Assignment #1: Malloc Library

ECE650 Spring2016

Guodong Hu

Part 1: Study of Memory Allocation Policies

Overview:

To implement malloc, sbrk() system call is used to increase the heap size and return a pointer to the start of the new data region. The size of the new allocated space is stored in the meta information for that block. So a struct block is created to hold the meta information and a doubly linkedlist data structure is used to represent the list of allocated memory region and track the state of the block in the list. Also, a void *first_block is declared to represent the first block in the linkedlist. The struct block will be something like:

```
struct block {
        size_t size;
        struct block* prev;
        struct block* next;
        long free;
};
```

The free variable here is used to mark the block to help with future malloc and free operation. The long type here is to make the block 8-byte aligned, which is a consideration for performance. Note that an size_t align8 (size_t size) is used to align the allocated size of the malloc.

For the malloc function to work, a XX_find_block function is designed to search through the list of existing allocated memory space and return the free block that is large enough for the malloc or NULL if the desired block is not found. If the find block function return NULL or if the first_block is NULL, the extend_heap function is called to use sbrk to allocate space and the space size should be the sum of the malloc size and block size, this function will return the first address of the new allocated space. Split_block function is used to split the exiting free block in the list if the size of it is much larger than the size we want to malloc and this is for performance consideration. The difference among the three allocation policies is the way they deal with find_block. For first fit, once the fit block is found, it is immediately returned without further searching through the list. For best fit policy and worst fit policy, all the block in the list should be traversed to find the best fit block or largest block.

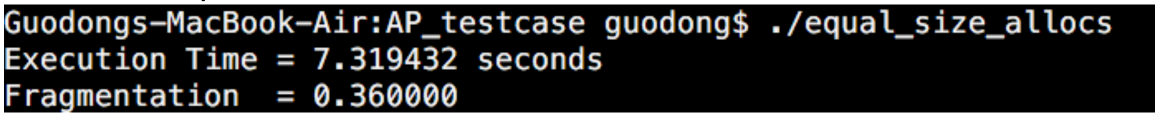
As for the free function, it first checks whether the address passed in is NULL or not. It will return immediately if it is NULL. If the address is not NULL, no valid address assertion is made and the free function just mark the free variable in the meta struct and merge the nearby free fragments to help with performance.

Performance study

Because all of the allocated blocks whether they are free or not are in the list, 10000 equal_size_allocs will run for a long time, So the default NUM_ITERS value is changed from 10000 to 100. Below is the result of the three different test program.

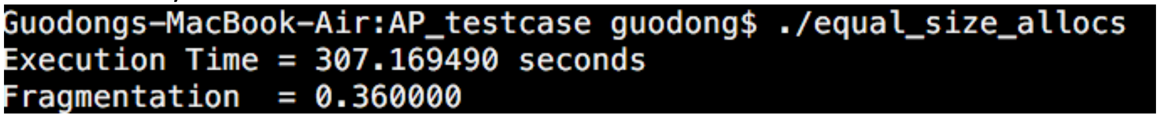
Equal_size_allocs:

First Fit Policy:



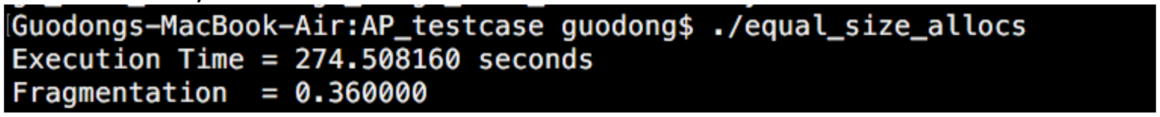
```
Guodongs-MacBook-Air:AP_testcase guodong$ ./equal_size_allocs
Execution Time = 7.319432 seconds
Fragmentation  = 0.360000
```

Best Fit Policy:



```
Guodongs-MacBook-Air:AP_testcase guodong$ ./equal_size_allocs
Execution Time = 307.169490 seconds
Fragmentation  = 0.360000
```

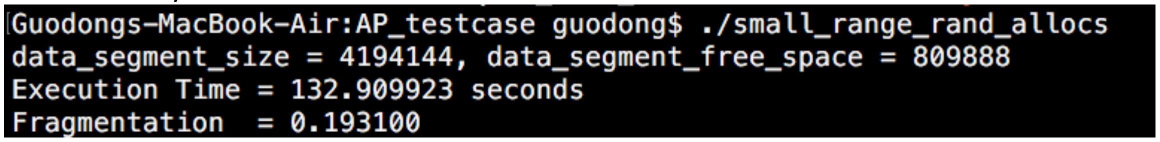
Worst Fit Policy:



```
Guodongs-MacBook-Air:AP_testcase guodong$ ./equal_size_allocs
Execution Time = 274.508160 seconds
Fragmentation  = 0.360000
```

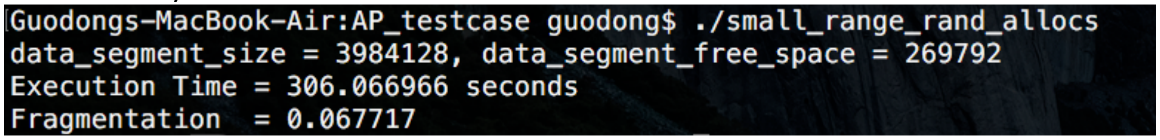
Small_size_allocs:

First Fit Policy:



