
- **Test 5 - Allocate Three; Free Middle, then Outer Blocks**: This test is designed to verify the coalescing abilities of my program. First, 3 blocks are allocated, and then are freed in such a way to test 2 of the 3 coalescing scenarios. Another `myalloc(...)` call is made at the end of this test to check that the program is still functioning properly after coalescing.
- **Test 6 - Allocate Three; Free Outer, then Middle Blocks**: This test is the same as *Test 5*, however the freeing order is reversed in order to test the final coalescing scenario.
- **Test 7 - Allocate Exact Size for Minimum Block Size to be Created**: This test may seem trivial, however it is testing an important edge case for the program, where an allocation is made to leave just enough space for a header and two free list pointers (my program's minimum allocation). This test is related to my word alignment extension, and is designed to work only for `mmap(...)` requests of 1024 bytes.

- **Test 8 - Allocate Max Size for No Extra Header**: An easily overlooked edge case for a memory allocation program - if there's no room for a new block, an allocation can still be made. You can see my `blockDescription(...)` output of the final free completed in this test in *Figure 3* below.

  The first block described in *Figure 3* is shown to not be free and to have a size of 1008 bytes, while the second block described is shown to be the final block (just a header). Since the size of a header is 8 bytes, the total memory being used by the program at this stage is 1024 bytes (1008 + 2 headers), meaning that I have passed this test successfully.

  The third and fourth blocks described in *Figure 3* are the same blocks, except the former of the two blocks is now available to be allocated.



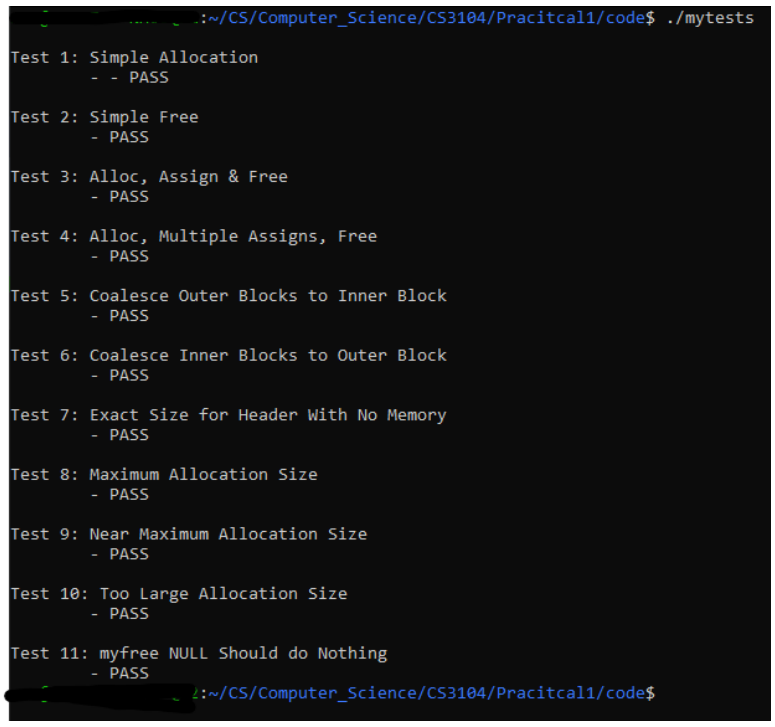
```
Test 8: Maximum Allocation Size
        - PASS
STARTING FREE


Block: 0x7f9112e20000
        - Size: 1008
        - Free Bit: 0
        - Prev Free Bit: 0
        - Final Block Bit: 0

        - Next Block: 0x7f9112e203f8
        - Previous Free Block Pointer: (nil)
        - Next Free Block Pointer: (nil)

Block: 0x7f9112e203f8
        - Size: 0
        - Free Bit: 0
        - Prev Free Bit: 0
        - Final Block Bit: 1

        - Next Block: 0x7f9112e20400
        - Previous Free Block Pointer: (nil)
        - Next Free Block Pointer: (nil)

Current first free header: (nil)
Current first header: 0x7f9112e20000

ENDING FREE


Block: 0x7f9112e20000
        - Size: 1008
        - Free Bit: 1
        - Prev Free Bit: 0
        - Final Block Bit: 0

        - Next Block: 0x7f9112e203f8
        - Previous Free Block Pointer: (nil)
        - Next Free Block Pointer: (nil)

Block: 0x7f9112e203f8
        - Size: 0
        - Free Bit: 0
        - Prev Free Bit: 1
        - Final Block Bit: 1

        - Next Block: 0x7f9112e20400
        - Previous Free Block Pointer: (nil)
        - Next Free Block Pointer: (nil)

Current first free header: 0x7f9112e20000
Current first header: 0x7f9112e20000
```

Figure 3: Test 8 Block Descriptions.

