COMP2310 – Assignment 1 Report (Akshat Jain, u7284072)

My memory allocator implementation included key features including fenceposts, metadata reduction, constant time coalescing with boundary tags, multiple free lists, getting additional chunks from the OS and optimizing large allocation requests by using the mmap-based allocator.

Foremost, the structure of free blocks and allocate blocks are described in Figure 1.1 and 1.2. Notably, each metadata block (struct MetaBlock) which indicates the size and allocation status of a block is of size `sizeof(size_t)` and the pointer block (struct PointerBlock) is twice this as it stores two pointers. This means the maximum amount of meta-data stored in a block is `4 × sizeof(size_t)`.

The Free-List root is an array of free-lists represented by the variable "freeListArray". Each bin in the array represents a size range with the minimum being for sizes $<2^6$ (=64) and the maximum bin being for mmap allocations larger than 2^{12} (=4096). The bins are separated by powers of 2 within this range which results in the array size of 8. Figure 2.1 shows the precise boundaries for clarity.

To initially allocate some memory to manage, first the method "initialise(int m)" is called where "m" represents the multiple of ARENA_SIZE the program would like to request. "m" is calculated by simple division by ARENA_SIZE. Once mmap is called, fenceposts, represented by single MetaBlocks on each end of the allocated memory. These blocks have the dummy "size" value of 64MB, which cannot be allocated by the allocator itself, which makes these fence posts easily identifiable. Then, the actual boundary tags are set adjacent to these fence posts, each with the appropriate size of the total allocated memory minus the size of the two MetaBlocks used for the fence posts. The pointers of the initial free block are set to NULL.

When a size request is made to "my_malloc", assuming it passes the size criteria, first the appropriate array index is calculated using the "getIndex()" function. the freeListArray at this index is checked, and if no free list with the appropriate size is initialised, the above process is done. We note that for large allocations, (i.e., More than 4MB), mmap in initialise is **always** called and stored in the last index. This represents the "optimising large allocations" task. The function then employs a first-fit policy. In the case that this process returns null, then another block is requested from memory with the same process and inserted to the start of the free list. This represents the "getting additional chunks" feature. If the found block is more than twice the needed size, the block is split into two using "splitBlock()".

Now, the block can be returned to the user, so it needs to be removed from the free list. In the case that it is **not** the root of the relevant free list, then the next and previous blocks in the free list simply point to the second block in the split (if it exists), or each other. In the case that the block being returned **is** the root of a free-list, then again, either the second block from the split is made the root, or the next element in the free-list is made the root. In the case that neither the split block exists and there is no "next" element, then there is again newly allocated memory which becomes the new root of the free list. Finally, for both the header and footer boundary tags of the returned block, 1 is added to the size parameter to set the last bit in boundary tags to indicate the block is allocated. The block is then cleared and returned to the user.

When freeing, the pointer is first checked to not be null, or allocated, and the free list is checked for not being all NULL, so as to verify the free request is valid. Then, using the metadata block, the free-list index is calculated, and the allocated bit is cleared from both the header and footer. The bulk of the logic is then done in the "coalesce()" function. Here, both the **physically** left and right neighbours are checked to see if they are in the free list. If they are, their pointers are stored, and their size is recorded in the "newSize" variable. If the left neighbour is free, a root variable is also updated as that will become the new left MetaBlock point of the block being freed. Then, the 4 cases of coalescing are handled. If both the left and right blocks were not a part of the free list, then the block being freed is simply added to the start of the free list. If only the left block was free, then the size of this block in the free list is updated, as well as the position of the pointers within that block. If only the right block was free, then the same instructions as the previous case are done, however, the next and previous elements in the free list need to be updated so that they point to the new correct root. Finally, if both the left and right blocks are free, the right block is deleted from the free list by updating its specific next and previous block's pointers. The size and pointer of only the left block is then updated, as now it includes the deleted right block.

A key challenge to the task was ensuring that desired free list properties were maintained throughout the entire process, especially without the use of a visual debugger. A small example is ensuring that each free list root's "prev" pointer was always NULL, or that the result of the getPointers() function was always larger than the input. Many of the functions relied on such assumptions being always true and hence, it was crucial that this integrity was maintained. To enforce this, I regularly used many "assert" statements to check these properties giving me confidence that the properties held.

Another challenge I had was dealing with non-deterministic edge cases in my logic. For example, in my initial implementations, I often employed shortcuts in pointer operations by direct arithmetic on the pointer values, at some key points during a function. However, this would not always work depending on the pointer value giving occasional faults which could not be predicted. To fix this, I made an active effort to decrease "hardcoding" as much arithmetic as possible, but rather rely on a more axiomatic approach where only operations known to produce predictable outputs were used.

A key observation that I got from testing and benchmarking is the importance of pointer arithmetic order. Whilst logically, many consecutive pointer operations seemed to be order independent, in practice they were not, causing incorrect operations and commonly Segmentation Faults. For example, in many cases I would use the size parameter to traverse the block with pointer arithmetic, then update the block's metadata. However, the very fact that the size parameter was "outdated" relative to the operation I was doing, meant this operation was wrongly ordered causing faults. Hence, much care and testing were done to ensure that pointer operations were done in a consistent order.

Another aspect I learnt is the power of coalescing and specifically which coalescing policies had the most impact. Prior to having implemented constant time coalescing, my benchmark would run at around 60s. After implementing coalescing, I average roughly 0.2s. Additionally, as I implemented each of the four coalescing cases, I noticed that cases where there was **one** adjacent free block had the most impact in practice. Hence, in running the tests and benchmark, I was able to see how and why coalescing was impactful.

Appendix:

Figure 1.1 – Free Block Structure

Figure 1.2 – Free Block Structure

Figure 2.1 – FreeListArray Bin Sizes

FreeListArray Index	0	1	2	3	4	5	6	7
Stored Block Sizes (Bytes)	< 64	[64,128)	[128,256)	[256,512)	[512,1024)	[1024, 2048)	[2048,4096)	4096 ≤