

CS3104 Malloc Practical Report

170011474

October 2019

1 Overview

This practical requires students to implement a memory allocation library similar to `malloc()` and the corresponding `free()` in C. The memory allocation tool allows users to request variable sized regions of memory with a standard free list implemented to manage available memory blocks and release those blocks. Moreover, merging adjacent free blocks in the free list is required for the better performance and utilization rate.

2 Design & Implementation (Including Extensions)

2.1 A proper free list: the manager of free blocks.

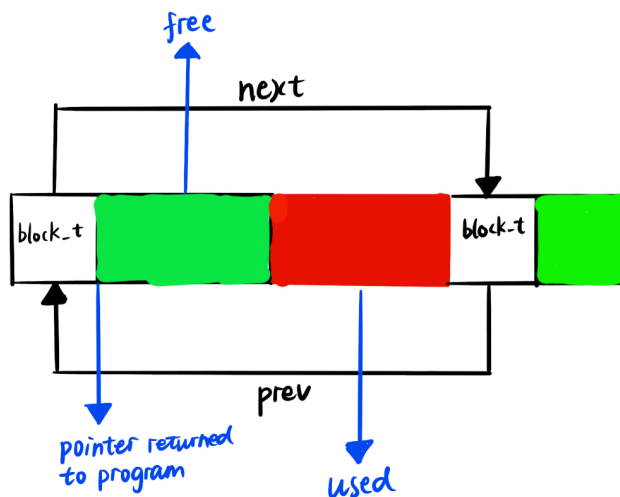


Figure 1: Free list

A free list is based on a doubly linked list with "block_t" as the node type. Each block_t performs as a *header* for each memory block, including information about the user size of the block (without the header its own size), an integer for free status (1 for free, 0 for occupied), a pointer to the previous block_t and a pointer to the next block_t (Wilson et al., 1995). The free list contains multiple basic operations: `add_block_to_freelist()` and `remove_block_from_freelist()` which adds or removes a memory block to or from the free list according to the current parameter block_t; `split()` will split a large block into two and add the remaining block into the freelist.

Requesting space from the free list and merging two continuous blocks A call of requesting function takes a size integer as a parameter and remove the first free block whose size is equal or greater than the desired size. *Notice that the de-*

sired size should be block_t size + size requested by users. If it is larger than required size, invokes a `split()` and return its remaining free memory block into the free list. `merge_adjacent_blocks()` will be invoked immediately after adding functions. **Therefore, it prevents the free list from duplicating blocks and lost blocks after requesting space. The merging function significantly eliminates external fragments since continuous blocks can be combined into a large one for the next request.**

2.2 Extension 1: Incremental Memory Allocation

```

    // request a free block from the free list
    ptr = request_space_from_freelist(size);

    if (!ptr)
    {
        ptr = request_space_from_heap(size);
        if (ptr == NULL)
            return NULL;
    }
    block_t *block = (block_t *)get_block_header_start(ptr);
    block->free = STATUS_NONFREE;

```

When a user request a memory block with **size** *n*, the program will

1. Firstly search for the current free list to see if an available blocks exists. If yes, then return the first valid block (if the block is larger than expected block, a `split()` will be used and the remaining block will be added to the free list)
2. If no, then it turns to the heap to ask for a new large chunk of memory via `mmap()`. The simplest way to obtain a block is just asking for the exact size requested by the user plus the size of `block_t`, while this would significantly increase the number of calls to `mmap()`. To reduce it, we allocate a new large chunk with a fixed size 3 pages, defined as the `MIN_ALLOC_UNIT`:

```

#ifndef MIN_ALLOC_UNIT
#define MIN_ALLOC_UNIT 3 * sysconf(_SC_PAGESIZE) // 3 pages
#endif

```

```

int alloc_size = size >= MIN_ALLOC_UNIT ? final_size : MIN_ALLOC_UNIT;
block_t *new_block = mmap(0, alloc_size, PROT_READ | PROT_WRITE | PROT_EXEC,
    MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);

```

Notice that if the requested size is larger than unit, then the heap would allocate a block with exact size.

3. After `mmap()`, the metadata `block_t` on this block occupied the first 24 bytes. Now we invoke a `split()` to separate 24 bytes + requested size from the chunk and leave the remaining part into the free list. Since the metadata `block_t` header should be unseen from users, we then return a void pointer pointing to the genuine user block back.

