Project 1 Report

Implementation Description

Data Structure for the list of free memory blocks

```
typedef struct header_tag {
  size_t size;
  char status;
  struct header_tag * next;
  struct header_tag * prev;
} header;

static header FLL = {0, 'H', 0, 0}
```

The header type stores the metadata for the memory block and implements a linked list to keep track of the list of free memory blocks . A dummy head (FLL) is initialized to keep track of the list of free memory blocks. The header type stores the size of the memory block (including the header size), and the status of the memory block to indicate whether it is free or allocated. Furthermore, it stores the pointer to the next and previous header to keep track of the list of free memory blocks.

Malloc

For the malloc with first fit policy, we search through the list of free memory blocks and find the first block that fits the requested size. If no suitable block is found in the list, we use sbrk() to increase the process data segment by the requested size. If a suitable block is found in the list, we determine if it needs to be splitted by comparing the block size and requested size. If the requested size is smaller than the block size by the amount of our determined MIN_BLK_SIZE, we will split the block and keep the remaining free memory block in the list. If we determine the memory block does not need to be splitted, we allocate the entire block and remove it from the list of free memory blocks.

For the malloc with first fit policy, we basically go through the same operation except on how we search for the suitable memory block. We search through the list of free memory blocks and keep track of the best memory block on the way. If we found a memory block size that fits perfectly with the requested size, we will break the loop and stop searching.

Free

To free a memory block, we add it to our list of free memory blocks. We add the block to the list such that the list is ordered by ascending address value. Storing the free memory blocks in this fashion benefits us when we coalesce the adjacent memory block. We coalesce the free blocks when a block of memory is freed and we check the next and previous free memory block in the list to see if their address is contiguous. If the freed blocks addresses are contiguous, we merge the free blocks by increasing the size of the first block (address ordered) and removing the second block.

Implementation Improvement

Initially, I implemented the free feature by adding every freed memory block to the head of the list. However, I found that the execution time of merging/coalescing blocks are taking too long since the list is not ordered in address, which does not utilize the caching system when we are reading the blocks. Therefore, I changed the implementation such that the list is ordered in their addresses. This greatly reduced the execution time.

Furthermore, to improve the execution time of running the “equal_size_allocs.c”. I modified my best fit algorithm such that it stops searching the best fit once a memory block size that fits perfectly with the requested size is found. This modification greatly reduced the execution time of the test code from 1 hour to 25 seconds.

Analysis

Test	First fit		Best fit	
	Execution time	Fragmentation	Execution time	Fragmentation
Equal	17.42 s	0.45	17.94	0.45
Small	5.60 s	0.06	1.32	0.02
Large	36.09 s	0.09	42.34	0.04

For the “equal_size_allocs.c” test case, the execution time and fragmentation of both first fit and best fit practically the same value. This occurred because the mallocs are all the same size, which allows best fit to find the perfect fit within the first search. This property is the exact same policy for the first search. Therefore, the two policies will have the same test results.

Besides the “equal_size_allocs.c” test case, we can observe that the fragmentation for the first fit is always larger than the best fit. This result shows that the best fit efficiently uses the

memory resource such that not much memory is “wasted” when splitting the memory block. In contrast, first fit uses the first free block disregarding its size relative to the request, which would “waste” more memory when splitting.

For the “small_range_rand_allocs.c” test case, the execution time is higher for the first fit. I believe that this long execution time is caused by calling more sbrk() using the first fit policy. For the “large_range_rand_allocs.c” test case, the execution time is higher for the best fit. I believe that this long execution time is caused by long iterations through the list of free memory blocks to find the best fit.

In real life, we allocate memory resources with a diverse range and I think the “large_range_rand_allocs.c” depicts the real life situation. As a result, I would think that the first fit policy is more efficient in the run time. The one big disadvantage is that it causes almost double the memory lost as compared to the best fit policy.