CS 3104 Memmory Allocator ID: 150007637 Lecturer: Kasim Terzic Date:20.10.2017

Design & Implementation

My implementation of the memory allocator consists of mymalloc() and myfree() as main functions and also a couple of helper functions. I have also implemented incremental memory allocation by allocating smallest possible size of a page returned by malloc (4096). I have two "define" statements for better readability and also the prototypes of some of my functions above the rest of the code. I also have three helper global variables. I have made the decision to use a doubly linked list as my main structure because it seemed most straightforward in terms of accessibility and flexibility to have next and previous pointers for my free blocks. Another important tradeoff that I made is linear search of blocks for coalescing in place of constant adding. Because you can order addresses linearly based on how big is the address and store their index in a table, then search for them at constant time.

Helper functions

initMem – it is used only once the first time the user requires space for allocation. It simply mmaps one page(4096 bytes) to my beginning of heap and sets pointers accordingly for cyclic structure.

Cut – it takes pointer to a block, which will be returned to the user and a size of the block(payload + header size). It adjusts sizes of requested block and remainder and adjusts pointers so that "fit" is swapped out for the new remainder block, which becomes a part of the free list.

find_first_fit – This function takes a size for requested block as a parameter, traverses through our free list and returns a pointer to a free block with sufficient size or NULL.

TryCoalesce – This function is used to merge adjacent free blocks in memory. It takes a pointer to a payload of a free block and returns a single character used as a boolean to indicate 'f' for failure to find adjacent blocks and 's' for success. It first checks if there are any free blocks in the free list. After that pointer to current header is adjusted and list is traversed in both directions until starting point is reached or NULL value is found, or at least that is how it should be. Unfortunately, in the limited time, I did not manage to find the bug in my cyclic structure, so if the "counter < 50" is removed from the condition there is a risk of an infinite loop. Inside the loop block we check if our current nodes are actually the beginning of the heap in which case we just skip the current iteration, because the beginning of the heap is not to be touched according to my implementation (more practical solutions are of coarse present which will be mentioned below). Otherwise, we try to match the end of our current payload to the start of the header of the next block or the header of the previous pointer by substracting payload size of previous from our current pointer. Then we want to add the sizes and adjust pointers of merged block accordingly.

Main Functions

myFree – this function takes a pointer to a payload and either tries to coalesce adjacent blocks or insert new block in the free list using the last in first out procedure. Pointers are adjusted accordingly so as the newly freed block is after the beginning of the heap. This does not actually unmap the memory and hand it back to the OS, but it instead allows the user to overwrite the memory.

Myalloc() - myalloc takes size of user request as a parameter and returns a void* to a memory block with that size. The first time it is called initMem() helper function is invoked after that second helper function is invoked find_first_fit() to look for a free block. If no free block is found I decided depending on how big the memory request is by the user to mmap a memory region, adjust its pointers and call helper function cut() if memory block is big enough. I return NULL if no memory is available. If fit from free list is found I try to cut() that block otherwise just return the payload to the user.