2.3 Extension 2: Thread-safe Implementation

A global mutex to protect `mylloc()` and `myfree()`. When there are multiple threads requesting memory or freeing memory blocks concurrently, the critical section should be in the free list. For example, two threads might fight for the only valid block and manipulate the free list at the same time. This situation would result in removing a nonexistent block which was taken by the first thread even more severe consequences. So the global mutex takes the responsibility to lock the free list when `myalloc()` and `myfree()` are running. In this way, it can handle multiple threads without concerning the safety and security of the free list.

```

pthread_mutex_t global_mutex = PTHREAD_MUTEX_INITIALIZER; // the global mutex

```

2.4 Debugging methods

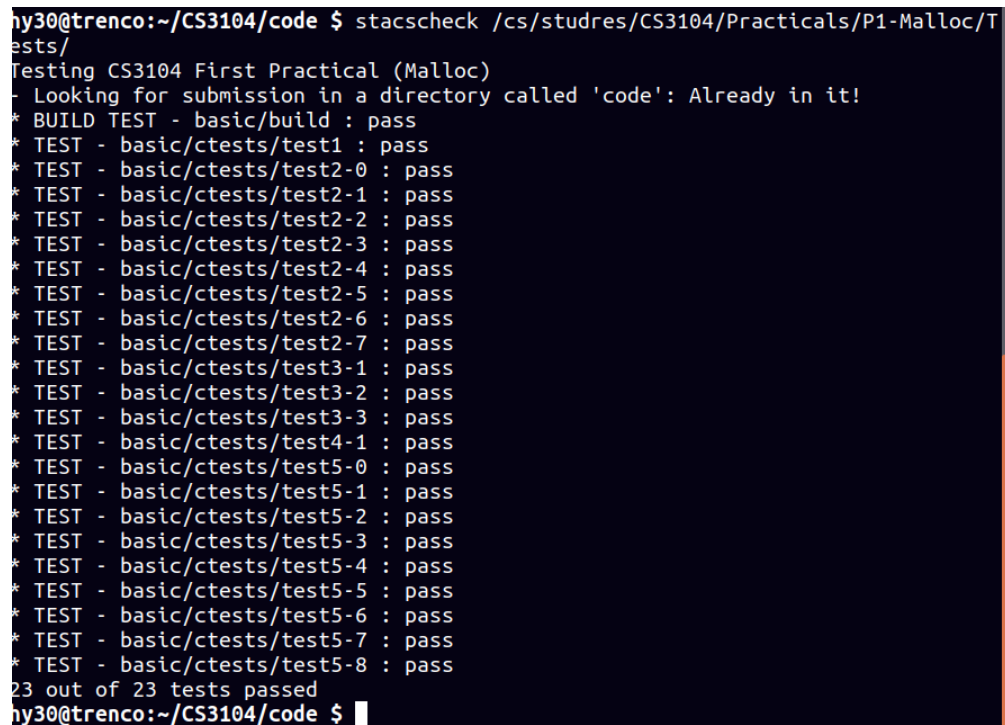
Debugging methods. Printing out debugging information is indispensable for a low-level C program, so a simple implementation of a debugging method was written. Each time a block is successfully allocated, related information of the block, including starting address, size, and free status, will be printed out.

Therefore, in the following testing phase, it would offer a understandable trace of memory allocation, hence enhances the efficiency of testing and debugging.

```
=====
Block Information
Block header starting address: 139966497779712
Block userspace starting address: 139966497779736
Block userspace size: 16384
Block free status: 0
=====
```

3 Testing

3.1 Autocheck



```
hy30@trencos:~/CS3104/code $ staccoscheck /cs/studres/CS3104/Practicals/P1-Malloc/T
ests/
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
hy30@trencos:~/CS3104/code $
```

Figure 2: Autocheck Results

3.2 Self-Testing for Main Practical(Incremental Allocation)

1. Invalid Input (Allocated size: -1)

```
Allocating -1 byte...
Invalid memory size request: size need to be greater than 0
```

2. Invalid Input (Allocated size: 0)

```
Allocating 0 byte...
Invalid memory size request: size need to be greater than 0
```

3. Edge Case: Size larger than MIN_ALLOC_UNIT: 4 pages (16384 bytes)

```
Allocating 4 pages bytes...
=====
Block Information
Block header starting address: 139966497779712
Block userspace starting address: 139966497779736
Block userspace size: 16384
Block free status: 0
=====
```

From above, we can see the size of the header `block_t` is $139966497779736 - 139966497779712 = 24$, which is exactly the same as the `sizeof(block_t)`. The free status now becomes 0 since 0 represents `STATUS_NONFREE`.

