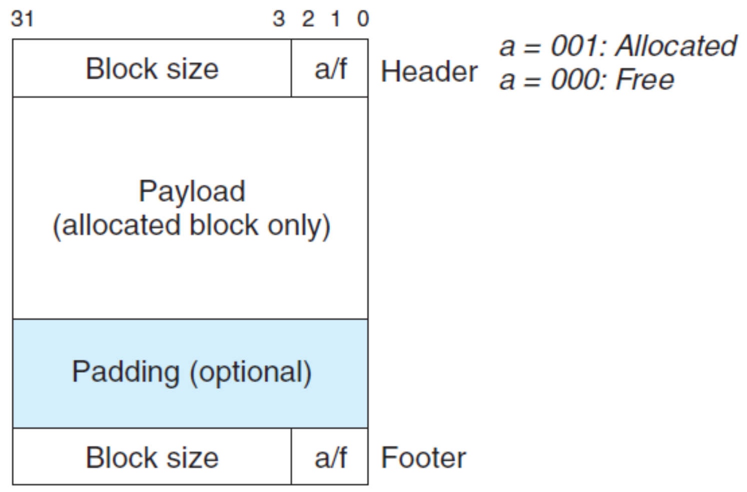
CS 310 Lab 1: Heap Manager   Due: 9/11/2014. Submitted: 9/12/2014   Talal Javed Qadri

I used the following references: course lectures and Chapter 9 from Computer Systems: A Programmer's Perspective by Randal E. Bryant and David R. O'Hallaron. I utilized the idea of boundary tags described in the above mentioned Chapter 9. The figure below is taken from Chapter 9. The heap block in my heap manager are accurately represented by this figure.

Figure 9.39
Format of heap block that uses a boundary tag.



The only difference in my implementation is that the header and the footer of a block are 8 bytes each since they are of size_t. The header and the footer contain the size of their heap block (which includes the size of the header, the footer, the payload, and the alignment padding). Since, my heap is aligned on an 8 byte boundary; the lower 3 bits of the size of a heap block will always be zero. Hence, these bits can be used for other purposes. I used the lowest bit to determine whether a block was allocated or free. If the bit was 0 then the block was free and vice versa. I also used prologue and epilogue blocks in my heap to avoid running into errors at the edges of the heap by trying to allocate beyond the heap boundaries.

The minimum size of a heap block was 24 bytes. 16 bytes were needed for the header and the footer whereas a minimum of 8 bytes were needed for the payload (+ padding) to maintain the longword alignment. I did not use the provided metadata structure. If I had used that structure, I would have had an overhead per block of 24 bytes (2 pointers of 8 bytes each, and a size_t type to store block size) compared to the 16 byte overhead of my implementation (a header and a footer of 8 bytes each). Hence, my implementation had less memory overhead compared to the metadata structure.

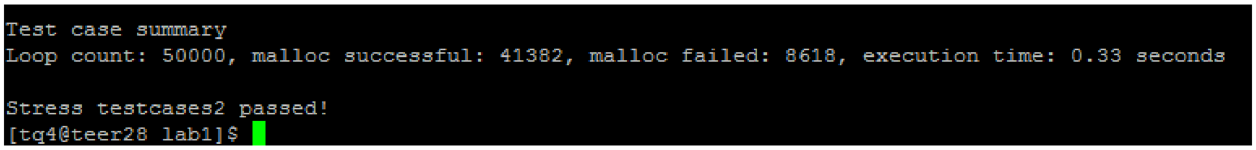
Although, the footer increased overhead, I felt it was worth it since it allowed me to find the address of the previous block in constant time which allowed for the implementation of free operations and coalesce operations (**extra credit**) in constant O(1) time. I found the macros in Chapter 9 to be very useful and incorporated them in my code after appropriately modified them for my alignment constraints.

I initially tried to keep an explicit free list, so that dmalloc() would take O(#free), by storing next and previous pointers in a free heap blocks. This implementation increased the size of my minimum block to 32 bytes (16 bytes for 2 pointers, and 16 bytes for header and footer). Although I was able to pass test_stress1, test_basic, and test_coalesce with this implementation, I kept running into segmentation faults for test_stress2 and test_stress3. Therefore, I reverted to an implicit free list where dmalloc() can potentially take O(#heap blocks). This slows down dmalloc() if a number of smaller blocks are allocated.

