CS 214: Systems Programming
Assignment 1: ++Malloc

0. Abstract

C allows you great freedom and control in your use of memory. It allows you to dynamically request heap space with the malloc() function and the ability to directly access any of your memory with a pointer. Malloc() requires no type information, but only the number of bytes you want to use, enabling you to store anything you want in that space. This is a fundamental departure in concept and function from many object-oriented languages where a 'type' is an elemental, Hermetic atom. In C a 'type' is a length and a format, but little else. You can freely store whatever information you want in whatever memory you own. This project will illustrate that directly, as you will build malloc for your self and go on to enhance free() to be more intelligent.

1. Introduction

In this assignment, you will implement your own version of the system's malloc() and free() that not only provide the same basic functionality, but also detect common programming and usage errors.

Malloc(size_t size) is a system library function 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[4096]). 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. Your malloc() function should use a “first fit” algorithm to select blocks of memory to allocate.

2. Methodology

2.1 mymalloc() definition

You'll be implementing your malloc as a macro in a user-defined library. You should write a 'mymalloc.h' library file that has:

- A macro that will replace all calls to 'malloc(x)' with calls to 'mymalloc(x)' and free(x) with myfree(x)
- A function signature for your mymalloc(x) code in 'mymalloc.c'
- A definition for a static array of size 4096 to allocate from

2.1.1 mymalloc() implementation:

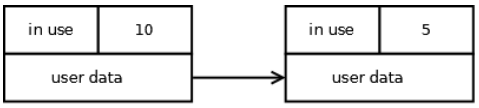
Your mymalloc code needs to perform the same basic function as malloc(), namely returning a pointer to at least as many byte as asked for, and NULL if that is not possible. You can only allocate space out of your large, static byte array. You need to keep track of how much and what other memory you have allocated before, but you can not require the user to hand you that information. You also can not use malloc() within mymalloc(). Since you need metadata that persists between calls to mymalloc() and you can not use dynamic memory to move data outside of your local scope on to the heap, the only memory that remains is - your big array. You will need to store organizational metadata in your static byte array alongside the space that you are setting aside for the user.

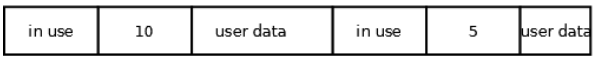
2.1.2 mymalloc() setup

Be careful when writing your mymalloc() code. It must be able to handle being invoked any number of times. It should not presume metadata structures are present in your large static array since the first time it is invoked, there will be no initialization. Be careful that your mymalloc() can detect if this is the first time vs the not-first-time it has been run.

2.1.3 mymalloc() metadata

You will need to store your metadata alongside space you set aside for user data since you do not have access to any other persistent memory. You can use any kind of metadata, but additional credit will be available for being space efficient (see sect.6). It is recommend that you view your structure as a flattened linked list, a linked list whose 'nodes' are all separate blocks of memory that just so happen to be contiguous:





2.2 myfree() definition

Your library should also have a definition for myfree(), since you alone will know how to deallocate space that you have allocated. Given how easy it is to make free() crash, you'll also be improving your own version to detect and deal with these common issues.

3. Detectable Errors

3.1 Free()

Your myfree() implementation should be able to catch at least the following errors:

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

```
int x;
free( (int*)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.
```

3.2 Malloc()

Your mymalloc() implementation should be able to catch at least the following errors:

A: Saturation of dynamic memory:

```
p = (char*)malloc(4097);
- or -
p = (char*)malloc(4096-(sizeofmetadata));
q = (char*)malloc(1);
  ... your code must gracefully handle being asked for more memory than it can
    allocate.
```

3.3 Responding to Detected Errors

Your mymalloc() and myfree() 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__ )
```

4. 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 and immediately free it - do this 150 times

B: malloc() 1 byte, store the pointer in an array - do this 150 times.
Once you've malloc()ed 50 byte chunks, then free() the 50 1 byte pointers one by one.

C: Randomly choose between a 1 byte malloc() or free()ing a 1 byte pointer
> do this until you have allocated 50 times
- Keep track of each operation so that you eventually malloc() 50 bytes, in total
> if you have already allocated 50 times, disregard the random and just free() on each iteration
- Keep track of each operation so that you eventually free() all pointers
> don't allow a free() if you have no pointers to free()

D: Randomly choose between a randomly-sized malloc() or free()ing a pointer – do this many times (see below)
- Keep track of each malloc so that all mallocs do not exceed your total memory capacity
- Keep track of each operation so that you eventually malloc() 50 times
- Keep track of each operation so that you eventually free() all pointers
- Choose a random allocation size between 1 and 64 bytes

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

Your memgrind.c should run all the workloads, one after the other, 100 times. It should record the run time for each workload and store it. When all 100 iterations of all the workloads have been run, memgrind.c should calculate the mean time for each workload to execute and output them in sequence. You might find the gettimeofday(struct timeval * tv, struct timezone * tz) function in the time.h library useful.

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.

5. Submission

You should submit a Asst1.tar.gz containing:

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

B: testplan.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: a Makefile that builds and cleans memgrind with your mymalloc library

6. 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

E: Space Efficiency - up to 15 additional points, based on proximity to minimal metadata size possible. Be sure to detail your metadata implementation in your documentation to be eligable.

7. Groups

You must have a group of two for this project that is documented on the course's Google spreadsheet. You can find a link to it in the Sakai announcements. If your NetID does not appear on that list, you will not receive a grade for this assignment.