CS 214 Homework 4

Fall 2021

Introduction

In this assignment, you'll implement a custom version of malloc, free, and realloc. You should implement these in mymalloc.c and mymalloc.h. These files should not contain a main function, so you may want a separate .c file for testing. You'll also write a performance testing program in memperf.c.

Library functionality

You should implement these functions, described below:

```
void myinit(int allocAlg);
void* mymalloc(size_t size);
void myfree(void* ptr);
void* myrealloc(void* ptr, size_t size);
void mycleanup();
double utilization();
```

• myinit(allocAlg)

Create a 1 MB "heap" and perform any other initializations your code needs. You can assume any application using your library will call this first.

The allocAlg argument describes what algorithm to use to find a free block (see Lec. 17, p. 19):

- 0: first fit1: next fit2: best fit
- mymalloc(size)

From the "heap", allocate a region of at least the requested size and return a pointer to the beginning of the region. If it cannot be allocated, return NULL.

All returned addresses must be 8-byte aligned. That is, the region you allocate should start at an address that's divisible by 8.

If size is 0, mymalloc does nothing and returns NULL.

• myfree(ptr)

Mark the given region as free and available to be allocated for future requests. It should be coalesced with adjacent free regions.

You should maintain an explicit free list.

If ptr is NULL, myfree does nothing.

You should detect a number of error conditions:

- trying to free an invalid address

If the address given isn't in your "heap", print "error: not a heap pointer". (For example, if your "heap" ran from 0x4000 to 0x6000, but the ptr given to myfree was 0x10.)

If the address given is in your "heap" but wasn't returned by mymalloc, print "error: not a malloced address". (For example, if mymalloc returns 0x4040, but the ptr given to myfree was 0x4041.)

- double free

If the address given was returned by mymalloc but has already been freed, print "error: double free".

• myrealloc(ptr, size)

Reallocate the region pointed to by ptr to be at least the new given size. If this cannot be done in-place, a new region should be allocated, the data from the original region should be copied over, and the old region should be freed.

If the reallocation can't be done, return NULL.

If ptr is NULL, this is equivalent to mymalloc(size).

If size is 0, this is equivalent to myfree(ptr) and myrealloc returns NULL.

If both ptr is NULL and size is 0, myrealloc does nothing and returns NULL.

• mycleanup()

Free the 1 MB "heap" and perform any other cleanup your code needs. You can assume any application using your library will call this last.

Your library should support "resetting" everything by calling mycleanup followed by myinit, since the performance testing program will need to do this.

• utilization()

Calculate space utilization as a ratio of memory used vs. space used in the "heap". For example, if you called mymalloc(50) followed by mymalloc(64) and determined that you needed 128 bytes on the "heap" to satisfy these requests, the ratio would be

$$(50+64)/128 \approx 0.89$$
.

Blocks that have been freed should not count towards memory used. The space used in the "heap" is the distance from the beginning of the "heap" to the end of the last allocated block.

C standard library usage

Outside of myinit and mycleanup, you should not call the standard malloc, calloc, free, or realloc functions.

Performance

You should write another program, memperf.c, that uses your library and attempts to measure its performance. It should randomly allocate, reallocate, and free blocks, and then analyze throughput and utilization for each allocation algorithm (call mycleanup to finish testing one algorithm and myinit again to start another). Blocks should be randomly sized, say from 1 byte to 256 bytes.

Utilization is defined above in the description of the utilization() function. Throughput should be measured as number of mymalloc / myrealloc / myfree operations per second. You may need to tweak the number of operations and block size to get reliable results without memperf taking too long to run or exhausting the available memory in your "heap". About 1,000,000 operations may be a good starting point.

Some possibly useful functions: rand, srand, time, gettimeofday.

The output format should be as follows:

\$./memperf

First fit throughput: 7812.5 ops/sec

First fit utilization: 0.85

Next fit throughput: 15625 ops/sec

Next fit utilization: 0.9

Best fit throughput: 3906.25 ops/sec

Best fit utilization: 0.92

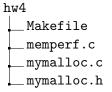
Compiling and testing

You should create a Makefile so that running make or make all builds your program (i.e., memperf). You should use similar CFLAGS to gcc as in previous homeworks, e.g.:

```
-g -Wall -Wvla -fsanitize=address
```

Submission

If you develop on your local machine, please be sure to test your code on ilab before submitting. Please submit the assignment on Canvas as a tar file hw4.tar that, when expanded, produces a hw4 directory (possibly with additional .c/.h files):



If you work in a group, please include a README.txt file with the names and netIDs of everyone in your group. Only one person needs to submit on Canvas.