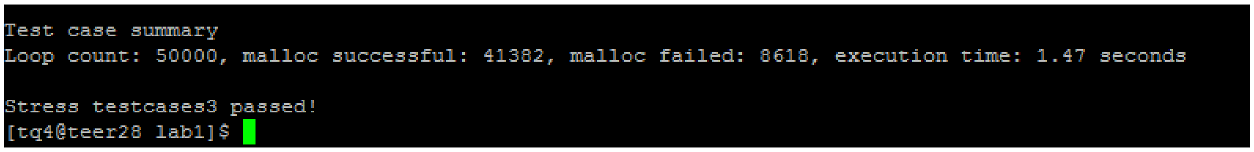
However, I spent 12+ hours trying to fix the explicit free list implementation without luck so I used the implicit free list implementation reluctantly.

I spent around 28 hours on this lab. Most of my time was spent debugging the explicit free list pointer implementation (which I eventually abandoned). I am submitting this lab 1 day late but I did spend a lot of time on it. I would have been able to submit it on time if I had not run into the explicit free list roadblock. The results for running the tests are presented below. Note, that although I free and coalesce in O(1) time, the time for test_o1 increases as n (number of blocks) increases. The reason for this was pointed out on Piazza by a student. Dmalloc() is called in the timed segments. The time would only stay constant if dmalloc() was O(1) which is not a requirement for the extra credit. Furthermore, with an implicit free list and many dmalloc() calls, the time for dmalloc() increases for my implementation leading to timing differences in test_o1 for different number of blocks.

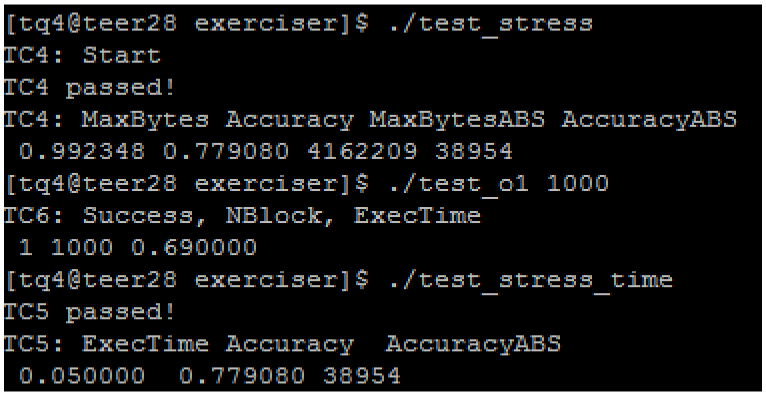
Overall, I felt this lab was a good exercise in dealing with pointers in C and in also understanding how the heap operates under the hood.


```
Test case summary
Loop count: 50000, malloc successful: 41382, malloc failed: 8618, execution time: 0.33 seconds

Stress testcases2 passed!
[tq4@teer28 lab1]$
```
Figure A: Results for test_stress2.c


```
Test case summary
Loop count: 50000, malloc successful: 41382, malloc failed: 8618, execution time: 1.47 seconds

Stress testcases3 passed!
[tq4@teer28 lab1]$
```
Figure B: Results for test_stress3.c


```
[tq4@teer28 exerciser]$ ./test_stress
TC4: Start
TC4 passed!
TC4: MaxBytes Accuracy MaxBytesABS AccuracyABS
 0.992348 0.779080 4162209 38954
[tq4@teer28 exerciser]$ ./test_o1 1000
TC6: Success, NBlock, ExecTime
 1 1000 0.690000
[tq4@teer28 exerciser]$ ./test_stress_time
TC5 passed!
TC5: ExecTime Accuracy  AccuracyABS
 0.050000  0.779080 38954
```
Figure C: Results for test_stress.c, test_o1.c (with 1000 blocks) and test_stress_time.c