Tom Burns and Andrew Dulichan
03/10/2019
CS 214 – Systems Programming
Assignment 1 README

Introduction

Our Assignment 1 implements malloc() and free() in our own unique way. We do not use a conventional data structure in our code; instead we use our own form of a custom “data structure” which is efficient and sufficient for this assignment. We chose to implement a pseudo linked list as this allowed us to reduce the total metadata size.

The metadata is held in this custom data structure and its implementation is as follows:

We use a struct that is 6 bytes in size which contains three variables. We have a flag of type char to check if a block of our static array has dynamically allocated memory from mymalloc(). This is useful as it allows us to check certain conditional statements on the fly and appropriately run code depending upon if there is allocated space in a specific block at any given time. Secondly, our magic number is stored as a short which allows us to check if the myblock memory allocation structure has been set up yet, as well as test if specific parts of the array are metadata for use in either malloc or free. We chose a “safe” magic number of 5218, which is 2 bytes. The likelihood of a collision with this value is 1/2^16. Lastly, we store the size of the metadata as a short as that is the minimum data type size which covers the whole of our 4096-byte array. (char would be too small at 2^4) By knowing the size of our metadata, we can traverse through our pseudo-data structure without any pointers to the next point of data in the structure (that being a flattened linked list with implicit nodes and ‘pointers’ – though they’re not really pointers). We know where each point in the list is due to the size of the metadata as well as the size of the blocks that they represent.

While our implementation is efficient, it is not as efficient as possible. The best possible theoretical implementation for the metadata in this assignment would have been a 4 byte struct, containing a short for the magic number and another short that could be referred to as data. The data variable would be very special, because you would have to perform bit operations on it to obtain the information you are looking for. The first bit of the number could be used as the flag, like the char we used in our implementation, while the remaining 15 bits would allow a size of up to 2^15. We chose not to go down this route of implementation as it would have been very difficult to code and would not be easy to deal with and debug because of the bit operations, compared to having a struct with easy to work with chars and shorts. In the end our implementation was 2 bytes less efficient than one of the most efficient implementations.

Failure Cases

Our overall code handles the common failure cases properly. Our myfree() method contains the error checking code. We tested three failure cases which are:

1. Allocating 4097 bytes of memory in one go which one byte too large. The call is denied and handled properly.

2. Allocating 4096 bytes of memory minus the size of the metadata itself, and then immediately allocating another single byte from the array. This results in a similar situation where the final size allocated is again one byte too large, which ends up being denied due to our error-checking implementation.

3. Allocating memory in a way that is similar to test case 2, but we directly allocate 4097 bytes of memory over two lines of code (two separate malloc() calls, one for 4096 bytes and one for 1 byte). We don’t use the size of the metadata at all in this failure test case.

<u>Workload findings and results</u>

We measured our workloads in microseconds. Here are our results for Workloads A through F.

```
--------------------WORKLOAD A---------------------

Average Run Time: 129.400000 microseconds

Memory after run:
Inuse: 0        Size: 4090      Magicnumber: 5218

--------------------------------------------------------------

--------------------WORKLOAD B---------------------

Average Run Time: 182.200000 microseconds

Memory after run:
Inuse: 0        Size: 4090      Magicnumber: 5218

--------------------------------------------------------------

--------------------WORKLOAD C---------------------

Average Run Time: 228.700000 microseconds

Memory after run:
Inuse: 0        Size: 4090      Magicnumber: 5218

--------------------------------------------------------------

--------------------WORKLOAD D---------------------

Average Run Time: 239.120000 microseconds
```

```
Memory after run:
Inuse: 0          Size: 4090       Magicnumber: 5218

------------------------------------------------------------

--------------------WORKLOAD E----------------------

Average Run Time: 247.530000 microseconds

Memory after run:
Inuse: 0          Size: 4090       Magicnumber: 5218

------------------------------------------------------------

--------------------WORKLOAD F----------------------

Average Run Time: 470.440000 microseconds

Memory after run:
Inuse: 0          Size: 4096       Magicnumber: 5218

------------------------------------------------------------
```

In short, we found our workloads to function properly and quickly as well. See our textplan.txt for more information on what we chose for workloads E and F, and why we chose them.

Conclusion

As a whole, our code is implemented efficiently and is also robust, as we see from our speedy workload data results and all failure test cases being handled successfully. Our version of the assignment handles all of malloc() and free()'s jobs gracefully!