CS 214: Systems Programming, Spring 2015
Assignment 5: Error detecting malloc() and free()

1 Introduction

In this assignment, you will implement malloc() and free() library calls for dynamic memory allocation that detect common dynamic memory programming errors. You can use the malloc() and free() code from the Kernighan and Ritchie C book or use the algorithms presented in lecture. To keep things simple, you can use a static char array (e.g. static char myblock[5000]) for your malloc() and free() to use to manage dynamic memory. This eliminates worries about where the dynamic memory comes from.

The basic functionality of malloc(size_t size) is to return a pointer to a block of the requested size. This memory comes from a memory resource managed by the malloc() and free() functions. To keep things simple, you can use a static char array (e.g. static char myblock[5000]) for your malloc() and free() as your memory resource. This eliminates worries about where the dynamic memory comes from. The free(void *) function returns the allocated block to the memory resource, making it available to use in later malloc() calls.

The reason we need an error-detecting dynamic memory manager is to detect commonly-made errors made by programmers using dynamic memory. Some of these problems are described in the next section.

2 Detectable Errors

Your malloc() and free() implementation should be able to catch at least the following errors:

- free()ing pointers that were never allocated. For example

```
int x;

free( &x );
```

- free()ing pointers to dynamic memory that were not returned from malloc(). For example

```
p = (char *)malloc( 200 );

...

free( p + 10 );
```

- redundant free()ing of the same pointer. For example

```
free( p );

free( p );
```

is an error, but

```
p = (char *)malloc( 100 );

free( p );

p = (char *)malloc( 100 );

free( p );
```

is perfectly valid, even if malloc() returned the same pointer both times.

- Saturation. What happens when you have a test program that allocates all of available dynamic memory? Your implementation should be able to handle this contingency.
- Fragmentation. It is possible to allocate and free many small blocks in a way that leaves only small blocks available for allocation. When a request for a large block is made, the request fails even though the total amount of free memory is larger than the requested block size. There are ways to deal with this problem. First, one part of your memory resource can be reserved for small blocks, leaving the rest available for larger blocks. Second, large blocks can be allocated from one end of your memory resource and small blocks can be allocated from the opposite end. You can use these ideas or invent your own.

Of course, there are other possible dynamic memory management errors. You are always free to invent ways to detect new errors in your malloc() and free() implementation.

3 Responding to Detected Errors

Your modified malloc() and free() should report the precise calls that caused dynamic memory problems during program execution. Let's take advantage of a couple of preprocessor features in the following sample macro definitions:

```
#define malloc( x )     mymalloc( x, __FILE__, __LINE__ )

#define free( x )       myfree( x, __FILE__, __LINE__ )
```

I'll leave it to you to make the best use of what's in these macros.

4 What to turn in

- A writeup documenting your design.

- A file called hwextra-testcases.txt that contains a thorough set of test cases for your code, including inputs and expected outputs.
- All source code including both implementation (.c) and header(.h) files.
- A makefile for producing executable test program, with multiple targets allowing selection of test programs.

Your grade will be based on:

- Correctness (how well your code is working).
- Testing thoroughness (quality of your test cases).
- Efficiency.
- Good design (how well written your design document and code are, including modularity and comments).