Connor Grehlinger ECE650: HW1 01/17/18

Section 1: Problem & Setup

This homework requires a DIY implementation of malloc and free. The motivation for implementing these C library functions is to gain a better understanding of memory management for processes and how memory allocation requests are implemented under-the-hood. Malloc and free are each implemented with a first-fit, as well as a best-fit policy; resulting in four major functions for this homework assignment. In both malloc implementations previously freed blocks are searched before growing the heap. In both free implementations, adjacent free blocks are coalesced to form larger contiguous blocks. These features enforce a minimization policy of the processes heap segment.

Code for this assignment was written in C and developed in the Duke Linux environment at login.oit.duke.edu. Various tests are included in this homework assignment to verify the correctness of each malloc and free implementation. A performance analysis is also conducted. The performance analysis measures execution time and fragmentation, and is followed by a discussion of the results.

The subsequent sections of this report are structured as follows. Section 2 details how each malloc and free function was implemented and describes any helper functions used. Section 3 discusses the performance metrics of each malloc and free implementation and analyzes the results of three different benchmarks.

Section 2: Implementation

Memory management data structure:

The data structure used for managing allocated memory is a doubly-linked list. The nodes in the list are block_node structs which contain the following meta data: size (size of data payload plus the block_node struct), next (address of next block_node), and prev (address of the previous block_node):

```
typedef struct block_node_t {
        size_t size;
        struct block_node_t * next;
        struct block_node_t * prev;
} block_node;
```

A state variable for the blocks used/free status is not included. This is because the list consists of only free blocks. Used block addresses are returned by malloc and are the argument to free calls, so there is no need to track used blocks (that falls on the user). Tracking only free blocks greatly improves performance as it reduces the number of blocks searched on a malloc call.

The doubly linked list of block_nodes is sorted by ascending address. This ensures that the next and previous free block pointers in a free block's block_node struct point to the closest neighboring free blocks in memory. This makes for easy free block coalescing, since blocks can compare their next and previous pointers to their adjacent memory blocks. For example, if you have a block_node * current_block, and [(block_node *)((char *)current_block + current_block->size)] == [current_block->next] evaluates to true, then you know the current block and the next free block are adjacent in memory. These blocks are then coalesced to form one larger, contiguous block.

First-Fit: ff_malloc & ff_free

A first-fit policy dictates that when searching for a new block of memory to fulfil a memory allocation request, the first block able to accommodate the space for that request is used. This is relatively simple to implement but clearly is not the most efficient use of memory (as opposed to best-fit, which will be discussed next).

The malloc function for first-fit policy (ff_malloc) was implemented with the help of several helper functions:

block_node * try_block_reuse(size_t size);

- This function searches the list of free blocks and returns a pointer to the first block found whose size is greater than or equal to the size parameter. If no block is found, NULL is returned.

void attempt_split(block_node * to_split, size_t size_needed);

- This function splits the block found for re-use (for an allocation request of size_needed bytes) and creates a new block_node from the remaining size (to_split->size – size_needed) if the remaining size of the block is greater than or equal to a MIN_SIZE constant defined in my_malloc.c
- The MIN_SIZE constant can be tuned based on the workload for malloc but must be at least sizeof(block_node) (i.e. 24 bytes) to avoid error (otherwise the attempt_split function could create a new block_node in a memory location which could over-write another block or data). Increasing the MIN_SIZE will also increase the over-all size of the running process's heap. This is because the size blocks must be in order to split increases, and blocks with larger sizes may be selected for smaller allocations, resulting in unused space. Complete minimization of heap is done when MIN_SIZE == sizeof(block_node).

void remove_from_free_list(block_node * to_remove);

- This function is called if the malloc function successfully finds a block for re-use. This function is responsible for removing that block from the list of free blocks.

block_node * grow_heap(size_t size);

- This function increases the size of the processes heap by size bytes with the sbrk system call. Grow_heap is only called when no free blocks exist which can satisfy the request. A pointer to the newly allocated block is returned. If the sbrk call fails, NULL is returned.

