A4: Malloc Replacement Assignment

CS 6013
Samantha Pope
03/14/2024

How to run my tests: I included all of my tests in the testStruct.cpp, which holds my main() function. If you run this file, then all of the tests in that file will run and print out if they pass or fail. Each function has a comment above it specifying what it tests.

I tested the average time of allocation and deallocation in my testStruct.cpp file. It is called "TimingBenchmark". I set up a test that had a loop of 10,000 iterations. In each iteration, both the custom MyMalloc and built-in Malloc allocated and then immediately deallocated 128 bytes of memory. My function times the amount of time these two functions take, and then found the average at the end of the 10,000 iterations. Below are the results of two calls to this function:

```
Average allocation+deallocation time for MyMalloc object: 0.00137091ms
Average allocation+deallocation time for built in malloc: 6.19644e-05ms
Average allocation+deallocation time for MyMalloc object: 0.00116764ms
Average allocation+deallocation time for built in malloc: 5.63116e-05ms
```

The results show that MyMalloc is significantly slower than the built-in malloc for allocating this memory. MyMalloc uses mmap and munmap for memory allocation and deallocation. These system calls are slower than malloc and free. malloc can use a strategy that allocates larger blocks of memory from the operating system and then parcel it out form this pool. Because of this, malloc and free calls can be handled in the user space without needing to ask the operating system everytime, which makes it much faster. Another reason why it could be operating so much faster than mine is because the built in malloc implementation has been optimized to handle a lot of patterns efficiently. MyMalloc rounds up allocations to the nearest page size and uses mmap or munmap for each operation.

To speed this process up, we could reduce the amount of times we make system calls. Instead of calling mmap every time, we could ask for larger blocks of memory up front and then divide that out as it is needed. We could also use memory pooling. Memory pooling is a strategy where you keep a pool of recently freed blocks so you can reuse them for new allocations without having to use mmap or munmap. This would be especially helpful for the benchmark test that I created. This is because we used the same size (128 bytes) over and over. So if we found a way to have the next allocation use the same spot for those 128 bytes that we did, then we wouldn't need to run mmap or munmap everytime we allocate 128 bytes.