Rijish Ganguly
rg239

Malloc Implementation Report

The objective of this assignment was the implementation of **malloc()** and **free()** using two different algorithms to search for free blocks**.** Apart from implementation of these two functions using two different approaches, we studied the performance of different allocation policies. The metrics that we analyzed for our different malloc and free functions were:

1. The run-time of the program
2. Fragmentation (i.e. the amount of allocated data segment space divided by total data segment space)

Fragmentation occurs when a user program has allocated memory but doesn't use it. If machines had unlimited memory, fragmentation wouldn't have had much significance, but since memory is limited and processes compete for memory, fragmentation is bad and hence considered to be a waste of memory. In general, we want to minimize the fragmentation.

For my implementations, I used a doubly-linked list to keep track of the memory blocks. The head of the list is initially initialized to NULL.

First-Fit Algorithm

The algorithm is as follow:

We start at the beginning of the list of blocks. We sequence through the list and we stop when we reach the block that is big enough i.e. greater than or equal to the sum of the size requested by the malloc function and size of metadata.

Implementation

```
block find_free_block_FF(block * last, size_t size){

block curr = head;

while(curr && !(curr->free && curr->size >= (size + BLOCK_SIZE))){

        *last = curr;

         curr = curr->next;

        }

        return curr;

}
```

We can observe, that we keep iterating through the list as long as the block doesn’t meet our requirements i.e. curr block has to be free and the size of curr block has to be greater than the sum of the size of metadata and requested malloc size. The advantage to this approach is that apart from finding our desired block, we can also determine the last block which can be later used to assign new space using sbrk(). We break out of the loop when we find the first block that fits our requirements or if we don’t find a block at all that serves our purpose.

Best-Fit Algorithm

Performance of the best-fit algorithm for memory allocation:

The algorithm is as follow:

We start at the beginning of the list of blocks. We sequence through the list using a few conditions in the while loop. We have a current block used for iteration and a best_block which is the required desired block. The current block is initially set to point to the head of our linked list data structure. As long as the current block is free and the size of the current block is greater than the sum of the size requested by malloc and the size of the metadata and either the best_block is null or the size of the current block is less than the sum of the metadata size and size of the best_block, we iterate through the loop. As we iterate through the loop, we keep on updating our best_block. We optimize the iteration by having an if condition in which we break out of the loop if the size of the best_block is exactly equal to the sum of requested size and size of metadata. We then iterate through the linked list once more to set the last block which is utilized for growing the heap.

<u>Implementation</u>

```
block find_best_fit_block_BF(block * last, size_t size){

        block best_block = NULL;

        block current = head;

while(current != NULL) {

                                if ((current->free && (current->size >= size + BLOCK_SIZE)) &&
(best_block == NULL || current->size < BLOCK_SIZE + best_block->size)) {

                               best_block = current;

                                if (best_block->size == size + BLOCK_SIZE)

                                  break;

        }

        current = current->next;

}

current = head;

while(current != NULL){

if (((best_block - current) == 0))

                               return current;

                               *last = current;

                               current = current->next;

        }

        return current; }
```

Performance Analysis

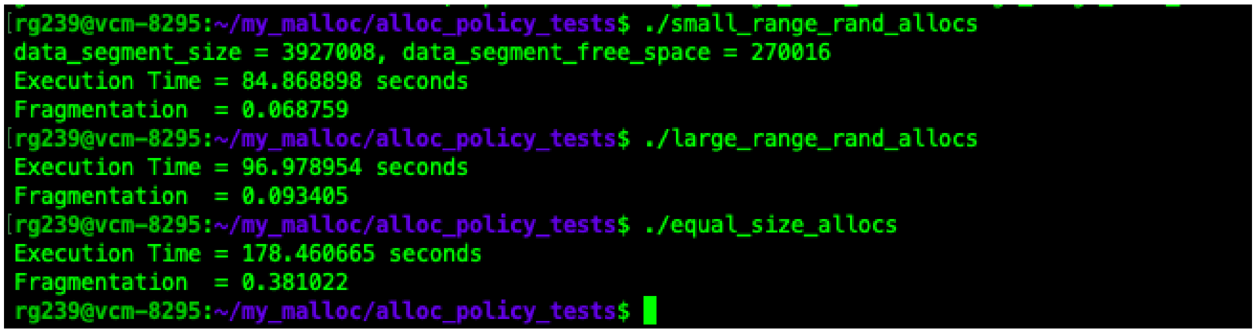
A) First-Fit Algorithm

- **Algorithmic Analysis**

  1. Best run-time for First-Fit Algorithm: O(1)It's the case where the first block is large enough.
  2. Worst run-time for First-Fit Algorithm: O(K) where K is the size of the Linked List Data Structure.