Together these helper functions form ff_malloc. A size is passed in as the single argument to ff_malloc. The size of a block_node struct is added to the size argument, and a block_node * called target_block is initialized to NULL. The malloc function then checks if it's the first time it's been called. If true, the grow_heap function is called, and its result is assigned to target_block which is then returned with an offset to the data payload (the address at the end of the block_node struct). If false, try_block_reuse is called. The try_block_reuse function searches the free list for a block of adequate size. If a block is found, attempt_split is called from try_block_reuse, which will split the block if possible based on the MIN_SIZE parameter. The result of try_block_reuse is assigned to target_block. If target_block == NULL at this point, then no blocks have been found for reuse, and target_block is assigned to the result of a grow_heap call and returned. If target_block is non-NULL, a valid block has been found for re-use, and remove_from_free_list is called to remove that block from the list of free blocks before returning target_block with an offset to the data payload.

The free function for first fit policy (ff_free) was implemented with the following helper functions:

void add_to_free_list(block_node * to_add);

- This adds a block to the list of free blocks using sorted insert by the block's address (ascending order).

void coalesce(block_node * free_block);

- This function checks if the next and prev pointers in free_block point to free blocks which are adjacent in memory and combines the blocks into a single contiguous block if true. This function may perform up to 2 coalesces per call (one for each block_node * in the block_node struct of free_block), and calls remove_from_free_list for each block it is able to coalesce.

These functions comprise ff_free. A void * is passed in as the single argument. The block_node* is found via pointer offset and add_to_free_list is called to add the block to the free list, making it available for reuse. The coalesce function is then called to coalesce the freed block if possible.

Best-Fit: bf_malloc & bf_free

A best-fit policy dictates that when searching for a new block of memory to fulfil a memory allocation request, the block which has the smallest difference in size relative to the allocation request size is used. This will use space in memory more efficiently, but requires additional searching of the memory management data structure to implement (you must search the entire linked list).

The malloc function for best-fit policy (bf_malloc) was implemented in exactly the same manner as the first-fit malloc. The only modification is to the function which is called to search for a block to re-use if it is not the first memory allocation (called "try_block_reuse_bf").

block_node * try_block_reuse_bf(size_t size);

- This function searches the entire list of free blocks and returns a block_node * to a block whose size is greater than the size argument of the function and has the smallest difference in size relative to that size argument. For example, if the free list consists of blocks with sizes 16, 8, 24, 12, 32, and a bf_malloc request is made for a size totaling in 10 bytes, then try_block_reuse_bf would return a pointer to the block_node with size 12, as 12-10 is the smallest difference.
- If a block is found for reuse, the attempt_split function is called to attempt to split the block then a pointer to that block is returned. If no block is found for reuse, NULL is returned.

The free function for best fit policy (bf_free) is exactly the same function as ff_free but with a different declaration. Both functions add the freed block to the free list, and call the coalesce function to attempt to coalesce that block.

Section 3: Performance Results & Analysis

To determine how well each version of malloc and free perform, both versions were used in three different benchmarks. The equal_size_allocs benchmark streams through memory malloc'ing blocks of equal size (128 bytes) and freeing them in the same order they were malloc'd. The small_range_rand_allocs benchmark uses allocations of random sizes ranging from 128 to 512 bytes in 32 byte increments and alternates freeing 50 random regions then mallocs 50 more. The large_range_rand_allocs benchmark is similar to the small_range_rand_allocs benchmark but uses sizes ranging from 32 bytes to 64 kilobytes. Results from these benchmarks for first fit and best fit are presented in tables below. The execution time shown for each benchmark is the typical time observed, though times can vary from run to run.

First-Fit: ff_malloc & ff_free

	Execution time (secs)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
Equal size	17.701	3040000	1368000	0.450000	100020000	0	19999
Small range	16.164	3702832	278384	0.075181	1010000	521628	532201
Large range	121.313	359581824	33592960	0.093422	510000	492699	503270

NUM_ITERS not modified for any benchmark

Best-Fit: bf_malloc & bf_free

	Execution time (sec)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
Equal size	1109.291	3040000	1368000	0.450000	10020000	0	19999
Small range	9.967	3525928	98104	0.027824	1010000	198931	209141
Large range	158.090	339757800	13764288	0.040512	510000	415406	425632

NUM_ITERS modified for equal_size_allocs (set to 1000 from previous 10000)
Unmodified in small_range_rand_allocs and large_range_rand_allocs

