Asst1: ++Malloc

readme.pdf
Rutgers CS 01:198:214 Systems Programming
Professor John-Austen Francisco
Authors: Anthony Siluk & Alexander Goodkind
Due: 10/15/2019

This project aims to create a first-fit algorithm drop-in replacement of the C89's standard `malloc()` and `free()`. The included header file `"mymalloc.h>"` defines a directive to replace all occurrences of `malloc()` with `mymalloc()` and `free()` with `myfree()`

metadata:

- We chose to use a struct that contains two fields: `char inUse` and `unsigned short blockSize`. Our motivation behind use char for inUse is that it only requires 1 byte of data & we treat `f` as false and anything else as true (for simplicity we use `t` when manually setting it true).

- For blockSize we decided that since the original assignment only allows up to 4096 bytes of data, we will never have a practical use for `size_t` (which is just an `unsigned long long` and requires much more data than that of an `unsigned short`).

we decided that we do not need to have a next pointer as we can always calculate the next node by using pointer arithmetic:

a. given a node pointer address, we can find the next block of data by using `address + sizeof(node) + 1`.
b. given a user address, we can find its corresponding metadata by using this formula: `userAddress - sizeof(node) - 1` .

- Our `getNext(char*)` function looks at a given node address and performs returns the result from a.)
- Our myfree looks at the given user address and converts it using b.)

note for grading: with our assumptions about heap size, we were able to produce a metadata with `sizeof(node)` = **4 bytes** *[assuming you are running on the x64 ilab machines with `gcc -O0 -std=gnu89` and with `gcc -v`: gcc version 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)]*

mymalloc:

there are two main functions that mymalloc has, and each function uses two helper functions to perform the tasks of classic `malloc()` and `free()`

1. `void * mymalloc(size_t size, char *file, int line)`
   a. when mymalloc is first called with a given size, it will first check if anything has been allocated yet to the array, it does this by checking the flag: `heapUninitialized`, which by default is true.
   b. mymalloc will mark the very first block as `inUse` and set it's size to `HEAP_SIZE - sizeof(node) - 1`.
2. `char * findOpenNode(size_t size)`
   a. findOpenNode will traverse uses `getNext()` until it finds a node with a.) enough space b.) not inUse.
3. `void * splitBlock(char *current, size_t size)`
   a. once an open node is found we call splitBlock where it will take the given node & allocate ONLY what space is needed, and then directly after will assign a new node that is not inUse with the remaining data.
4. `void myfree(void *address, char *file, int line)`
   a. given a user pointer we find the corresponding node pointer (see above) then we call:
5. `void combineFreeBlocks()`.
   a. this will run any time a pointer is free, when it is called it will traverse the entire array and look for directly adjacent blocks, the first free block in a series of 2 or more free blocks has its blockSize changed to an accumulation of the following adjacent blocks.
   b. Running this every time we free we can guarantee that we'll always have proper metadata stored and are able to find the next inUse block.

Error Handling:

1. When calling mymalloc, we check if the requested `size_t` will *even* fit on our heap: by doing `size > HEAP_SIZE-sizeof(node)-1`
   a. in the case there isn't we print an out of memory error and `return NULL`
2. Next when mymalloc calls `findOpenNode` we traverse until we find an open node that has a large enough size
   a. in the case we print an out of memory error and `return NULL`
3. When calling myfree, we first check that the user address is not NULL
   a. if it is just silently return without printing an error
4. Next, myfree checks if the address itself is within the actual array by doing `myblock + sizeof(node) + 1 > (char*) address || (char*) address > &myblock[HEAP_SIZE] - sizeof(node) - 1`
   a. if it isn't we print that the address is invalid and out of range, then we return
5. myfree then checks if the corresponding node address matches one in the heap
   a. if it does we then check if the address is already marked as free, we return an error if it has
   b. if it doesn't we return that the address hasn't been allocated yet

memgrind:

We run 6 tests, 2 of which are custom made:
our results after running on the ilabs are:
(ns = in nanoseconds)

- `Average run time for A = 28421.3 ns`
  - Here we are malloc'ing 1 byte and immediately freeing it, 150 times
    - we can interpret this as 28421/150 = ~189 ns, so on average it takes ~189ns to find a free node, mark it as allocated, free it (which includes `combineFreeBlocks()`)
- `Average run time for B = 65230.4 ns`
  - Here we are malloc'ing 1 byte, storing the pointer in an array, 150 times and After 50 times, free 50 1 by 1.
    - our interpretation of this test case meant that essentially we malloc 50 times, then free 50 times, 3 times total (3*50 = 150)
    - and this follows that if we take a look at B's average run time is about 3x longer than A's
- `Average run time for C = 7780.97 ns`
  - Here we are randomly choosing between a 1 byte malloc() or free()ing a 1 byte pointer until 50 bytes have been allocated. Then freeing the rest.

        - since there is a 50% chance that it will be a 1 or 0, we can take 7780.97/2 = ~3890ns
            - this can be naively interpreted that it took 3890ns to allocate 50 blocks, and the same to free them, however there is more that could be say about allocating but we don't have enough information since all our tests combine the timing of free & malloc
- `Average run time for D = 8319.12 ns`
    - Randomly choose between a randomly-sized malloc() (between 1 and 65 bytes) or free()ing a pointer.
        - This has a similar timing to part C, so we can make the same naive assumption given the little information we have.