Zohaib Zahid (zz165) & Kristina Poon (kp622) February 26, 2017
Assignment 1 - Memory Allocation++

Function/Design Descriptions

A Doubly Linked List (DLL) structure is used to implement the malloc function. As indicated in the assignment's instructions, the size of the main memory is 5000 bytes. When malloc(x) is called, the function will determine whether there is enough unused memory space available to store metadata (a struct) along with the size of the memory requested. This is accomplished by traversing the DLL and checking the parameter values of "isfree" and "size." There are exceptions explained more under functions, where only space is given and no metadata stored.

This project includes the following functions:

- **void* mymalloc(unsigned int size, char* file, unsigned int line)**
  - Here the function takes the size requested and traverses the main memory for space. Starting at the root of the main memory. First checks if the currentNode has enough size, if it does then check "isfree", if not free then move to next node. The next check is to see if the size is enough for the requested memory but not including the header, in this case it just returns the pointer to the memory without creating a header. And the last part is that it has enough space for both header and requested size, it then makes a new header and points it to the rest of the free memory while available, and then this node's along with the last node's prevs, nexts, and isfrees are updated.
  - Error statements are printed for the following cases:
    - Too many bytes requested at once (ie. malloc(5001))
    - If 0 bytes requested (ie. malloc(0))
    - If requested is bigger than what is left (ie. malloc(5000) which is good but then malloc(1) right after)

- **void myfree(void* p, char* file, unsigned int line)**
  - This function works almost as opposite of the malloc. It takes the pointer that is being freed and finds its location in the DLL created by malloc. Then casts the pointer called ptr to a metadata node to compare with a pointer that traverses the DLL called currentNode. Then when currentNode == ptr, it checks whether it has been freed already or not. If not it will free the ptr and change the "isfree" parameter and then check to the left and right of the currentNode to see if there are other nodes that are free and combine them so that there aren't chunks of small memory that are free that would later create problems for larger memory requests, where you have enough free memory but in chunks so it would not be able to allocate it.
  - Error statements are printed for the following cases:
    - Freeing a pointer twice (ie. free(ptr1) and then right after free(ptr1) again)
    - Freeing a pointer not allocated by malloc (ie. int x or int* y, then free(&x) or free(y) respectively)
    - Freeing a pointer that has been incremented not to an invalid pointer address (ie. char * p = (char *)malloc(200), and then free(p + 10))

File Descriptions

This project includes the following files:

- Makefile
  - make (which finds the directive to all)
    - So this will make and compile memgrind.c which relies on the object file mymalloc.o
  - memgrind.o
    - Here compiles an executable object file mymalloc.o which relies on the header file mymalloc.h
  - clean
    - This removes the a.out and object files created by the Makefile
- mymalloc.c
  - Includes the two functions that are described under functions
    - **void* mymalloc(unsigned int size, char* file, unsigned int line)**
    - **void myfree(void* p, char* file, unsigned int line)**
- mymalloc.h
  - Includes the struct of the doubly linked list called MemBlock
  - And defines the functions named under mymalloc.c seen above
- Testcases.txt
  - You will find a descriptive overview of the workloads created for E and F
- Memgrind.c
  - This is the file that the main is written in and it calculate the mean time for execution of each workload over the 100 executions and output it using printf.
  - Each workload is also described in the workload section below

Workload Data/Findings

**Workload A**

Workload A's functionality is to malloc() 1000 times and once that is complete, free() 1000 times. This is unable to be completed however because a 5000 memory cannot accommodate 1000 calls of malloc(). The total byte size to be stored between the actual data byte and then metadata is 1 byte and 32 bytes respectively. Each call of malloc() would require 33 bytes to be stored each time. The maximum malloc()s this would allow is 150 as 33*150 = 4950. Hence, errors are encountered. Besides this, the structure of workload A required the usage of two consecutive for-loops. Compared to workload B, it was marginally faster. Its runtime efficiency is O(n).

**Workload B**

Workload B's execution time is similar to that of workload A's. Although A and B ultimately end with the same empty memory, unlike workload A's procedure, workload B malloc()s and immediately free()s the memory it just malloc()ed within the same single loop. Its commands within the loop prevent errors from arising. Its runtime efficiency is O(n).

**Workload C**

This workload was slightly more complex than both A and B where instead of establishing a set time to malloc() or free(), the instances of both procedures were randomized. However, to prevent error, a condition was established so that if the entirety of the memory was being unused and there was nothing to execute free() with, the function malloc() would be defaulted. Once malloc() has been called 1000 times, which has been accounted for using a while loop, the function begins emptying the memory if the size counter for the array is still >0. This required the usage of another while loop following the first. The runtime efficiency is O(n).

**Workload D**
Workload D's operations are similar to that of workload C except now the size of the data being stored is now also randomized between 1 byte and 64 bytes as opposed to a consistent 1 byte. Generation of the randomized byte-size and its storage causes an increase in execution time making this comparatively slower than C as seen in the increase in workload time seen below. The usage of two separate consecutive while-loops remains as previously shown, making the runtime efficiency O(n) with n being the number of loops that whichever while-loop looped moreso.

**Workload E**
Workload E's purpose was to observe the effects on execution time if free() was executed on a pointer that was never at an endpoint. As it turns out, this caused a significantly large hindrance as seen from execution times among all workloads. While this workload also used larger data bytes, the effects of large data were seen to have an effect of a lesser degree as exhibited in workload F, which ran at nearly half the speed that workload E had. Despite the spike in execution time, workload E's runtime efficiency remains at O(n).

**Workload F**
Workload F's intended purpose was to observe the effects of large data on execution time. The workload runs malloc() for data byte sizes ranging between 1 and 1024 until the memory reaches a capacity that falls within 1024 bytes of 5000 bytes. Once this range of capacity is reached, the memory empties itself and repeats the process until this has been done 1000 times. The runtime efficiency of F is unique from all other workloads. There is a do-while-loop followed by a while-loop both nested within a for-loop. The runtime of the two consecutive loops results in a simplified runtime efficiency of O(n), but when nested within a for loop, the run-time efficiency becomes $O(n^2)$. Despite the large data size it handles and its larger runtime efficiency, the workload still maintains an execution time that is within close range of the other workloads' execution times. This, of course, indicates that as the number of mallocs continuously scales, execution time will slow down.

Execution Times for each Workload in microseconds:

```
workloadA: 194390.020000
workloadB: 200917.390000
workloadC: 3346167.970000
workloadD: 196351.980000
workloadE: 585288.430000
workloadF: 198260.150000
```