Equal_size_allocs
The ff_malloc and ff_free functions performed much better on the equal_size_allocs benchmark. This is to be expected, since the try_block_reuse function for ff_malloc is able to return the first available block of adequate size. Because all allocations are the same size in this benchmark, the first free block checked (the head of the free list) will always be returned if it exists. Whereas the try_block_reuse_bf function used by bf_malloc will traverse the entire free list for each malloc request, and only return when it reaches the end of the list. The bf_malloc function is particularly ill-suited for this benchmark since all blocks are the same size (thus, there is no "best fit"). When the NUM_ITERS variable is unchanged, the equal_size_allocs benchmark for bf_malloc does not finish in a reasonable time (I've yet to wait long enough). If the target work load requires many allocations of the same size, then clearly ff_malloc and ff_free are the better choice.

Small_range_rand_allocs
The bf_malloc and bf_free functions outperformed the ff_malloc and ff_free functions in this benchmark in terms of execution time, fragmentation and amount of memory used. This can likely be attributed to the roughly 62% decrease in block split operations and 61% decrease in coalesce operations. This reduction occurs because choosing a better fitting block reduces the need to split blocks, and splitting less blocks reduces the amount available to be coalesced. These operations are somewhat costly as splitting a block requires creating a new block_node struct, modifying its fields, and adding it to the free list. Coalescing blocks similarly requires creating one or more block_node structs, evaluating multiple conditional statements, modifying struct fields, and removing blocks from the free list. These instructions add up in a large workload.

Large_range_rand_allocs
The ff_malloc and ff_free functions have better execution time on this benchmark (by around 35 seconds on average) but the bf_malloc and bf_free functions experience less than half the amount of fragmentation. This is consistent with the expected behavior of these functions when working with random allocation sizes. Best fit uses memory more efficiently, but at the expense of more free list searching. First fit performs faster, but at the expense of not using the most efficient block for a memory allocation.

Modifying MIN_SIZE:
This implementation of malloc and free can be tuned slightly based on workload needs. This is done by adjusting the MIN_SIZE parameter, which is used to determine whether or not a block is large enough to be split. If a block is to be split, the remaining memory must be large enough to hold at least the meta data (block_node struct) for the newly created block. This is so the remaining free space can be kept track of, and so that the newly created block_node struct is within the free region and doesn't over-write allocated data. However, though this minimizes the use of process memory (since an sbrk call is avoided), the newly created block is not actually able to hold anything and won't be useful to the program until it is coalesced.

In fact, if speed is the priority, then having the MIN_SIZE set to the size of the block meta data is detrimental, since it creates another node in the free list (which is searched on each malloc request) and the added node can't even hold anything! This led to some experimentation with that parameter.

Best fit policy with the large_range_rand_allocs benchmark was used for the below results:

If MIN_SIZE is set to the size of meta data + 8 bytes:

Execution time (secs)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
155.532	340045824	14049880	0.041318	510000	411703	421934

If MIN_SIZE is set to the size of meta data + 64 bytes:

Execution time (secs)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
145.527	339889240	13844080	0.040731	510000	358668	368896

If MIN_SIZE is set to the size of meta data + 128 bytes:

Execution time (secs)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
134.498	339691688	13563632	0.039929	510000	321506	331732

If MIN_SIZE is set to the size of meta data + 512 bytes:

Execution time (secs)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
60.047	340383664	13524880	0.039734	510000	205860	216097

If MIN_SIZE is set to the size of meta data + 4096 bytes:

Execution time (secs)	Data segment size (B)	Data segment free space (B)	Fragmentation	Number of allocations	Number of splits	Number of coalesces
2.520	344396456	10842624	0.031483	510000	39263	49565

Increasing the size needed for a block to be split (MIN_SIZE) has a significant performance increase in the context of execution time. This is due to the fact that splitting less blocks reduces the size of the free list which must be searched. Additionally, fragmentation and data segment size can be brought down in some cases where a MIN_SIZE is set to a threshold of a "typically sized" block; meta data size + 128 bytes seems to perform well. However, in other cases fragmentation only improves because the data segment size has grown (it hasn't been minimized). The MIN_SIZE constant can therefore be tuned to favor speed at the expensive of memory and fragmentation, memory and fragmentation at the expense of speed, or ideally try to strike a good balance of the two performance metrics.