- **Test 9 - Smaller Request, but Still No Extra Header**: This test is very similar to *Test 8*, however a smaller request size is given to `myalloc(...)` to make sure my program handles this situation correctly and does not insert a header.

- **Test 10 - Request Size Too Large**: This test is designed to make sure my program does not try to allocate when it has no suitable region of memory. As seen by the code in `myalloc.c`, a value of NULL is desired, as this is conformative with the interface in `myalloc.h`.

- **Test 11 - Freeing NULL Pointer**: This test is designed to make sure `myfree(...)` conforms with the C library call `free(...)` in that calling the function on a NULL pointer does nothing.



```
[redacted]:~/CS/Computer_Science/CS3104/Pracitcal1/code$ ./mytests

Test 1: Simple Allocation
        - - PASS

Test 2: Simple Free
        - PASS

Test 3: Alloc, Assign & Free
        - PASS

Test 4: Alloc, Multiple Assigns, Free
        - PASS

Test 5: Coalesce Outer Blocks to Inner Block
        - PASS

Test 6: Coalesce Inner Blocks to Outer Block
        - PASS

Test 7: Exact Size for Header With No Memory
        - PASS

Test 8: Maximum Allocation Size
        - PASS

Test 9: Near Maximum Allocation Size
        - PASS

Test 10: Too Large Allocation Size
        - PASS

Test 11: myfree NULL Should do Nothing
        - PASS
[redacted]:~/CS/Computer_Science/CS3104/Pracitcal1/code$
```

Figure 4: All Personal Tests Passed.

## 5 Extensions

For my extensions, I decided to challenge myself with word alignment and header/footer optimisations, as I was lacking an understanding of the former, and practice with bitwise operations, which are prominently used in the latter. These choices also fit together neatly, as the header/footer optimisations I chose compress the header into the size of a single word.

As mentioned in *Section 3.2*, Wilson's et al. article on dynamic storage allocation [1] influenced my header design greatly; an overview of my chosen structure is given in *Section 3.2*, and a diagram is shown in *Figure 1*. *Section 3.2* covers the majority of what work I have completed for this extension, however an expanded explanation of the masks is given below.

For each piece of data I created a word-long mask designed to target a unique set of bits in the header. To target the first bit (the previous block free bit), I used a mask of `1`, giving me a word of all 0s except the last digit. To target the second and third bits (the this block free bit and the final block bit), I used masks of `2` and `4`, giving me words of all 0s except the second and third to last digits respectively.

Since I completed word-alignment as my other extension, I was able to be sure that the smallest three bits were always going to be unused in the header, because the block size would always be a multiple of 8 - these are the bits discussed in the previous paragraph. To target the bits representing the block size, it was important I did not overwrite the tag bits when setting the size, and also did not let the tag bits affect the result when reading the block size - I used a mask of `7` to accomplish this, as it provides a word of all 1s except the smallest three bits.

Because the header is heavily optimised, I decided to store the pointers to the previous and next free blocks in the allocation space of any free block - this is discussed in *Section 3.3*. The downside of this is that the minimum allocation that `myalloc(...)` can complete is 3 words long - this means any request smaller than three words given to the function is rounded up to exactly three words, which may be a large problem for a system with a small amount of RAM wanting to request only a few memory blocks.

Wilson's et al. article [1] also mentions that systems which are not word aligned are generally slower, and some are even illegal in relation to the computer architecture. My header, free header pointers, and minimum request size are already word aligned, and therefore by rounding any request up to the nearest word in `myalloc(...)`, I have made my program comply with word alignment.

## 6 Evaluation & Conclusion

Overall I believe I have been successful with this practical - I have implemented the basic specification to a high standard, and have completed both testing and extensions to accompany my work.

Reflecting on the learning objectives for this practical listed in the specification, I have greatly improved my C programming skills and have learned the reward of planning a project thoroughly - much less debugging! I have gained experience with using the `mmap(...)` system call, and have also gained much knowledge surrounding the topic of memory management algorithms and data structures.

Given more time, I would enjoy implementing incremental memory allocation to avoid the "hack" discussed in *Section 4.2*. Experimenting with page alignment and guard pages is also of interest to me.

## References

[1] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. *In: International Workshop on Memory Management*, 986:27–29, 10 1999.