CS 214: Systems Programming, Fall 2016
Assignment 1: A better malloc() and free()
(Part 0)

0. Introduction

In this assignment, you will implement malloc() and free() library calls for dynamic memory allocation that detect common programming and usage errors. For this project, use a "first free" algorithm to select free blocks to allocate.

Malloc(size_t size) is a system call that returns a pointer to a block of memory of at least the requested size. This memory comes from a main memory resource managed by the operating system. The free(void *) function informs the operating system that you are done with a given block of dynamically-allocated memory, and that it can reclaim it for other uses.

You will use a large array to simulate main memory (static char myblock[5000]). Your malloc() function will return pointers to this large array and your free() function will let your code know that a previously-allocated region can be reclaimed and used for other purposes.

Programmers can easily make some very debilitating errors when using dynamic memory. Your versions of malloc() and free() will detect these errors and will react nicely by not allowing a user to do Bad Things.

1. Detectable Errors

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

A: Free()ing addresses that are not pointers:

```
int x;
free( &x );
```

B: Free()ing pointers that were not allocated by malloc():

```
p = (char *)malloc( 200 );
free( p + 10 );
- or -
int * x;
free( x );
```

C: Redundant free()ing of the same pointer:

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

D: Saturation of dynamic memory:

p = (char*)malloc(5001);

- or -

p = (char*)malloc(5000);

q = (char*)malloc(1);

… your code must gracefully handle being asked for memory than it can allocate without exploding.

2. Responding to Detected Errors

Your modified malloc() and free() should report the precise calls that caused dynamic memory problems during program execution. Your code should use the preprocessor LINE and FILE printf directives to print informative messages:

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

3. Testing and Instrumentation

After you are sure your code compiles and operates, you should test and profile your code. Writing code that works on basic test cases is nice, but in order to have useful code that you can trust, you must test it thoroughly and understand how your design decisions affect its operation. To this end, you will generate a series of workloads to test your implementation. Write a test program, memgrind.c, that will exercise your memory allocator under a series of the following malloc()/free() workloads:

A: malloc() 1 byte 3000 times, then free() the 3000 1 byte pointers one by one

B: malloc() 1 byte and immediately free it 3000 times in a row

C: Randomly choose between a 1 byte malloc() or free() 6000 times
- Keep track of each operation so that you eventually malloc() 3000 bytes, in total
- Keep track of each operation so that you eventually free() all pointers

D: Randomly choose between a randomly-sized malloc() or free 6000 times
- Keep track of each malloc so that your mallocs do not exceed your memory capacity
- Keep track of each operation so that you eventually malloc() 3000 times
- Keep track of each operation so that you eventually free() all pointers

E,F: Two more workloads of your choosing
- Describe both workloads in your testplan.txt

Run all of the workloads above (including yours) 100 times, while timing each workload. You might find the gettimeofday(struct timeval * tv, struct timezone * tz) function in the time.h library useful. Your memgrind.c should calculate the mean time for execution of each workload over the 100 executions and output it.

You should run memgrind yourself and include its results in your readme.pdf. Be sure to discuss your findings, especially any interesting or unexpected results.

4. Submission

You should submit a Asst1.tar.gz containing:

A: readme.pdf documenting your design and workload data and findings

B: testcases.txt that describes your two workloads and why you included them

C: mymalloc.h with your malloc headers and definitions

D: mymalloc.c with your malloc function implementations

E: memgrind.c with your memory test and profiling code as described above

F: Makefile that builds and cleans memgrind with your mymalloc library

5. Grading

A: Correctness - how well your code operates

B: Testing thoroughness - quality and rationale behind your test cases

C: Design - how well written and robust your code is,
including modularity and comments

D: Analysis - your analysis and documentation of results in your readme.pdf