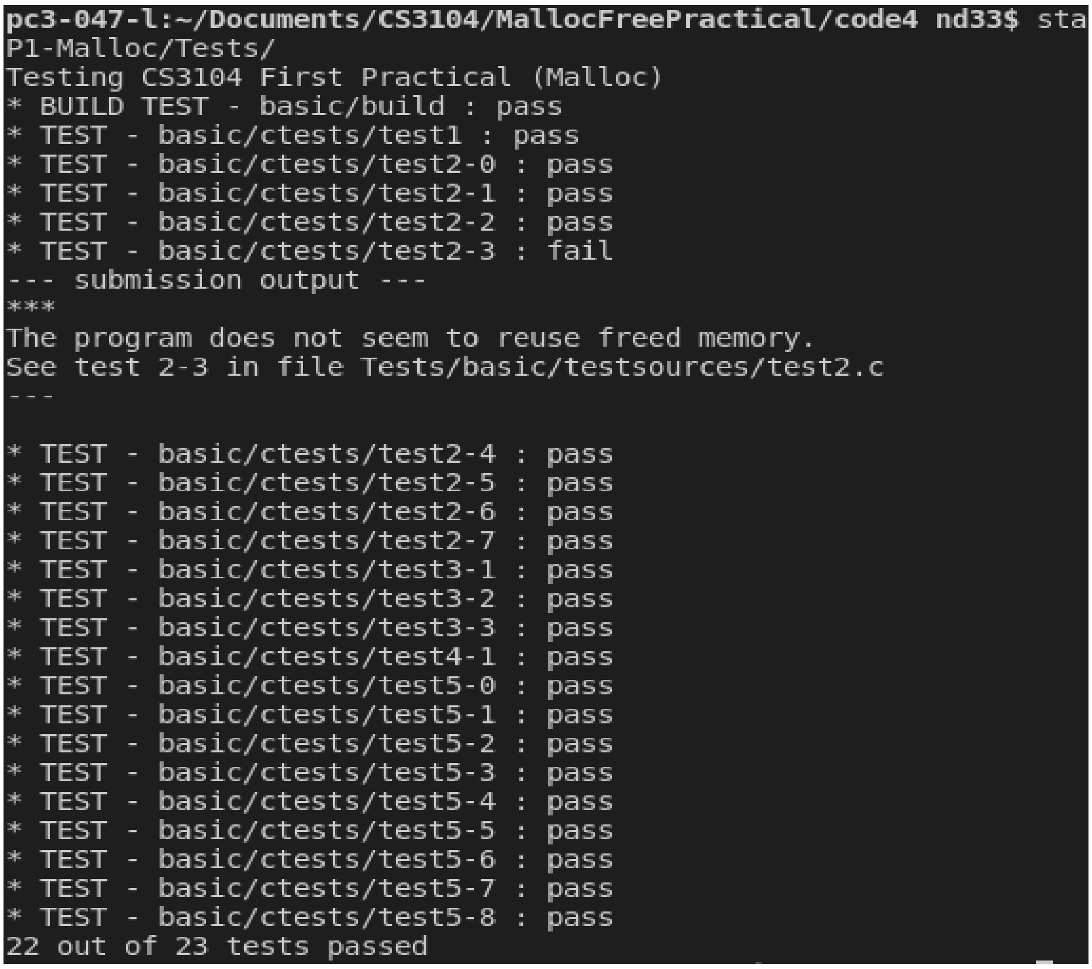
My implementation does not have fully error prone support of coalescing. There is a minor mistake somewhere in my free list pointers or coalescing implementation, which given the cyclic structure of my doubly linked list can lead to infinite loop when searching for adjacent memory blocks in the free list. I have prevented that temporarily by introducing a counter which counts the maximum number of times the loop can iterate (50 seemed reasonable number for iterations). However, this limits the two-way (forwards and backwards) search of my implementation, but I made it two-way so it can still cover reasonable part of the list nodes. Anyway, given more time I would have found a way to fix that issue. The other minor flaw in my implementation is that for the beginning of my heap I initialise a BEGIN_OF_HEAP block header pointer with size of 4096 bytes just to keep track of my bottom address. I do not use the allocated space, this can be amended by either using just a header size for the beginning of heap or by utilising free space. I also have 24 byte headers, which contain pointer to previous and next nodes and a size (pointers can be omitted for allocated blocks).

Testing

For the purpose of testing my implementation I have thoroughly examined normal and corner cases. For instance, allocating and freeing variables of different length, allocating huge amounts of space supporting arrays, tree based structures, lists etc. I have used valgrind to check my program for memory leaks and other memory related issues

Stacscheck

Test 2 fails, because of my minor bug in the coalescing, which I described in the previous section and did not have the time to address but wound do so in the future.



```
pc3-047-l:~/Documents/CS3104/MallocFreePractical/code4 nd33$ sta
P1-Malloc/Tests/
Testing CS3104 First Practical (Malloc)
* BUILD TEST - basic/build : pass
* TEST - basic/ctests/test1 : pass
* TEST - basic/ctests/test2-0 : pass
* TEST - basic/ctests/test2-1 : pass
* TEST - basic/ctests/test2-2 : pass
* TEST - basic/ctests/test2-3 : fail
--- submission output ---
***
The program does not seem to reuse freed memory.
See test 2-3 in file Tests/basic/testsources/test2.c
---

* TEST - basic/ctests/test2-4 : pass
* TEST - basic/ctests/test2-5 : pass
* TEST - basic/ctests/test2-6 : pass
* TEST - basic/ctests/test2-7 : pass
* TEST - basic/ctests/test3-1 : pass
* TEST - basic/ctests/test3-2 : pass
* TEST - basic/ctests/test3-3 : pass
* TEST - basic/ctests/test4-1 : pass
* TEST - basic/ctests/test5-0 : pass
* TEST - basic/ctests/test5-1 : pass
* TEST - basic/ctests/test5-2 : pass
* TEST - basic/ctests/test5-3 : pass
* TEST - basic/ctests/test5-4 : pass
* TEST - basic/ctests/test5-5 : pass
* TEST - basic/ctests/test5-6 : pass
* TEST - basic/ctests/test5-7 : pass
* TEST - basic/ctests/test5-8 : pass
22 out of 23 tests passed
```

Evaluation and Conclusion

This practical has been one of the difficult and at the same time the learning experience has been quite good. I gained significant knowledge in using doubly linked lists, dwelling in the low level C territory and understanding how memory allocation works.
In conclusion, I think that I have made a reasonable attempt at implementing this practical and despite some of my flaws have accomplished the problem in question up to a good degree of success.