HW1(a) Heap Manager

Wanning Jiang (wj43) (wjiang@cs.duke.edu)

How to test

I include a Makefile in the my_malloc folder. The Makefile will build load the my_malloc.c source code to a shared dynamic library as libmymalloc.so, and then build the executable file with the test codes. When you run the code, the result, as well as the current memory allocating strategy and test size will be printed to the standard output. For your reference, I put my own test results in result.txt.

Noted that to test the code, you need to change the path macro in Makefile to be the actual path of my_malloc folder on your machine.

Objective

Ensure correctness, achieve fast malloc time and leave small fragmentation.

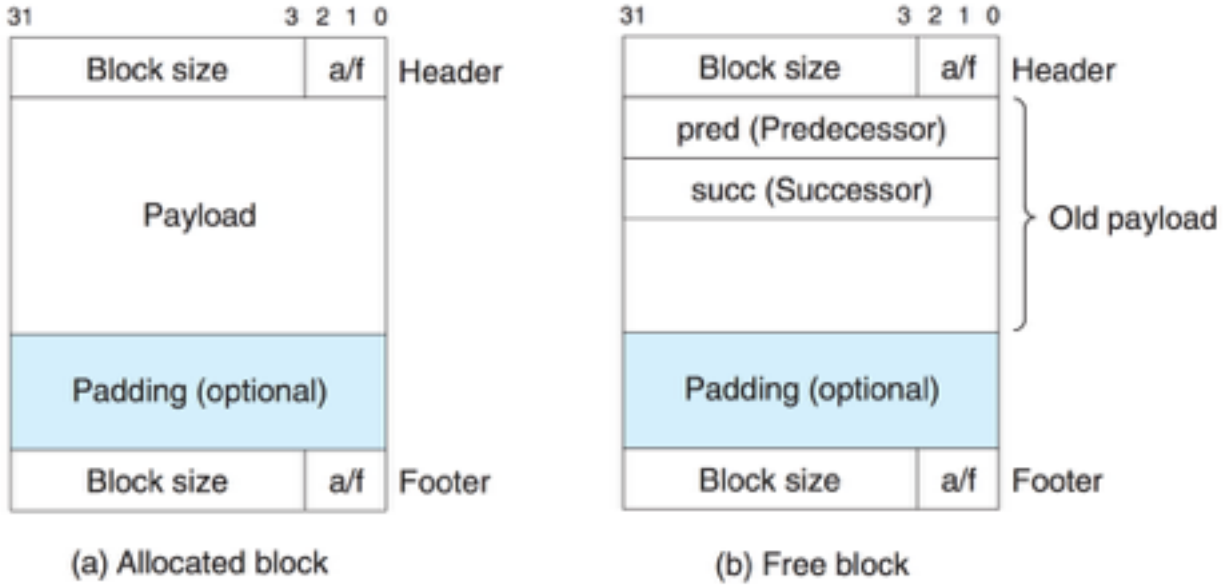
Implementation

1. Data Structure

   - Memory block

   Each memory block is pack in a uniform format as the right one of the below figure. The top of the block is a *metadata_t* structure including the block size, a bit of allocation indication, a predecessor pointer and a successor pointer. Pointers will point to other *metadata_t* objects. The bottom of the block is a *footer_t* structure including the block size and a bit of allocation indication. The blocks are aligned with 8 bytes.

   Block size is a *size_t* variable indicating the size of the real space of this block, namely, the neat size without metadata and footer. Since the block is aligned with 8 bytes, the last three bits of the *size_t* will always be *[000]* , so we utilise the last bit to indicate the boolean variable: whether the block has been allocated. The figure showing the structure is from [1].



- Linear free block list

Free blocks are linked by the *prev* and *next* pointers (to *metadata_t*) to be a doubly linked list.

- Segregated array of free block list

Segregated array of free block list is an advanced method to organise the free blocks. I arrange the blocks according to their space. The arrangement is implemented in the function *array_idx()*. The array entries are like a set of buckets and each of them accommodates free blocks with a fixed range of space, like:

$$\{1\}, \{2\}, \{3\}, \{4 - 7\}, \ldots, \{1024 - 2047\}, \{2048-4095\}, \ldots, \{\infty\}$$

Details of the advantages of doing so will be illustrated in the next section.

2. Prologue and Epilogue

The beginning of the heap is a single footer_t structure and the end of the heap is a single metadata_t structure. The two objects are all set allocated.

3. Memory allocation

To ensure malloc correctness, we assign a free block with space larger then required. The searching can be conducted in the following three policies.

- First fit

End searching upon finding the first qualified block.

- Best fit

Find the one with least free space among all the qualified blocks. In linear free block list, the time complexity is $O(n)$ where n is the number of total free blocks. In segregated array of free block list, the worst time complexity is $O(n)$ but the average time complexity is $O(\log n)$ if the size of free blocks follow a uniform distribution. The reason is that we can locate the proper entry of the array in $O(1)$ and each entry only contains $\log n$ blocks averagely.

- Worst fit

Find the one with largest free space among all the qualified blocks. The complexity is similar with best fit.

- Split

After finding the block to be allocated, I will check the space to decide whether I will split the space to allocate one part of it, add configure the other part as new smaller free block to be used by other malloc request in the future. If the left space is not enough to be packed into a block with metadata and footer, I will allocate that block as a whole. Otherwise, the new smaller free block will be packed and added to the free list.

- Other details

I pay attention to some other details during my implementation. If there is no qualified blocks in the list, I will extend the heap by calling *sbrk()* in the standard C library. After malloc, the block should be removed from the free list.

4. Memory free

The function *free()* is composed of two steps: coalesce and add to free list.

- Coalesce

The current block to be freed might be able to be coalesced with its neighbours. By neighbours, I mean the blocks continuous to the current block in the virtual address space. We can merge these two or three free blocks to compose a larger free block.

- Add to free list

We then add the coalesced free block to the proper entry of the free list array.

Result & Analysis

Testing environment: 64-bit ubuntu-12.04

1. Execution time of the test memory allocation cases

Execution time (s)	Equal size allocation	Small size allocation	Large size allocation
First fit	13.998	0.505	1.770
Best fit	13.382	0.383	1.431
Worst fit	58.036	0.748	0.987

The time of first fit and best fit is similar because we use segregated array strategy. Best fit time is slightly less then first fit in the first column, I guess that is because the best fit strategy make fuller use of the memory space so that it avoids frequent heap extension. Worst fit is obviously slower because we will search the free block from the very end of the free list array.

2. Fragmentation = free space able to be allocated / total size of memory heap

Fragmentation (%)	Equal size allocation	Small size allocation	Large size allocation
First fit	51.1%	13.6%	9.41%
Best fit	51.1%	15.4%	4.05%
Worst fit	50.4%	25.4%	24.9%

As the size of allocation goes larger, the best fit strategy perform significantly better than others. However, I will question this conclusion because the fit strategy is hard to be defined 'best' without a clear per-defined allocation policy. We can only say that generally, the best fit strategy performs better given the required block sizes are large. The risk of best fit is that there might be left small pieces of free blocks split from the block chosen by malloc function. These block are two small to be reused so they left as external fragmentation in the heap region. For the same reason, worst fit performs slightly better than the other two in equal size allocation test.

Reference

[1]. Gokhale, Maya B., and Paul S. Graham. Reconfigurable computing: Accelerating computation with field-programmable gate arrays. Springer Science & Business Media, 2006.

[2]. Some of the source code is modified from my own code in COMPSCI 310 (Operating System) lab, which is also a heap manager.