AUTHORS:

Jason Kwak RUID: 126007397 netID: jgk68

Anne Whitman RUID: 042007629 netID: alh220

NAME:

memgrind.c, mymalloc.c

DESCRIPTION:

mymalloc.c – this file holds a mymalloc( ) and myfree( ) method that overwrite the standard implementation of malloc( ) and free( ) system calls but catch more errors. We are using a large block of 5000 bytes to implement these methods. The mymalloc( ) function returns a pointer to somewhere inside of the 5000 byte array, and the free( ) method either frees that pointer or returns an intelligent message to explain why it cannot free the pointer.

memgrind.c – this file runs stress tests on the malloc( ) and free( ) implementations, times the function calls, and reports the mean time over 100 executions and then outputs the mean time.

FUNCTIONS:

mymalloc.c:

char * mymalloc(int numOfBytes, char * myfile, int myline);

- When malloc(x) is called, it gets replaced with mymalloc(x , __FILE__, __LINE__) from the header file definitions. The function call saves the file and line into variables to use for error reporting. It then takes the number of Bytes requested and checks to see if there is enough free space in memory (the 5000 bytes we've preallocated) to allocate. If there is not enough space, a null pointer is returned, otherwise a pointer to the block is returned to the user.

void myfree(char* userPointer, char* myfile, int myline); - When free(x) is called, it gets replaced with myfree(x, __FILE__, __LINE__) from the header file definitions. The function all saves the file and line into variables to use for error reporting. It then takes the user pointer and tries to find the matching pointer and free it, reporting errors when it cannot find the pointer. In particular, it reports back if a pointer was not given by malloc (not in range), if it's already been freed, or if it's misaligned (in the block of data, but not a pointer).

memgrind.c

void test(A); - 1000 separate malloc( )s of 1 byte, then free( ) the 1000 1-byte pointers one-by-one

void test(B); - first malloc( ) 1 byte and immediately free it - do this 1000 times.

void test(C); - randomly choose between a 1 byte malloc( ) or free( )ing a 1 byte pointer - do this 1000 times.

- -Keep track of each operation so that you eventually malloc( ) 1000 bytes, in total.
- -Keep track of each operation so that you eventually free( ) all pointers.

void test(D); - randomly choose between a randomly-sized malloc( ) or free( )ing a pointer - do this many times.

- -Keep track of each malloc so that all malloc( )s do not exceed your total memory capacity.
- -Keep track of each operation so that you eventually malloc( ) 1000 times.
- -Keep track of each operation so that you eventually free( ) all pointers.
- -Choose a random allocations size between 1 and 64

void test(E); - malloc( ) and free() the same random number repeatedly 100 times.

void test(F); - malloc( ) and free() different random numbers repeatedly 100 times.

Workload Average Execution:

```
Average TestA = 0.002300 seconds
Average TestB = 0.000000 seconds
Average TestC = 0.003200 seconds
Average TestD = 0.000100 seconds
Average TestE = 0.000000 seconds
Average TestF = 0.000000 seconds
```

ALGORITHM

Malloc Function

```
    if requested bytes = 0
        print  “no bytes requested”
        return null
    if head metadata does not yet exist
        create head metadata
        create first metadata and size
        return user pointer
    else
        while not end of array
            if there’ s room in between meta
                add meta and bytes requested
                return user pointer
                break
        add meta and bytes to end of array
        return user pointer
```

Free Function

```
    if nothing has been allocated yet
        return error
    if user pointer is not in range of byte array
        return error
    While user pointer != meta pointer + size of meta
        current meta = next meta
    If end of array is reached
        return error
```