4. Double free would cause Free Failed: double free error

```
=====
Block Information
Block header starting address: 139739131781120
Block userspace starting address: 139739131781144
Block userspace size: 48
Block free status: 0
=====
Free List Debug Information
=====
Block Information
Block header starting address: 139739131781120
Block userspace starting address: 139739131781144
Block userspace size: 12264
Block free status: 1
=====

Free Failed: double free
```

5. Formal Case: 48 bytes

```
Allocating 48 pages bytes...
=====
Block Information
Block header starting address: 139700305911808
Block userspace starting address: 139700305911832
Block userspace size: 48
Block free status: 0
=====
```

6. Multiple allocations: 60 bytes, 40 bytes and freeing

```
Allocating 60 * sizeof(int)...
=====
Block Information
Block header starting address: 140107905187840
Block userspace starting address: 140107905187864
Block userspace size: 240
Block free status: 0
=====
```

```

Allocating 40 * sizeof(int)...
=====
Block Information
Block header starting address: 140107905188104
Block userspace starting address: 140107905188128
Block userspace size: 160
Block free status: 0
=====

Freeing the allocated memory...
=====
Free List Debug Information
=====
Block Information
Block header starting address: 140107905187840
Block userspace starting address: 140107905187864
Block userspace size: 240
Block free status: 1
=====
Block Information
Block header starting address: 140107905188288
Block userspace starting address: 140107905188312
Block userspace size: 11816
Block free status: 1
=====

Free List Debug Information
=====
Block Information
Block header starting address: 140107905187840
Block userspace starting address: 140107905187864
Block userspace size: 12264

```

Here we can use a figure to visualise the block information and free list (see [Figure 3](#)).

- (a) First time we allocated a 240-byte block to users which returns `ptr1`, and then we allocated a 160-byte block to users which returns `ptr2` as the figure shows. The remaining large block's header `block_t` in the free list starting from 140107905188288.
- (b) **After freeing `ptr1`**, the 240-byte block was added into the free list with its header `block_t` starting from 140107905187840, ending at $140107905187840 + \text{sizeof}(\text{block_t}) + 240 = 140107905188104$.
- (c) **After freeing `ptr2`**, the 160-byte block was added into the free list with its header `block_t` starting from 140107905188104, ending at $140107905188104 + \text{sizeof}(\text{block_t}) + 160 = 140107905188288$.
- (d) Since the current 3 blocks are continuous, they are merged into one single large block. So there is only one block left in the free list as above shows in the last "Free List Debug Information". The block starts from 140107905187840 (including header `block_t`) and lasts for 3 pages.

Additionally, if we take a close look at their free status, we can see that information about an allocated block has a "0", indicating currently not free. In the free list, however, each block contains "1" in free status, indicating currently free. This is also correct.

So we now can see `myalloc()` performs correctly and also `myfree()` with a merging function called performs correctly.