- **Fragmentation**

  Performance of first-fit algorithm for memory allocation



```
[rg239@vcm-8295:~/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
 data_segment_size = 3927008, data_segment_free_space = 270016
 Execution Time = 84.868898 seconds
 Fragmentation  = 0.068759
[rg239@vcm-8295:~/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
 Execution Time = 96.978954 seconds
 Fragmentation  = 0.093405
[rg239@vcm-8295:~/my_malloc/alloc_policy_tests$ ./equal_size_allocs
 Execution Time = 178.460665 seconds
 Fragmentation  = 0.381022
 rg239@vcm-8295:~/my_malloc/alloc_policy_tests$
```

We can observe that the algorithm is relatively fast. Fragmentation is negligible for small_range_rand_allocs and large_range_rand_allocs. However, fragmentation becomes pretty significant ~ 38% for equal_size_allocs. The data shown for equal_size_allocs is for 100 iterations.

B) Best-Fit Algorithm

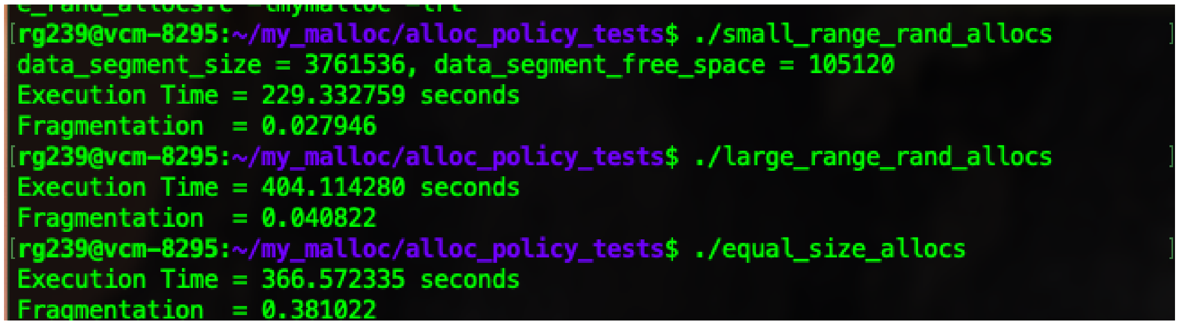
- **Algorithmic Analysis**

  1. Best run-time for Best-Fit Algorithm: O(K) where K is the size of the list
  2. Worst run-time for First-Fit Algorithm: O(K) where K is the size of the list

The performance remains the same for both best and worst case, as we have to iterate through the entire list for the algorithm in order to determine the best block which meets our purpose. The algorithm is much slower compared to first-fit algorithm.

- **<u>Fragmentation</u>**

  Performance of best-fit algorithm for memory allocation



```
[rg239@vcm-8295:~/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs      ]
 data_segment_size = 3761536, data_segment_free_space = 105120
 Execution Time = 229.332759 seconds
 Fragmentation  = 0.027946
[rg239@vcm-8295:~/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs      ]
 Execution Time = 404.114280 seconds
 Fragmentation  = 0.040822
[rg239@vcm-8295:~/my_malloc/alloc_policy_tests$ ./equal_size_allocs            ]
 Execution Time = 366.572335 seconds
 Fragmentation  = 0.381022
```

We can observe that the algorithm runs much slower compared to the first-fit algorithm. For small_range_rand_allocs and large_range_rand_allocs the fragmentation is slightly less compared to the first-fit algorithm albeit still negligible. However, for the equal_size_allocs we observe that the fragmentation increased dramatically and is identical to the first-fit case. The data for equal_size_alloc is for 100 iterations.

<u>Conclusion</u>

If we compare the performance of the two memory allocation algorithms, we can observe that although Best-Fit algorithm gives better performance i.e. less fragmentation, the fragmentation is almost comparable to the fragmentation performance given by the First-Fit algorithm. Hence, the Best-Fit algorithm doesn't reduce fragmentation significantly. Moreover, the fragmentation rate is identical for the equal_size_alloc case. However, the run time for Best-Fit algorithm is significantly more compared to First-Fit algorithm. Hence, in my opinion, the first-fit algorithm is a better memory allocation policy. Thie equal_size_alloc program uses the same number of bytes (128) in all of its malloc calls. The program equal size alloc streams through memory by mallocing 128B chunks, and then later freeing the chunks. As the allocation is of equal size and freeing is done after mallocing, this explains the comparable performance for both BF and FF algorithms. The small_range_rand_alloc program works with allocations of random size, ranging from 128 - 512 bytes (in 32B increments). The program first malloc's large number of these random regions. Then the program alternates freeing a random selection of 50 of these allocated regions, and mallocing 50 more regions with a random size from 128 - 512 bytes. The large_range_rand_alloc works in the same manner as small_range_rand_alloc however the size of the allocations vary from 32-64K bytes. As the size of allocation for large_range_rand_alloc is more than small_range_rand_alloc , it takes more time to find the suitable block as the program executes.