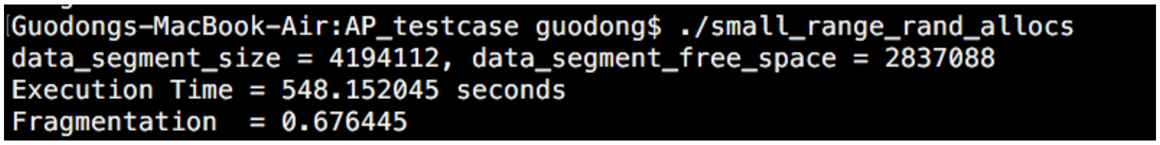
```
Guodongs-MacBook-Air:AP_testcase guodong$ ./small_range_rand_allocs
data_segment_size = 4194144, data_segment_free_space = 809888
Execution Time = 132.909923 seconds
Fragmentation  = 0.193100
```

Best Fit Policy:



```
Guodongs-MacBook-Air:AP_testcase guodong$ ./small_range_rand_allocs
data_segment_size = 3984128, data_segment_free_space = 269792
Execution Time = 306.066966 seconds
Fragmentation  = 0.067717
```

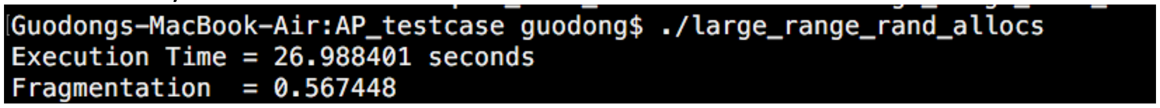
Worst Fit Policy



```
Guodongs-MacBook-Air:AP_testcase guodong$ ./small_range_rand_allocs
data_segment_size = 4194112, data_segment_free_space = 2837088
Execution Time = 548.152045 seconds
Fragmentation  = 0.676445
```

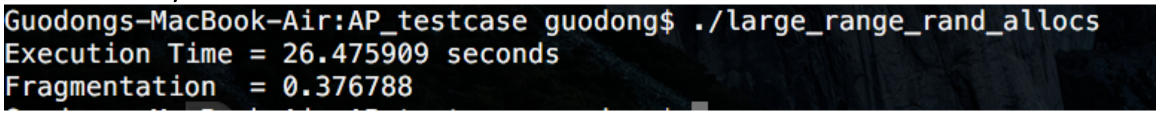
Large_size_allocs:

First Fit Policy



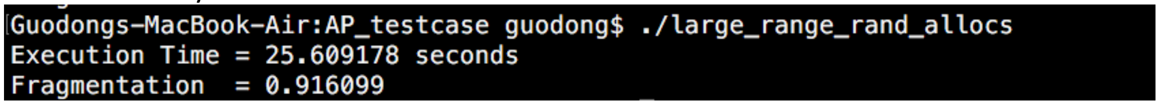
```
Guodongs-MacBook-Air:AP_testcase guodong$ ./large_range_rand_allocs
Execution Time = 26.988401 seconds
Fragmentation  = 0.567448
```

Best Fit Policy



```
Guodongs-MacBook-Air:AP_testcase guodong$ ./large_range_rand_allocs
Execution Time = 26.475909 seconds
Fragmentation  = 0.376788
```

Worst Fit Policy



```
Guodongs-MacBook-Air:AP_testcase guodong$ ./large_range_rand_allocs
Execution Time = 25.609178 seconds
Fragmentation  = 0.916099
```

From the result, it is not difficult to find that First fit usually has a better execution time performance compared with other two allocation policies. The results for Large_size_allocs execution time of the three allocation policies seem the same because in large_size_mallocs 32-64K bytes blocks will be malloced and the increments are only 32B, so the blocks are similar to each other. This may lead first fit to traverse the whole list before find the fit block. The execution time of best fit and worst fit policies are basically the same because they all need to traverse the whole list before getting the desirable block. For small_size_allocs, worst fit execution time is much larger than best fit allocation. It may attribute to the fact that worst fit policy is more likely to run the split_block function, which leads to the extra running time. Al last, if the allocated size is different, then best fit will always has the lowest Fragmentation performance because it makes the most use of the free space in the list.

Part 2: Thread-safe malloc() implementation

To improve the performance of thread-safe malloc(), the pthread_mutex_lock is added after the bf_find_block function. This approach will allow multi-threads access the linkedlist and find the suitable block for malloc() instead of wasting time waiting to find block if the lock is acquired. Two different locks are used to lock the critical section in the malloc() function. The malloc_lock1 is used to lock the critical section of changing the free flag in the linkedlist block after finding the suitable block. To prevent race condition from happening, each time the thread enter this critical section, it will have to check if the free flag is changed, if it is changed, then it should do another find operation. If no suitable block is found, then the malloc_lock2 will be acquired to lock the section of extend_heap operation. Also, in this critical section, curr->next will be checked to ensure that the curr always points to the last block in the list when it’s time to do extend_operation. At last, if the linkedlist Is empty, the malloc_lock2 will lock the section which is used to add the first block to the linkedlist and avoid the situation where two thread may at the same time find the empty list and race for adding the first block to the list. The reason for using malloc_lock2 to lock two different sections is that both of them involve with extend_heap operation.

For the ts_free, malloc_lock1 is acquired after the meta information for the ptr is found and freed after the free flag in the meta information is changed. Note that since the best fit malloc policy is used and for simplicity, no split and merge function is called in the ts_malloc and ts_free function.