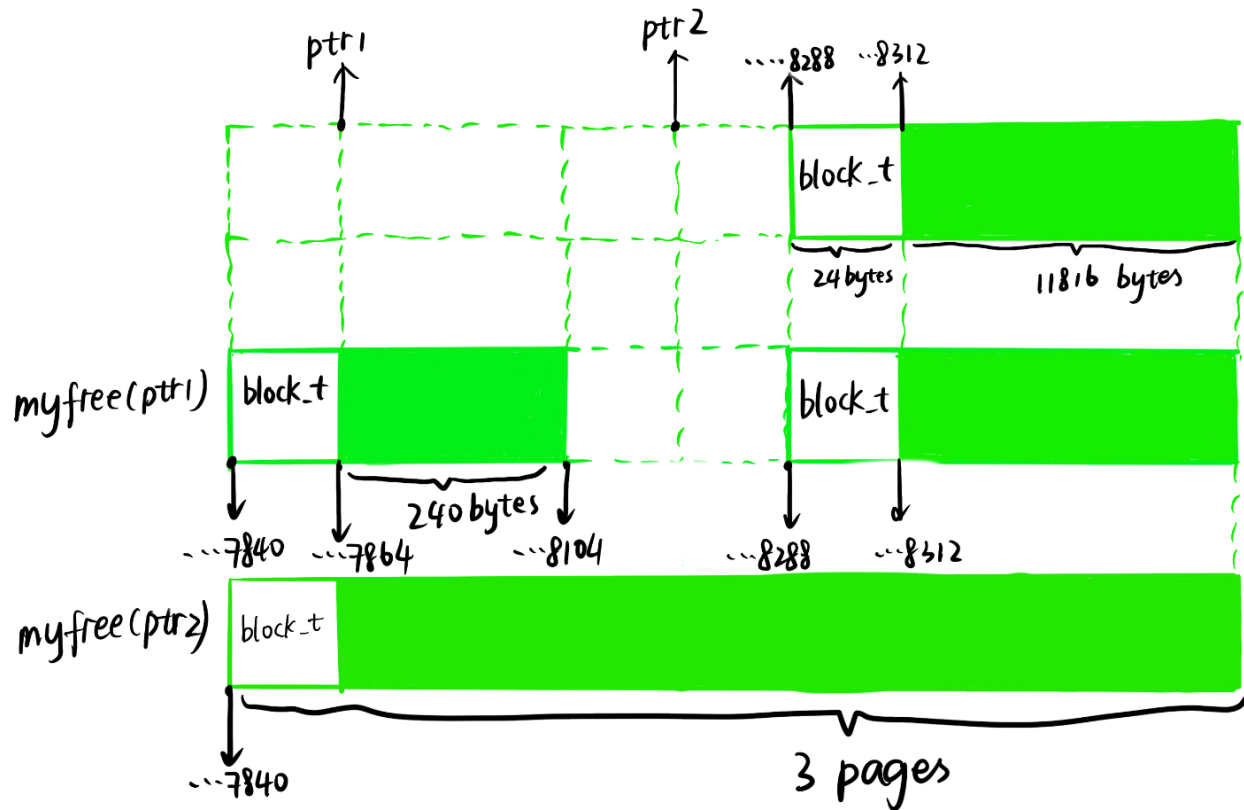


Figure 3: Free list

3.3 Self-Testing for Thread-safe Mylloc (Extension)

Thread 1 and thread 2 are created. Thread 1 and thread 2 allocate 12 blocks and then free them respectively. All blocks allocated by thread 1 have size which is less than 65, and all blocks allocated by thread 2 have size which is larger and equal to 65. Hence we can distinguish which thread is working by size of blocks allocated.

To make sure thread 1 and thread 2 can function at the same time, thread 1 need to `sleep(0.0001)` to wait for thread 2 until thread 2 is created. So each thread does two things: `myalloc()` 12 times and frees those allocated blocks.

```
void *ptrs[12];
int index = 0;
for(int i = 5; i < 65; i = i + 5){
    void *p = myalloc(i);
    ptrs[index++] = p;
}

for(int i = 0; i < index; i++){
    myfree(ptrs[i]);
}
```

Results from terminal:

```
Inside thread 1
Inside thread 2
=====
Block Information
```

```

Block header starting address: 140311115071488
Block userspace starting address: 140311115071512
Block userspace size: 5
Block free status: 0
=====
Block Information
Block header starting address: 140311115071517
Block userspace starting address: 140311115071541
Block userspace size: 10
Block free status: 0
=====
..... (Omitted)
=====
Block Information
Block header starting address: 140311115071683
Block userspace starting address: 140311115071707
Block userspace size: 30
Block free status: 0
=====
Block Information
Block header starting address: 140311115071737
Block userspace starting address: 140311115071761
Block userspace size: 35 // thread 1
Block free status: 0
=====
Block Information
Block header starting address: 140311115071796
Block userspace starting address: 140311115071820
Block userspace size: 65 // thread 2
Block free status: 0
=====
Block Information
Block header starting address: 140311115071885
Block userspace starting address: 140311115071909
Block userspace size: 70
Block free status: 0
=====

```

I added some green comments inside the output so we can clearly see what happened. After 0.0001 seconds' sleeping, thread 1 starts to request allocations, 5-byte block, 10-byte block,..., until 35-byte block. It is supposed to request a 40-byte block, while a 65-byte block is allocated as the block information shows. It indicates that now thread 2 is in charge and it starts to request blocks with size larger than or equal to 65 bytes.

Now the 35-byte blocks' header starts from 140311115071737 with size 35 requested by thread 1, indicating that it should end at $140311115071761 + 24(\text{sizeof}(\text{block_t})) + 35 = 140311115071796$. Now we look at the 65-byte block requested by thread 2: it starts from 140311115071796 !

We can now conclude that `myalloc()` is thread-safe, since when multiple threads compete with other threads, `myalloc()` ensures safety of the free list via a global mutex.

4 Evaluation & Conclusion

1. The full functionality of the standard specification satisfies requirements.

- (a) All stacschecks have been passed.
 - (b) Codes are clear and simple to follow, with sufficient comments and explanation.
 - (c) There is no duplicate method.
2. Extension perform well according to requirements from the specification sheet.
- (a) Extensions have passed self-developed testings.

In conclusion, during these 4 weeks, I have strengthened C programming and acquired understandings about memory allocation inside operating system by building a customised malloc() and free(). I also have deeper insights into all errors and faults may occur.

References

- [1] Wilson, P., Johnstone, M., Neely, M. and Boles, D. (1995). *Dynamic Storage Allocation: A Survey and Critical Review*. [online] pp.27-30. https://studres.cs.st-andrews.ac.uk/CS3104/Practicals/P1-Malloc/section3.2_pages27-30_wilson95dynamic. [Accessed 12 Oct. 2019].