***Malloc/Free Implementation Report***

Name: Yue Shao

NetID: ys270

***Definition:***

malloc() takes in a size (number of bytes) for a memory allocation, locates an address in the program’s data region where there is enough space to fit the specified number of bytes, and returns this address for use by the calling program.

The free() function takes an address (that was returned by a previous malloc operation) and marks that data region as available again for use.

***Implementation:***

1. Data structure (meta control block + piece of memory after the meta control block):

* size (the size of memory after the meta control block plus the size of meta control block)
* available (whether this block is free or not)
* next (the address of the next block)
* prev (the address of the previous block)
* fnext (the address of the next free block)
* fprev (the address of the previous free block)

1. Global variables:

* memstart (the first time we use sbrk(0)’s return value)
* lastaddr (the lastaddr of the entire heap)
* freehead (the head of the free list)
* freetail (the tail of the free list)
* hasinit (to indicate whether the other global variables has been initialized or not)

1. Implementation for malloc:

* first fit (void\* ff\_malloc(size\_t size)):

Traverse the free list for the first free block (available = 1) which has bigger size than the required size plus the size of meta control block. Set the available of this block to 0. Remove this block from the free list. If this block is much bigger than the size we required, this means we could split the block into two parts and only occupy the part which has exactly the size we required and add the remaining part of the original block to the free list. If we cannot find such block, we need to extend the heap by using sbrk and link the new created block to the link list of blocks.

* best fit (void\* bf\_malloc(size\_t size)):

Traverse the free list to calculate the value: (meta\_size - required\_size), if there is a zero, we could break out of the while loop, if there is not, we continue till the end of the loop to find the smallest value to ensure that we find the best fit of the required size. If we find such block, we set the available of this block to 0. If this block is much bigger than the size we required, this means we could split the block into two parts and only occupy the part which has exactly the size we required and add the remaining part of the original block to the free list. If we cannot find any block, we need to extend the heap by using sbrk and link the new created block to the link list of blocks.

1. Implementation for free (void bf\_free (void\* ptr) & void ff\_free(void\* ptr)):

* set the available of this block to 1
* merge adjacent free blocks, remove adjacent blocks from freelist if there available is 1
* add the newly formed free block to the freelist.

***Performance test:***

1. Evaluating indicator:

* data\_segment\_size: the entire size of the heap = lastaddr - memstart
* data\_segement\_free\_space: traverse the free list to calculate free space’s size
* execution time: the runtime of the testcase
* fragmentation: the ratio of data\_segment\_free\_space and data\_segment\_size

1. Compare of fragmentation between having a split function and not (using ff as an example):

* Result for not having a split function:

图片包含 屏幕截图

描述已自动生成

* Result for having a split function:

图片包含 屏幕截图

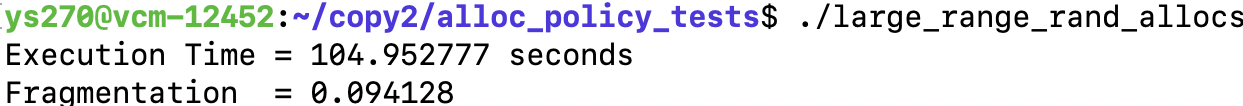
描述已自动生成

* Analysis:

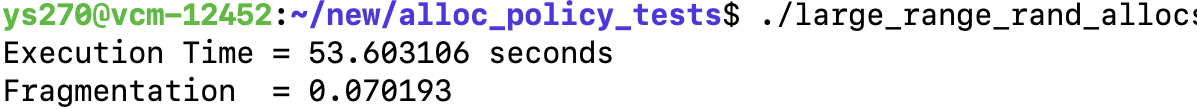
As we could see from the result, after adding the split function, the fragmentation increased from 0.000148 to 0.037012 for small range rand allocations, 0.000638 to 0.070193 for large range rand allocations, the heap’s size decreased dramatically for small range rand allocations, but the runtime increased in every testcase. This is easy to understand because we sacrifice some extra time in every allocation operation to find whether the block suitable for the required size can be split and if the answer is true, we add the remaining space to the end of the free list, not just removing the whole block out of free list. The heap size would be much smaller, and the free space would be much larger, thus the fragmentation increased.

1. Compare of runtime between having a free list and not (using ff’s large range rand allocation as an example):

* Result for not having a free list:



* Result for using a free list (add two pointers in the block’s struct pointing to the next and last free blocks of it):



* Analysis:

We could tell from the result that the runtime for not having a free list is much longer than having the free list. This is because the first method search through the whole list to find free blocks which has the suitable space for the allocation, the second method only traverse the free blocks to search.

There is also a little difference between the fragmentation of the two because the free list is not continuous in space in my implementation of this project. So when we search for the first block which has enough space for the required size, the first method strictly search from the heap’s head to its tail, while the second would search the free list whose order is related to the operations of the testcase. Thus, the two method would probably find two different blocks, and this would explain the reason for this difference.

1. Compare between first\_fit and best\_fit (having freelist and split function):

* Result for first\_fit:

图片包含 屏幕截图

描述已自动生成

* Result for best\_fit:图片包含 屏幕截图

  描述已自动生成
* Analysis:

For equal\_size\_allocs, the size of block is fixed, so every block in the free list is the same. Every time, it takes the front block as in either two ways, so the runtime is pretty close. For fragmentation, it is obviously the same since it is calculated halfway, both around 0.45.

For small\_range\_rand\_allocs, unlike the case in equal\_size\_allocs, the size of every free block is random since the size of blocks is uncertain. Theoretically, when encountering a block whose size is greater than requested, it is a first-fit case, but may not be a best-fit case. As the results above, the runtime of first fit is longer than best fit, and the size of the heap is larger. We could use an example to explain this: if we have a list with size of 20, 15, 10, we are requesting 15, 20, 10. In best fit, it will takes up these three. In first-fit, it would take up 20 for 15, resulting in a new sbrk() call, which in this case consumes more runtime than iteration.

For large\_range\_rand\_allocs, it is doing the same thing as small\_range\_rand\_allocs, but the size of blocks varies greatly compared with small\_range\_rand\_allocs. This would result in a rather long linked list because in this case it’s not easy to find the suitable block to allocate space, thus we need to extend heap many times. Then the iteration may consume longer time than sbrk().

In reality, things happen more likely as the case in large\_range\_rand\_allocs. Based on the result we have and the analysis we make, I would recommend first fit since it does not need to traverse the long free list. However, if we already know that the data range is rather small, we could use best fit instead to save some space.