Programming Project #3

Allocating some Memory

Introduction

After completing this assignment you will learn about dynamic memory (heap) allocation strategies and how to write tools that intercept calls from other programs. Your task is to write special versions of `malloc()` and `free()` that check for common allocation problems and mistakes. You'll test your functions with example buggy programs.

Your tasks:

- Write `slug_malloc()` and `slug_free()` functions that wrap regular `malloc()` and `free()`.
- Add extra checks to your functions to detect common problems.
- Write `slug_memstats()` that prints out all allocated memory and some statistics.
- Add a leak test facility for detecting memory leaks.
- Test your implementation on buggy programs, write more test programs.
- Clearly explain what errors your allocator is guaranteed to find, which errors it usually finds, and which errors it does not find.

Memory Allocation

C programs typically allocate and deallocate memory on the heap using the `malloc()` and `free()` function calls. These functions are implemented in the C standard library by calling lower-level memory allocation functions that the OS provides, such as `brk()` or `mmap()`. Decoupling application memory allocation from the direct OS mechanisms provides a nice layer of abstraction and allows more features and opportunities for optimization. Programmers don't always want 4KiB blocks of memory, they want the ability to allocate and free arbitrary sized chunks of memory.

Even though `malloc()` and `free()` provide more flexibility than mapping and unmapping entire blocks, the C programmer still is doing manual memory management which is a common source of bugs. In this project you are making replacement `malloc()` and `free()` functions that attempt to detect and report as many errors in the calling program as possible.

Your replacement functions will not implement allocation algorithms themselves. Instead you will use normal `malloc()` and `free()` to implement your new functions. The difference is that your functions will keep track of more information in new data structures. You will also allocate more memory than requested directly and return a pointer somewhere inside the region you allocate.

Users of your alternate allocator will need to recompile their program including a special header file that you provide. Your header file should make it unnecessary for users to replace actual `malloc()` and `free()` calls in their program; they should only have to add your include file and recompile.

Goals

Here are the goals that your allocator should satisfy.

- Correct programs should remain correct and should allocate/deallocate memory as normal, with the possible difference of being slightly slower or using more memory.
- If `slug_free()` is called on memory that was not allocated with `slug_malloc()` then an error should be shown and the program terminated.
- If `slug_free()` is called on memory that is not the first byte of memory allocated by `slug_malloc()`, a message should be shown and the program terminated.
- If `slug_free()` is called on memory that was already freed by `slug_free()`, an error should be shown and the program terminated.
- When the test program exits, any allocations that were made and not freed should be shown in a table.

Details

Here are the functions you need to write.

```
void *slug_malloc ( size_t size, char *WHERE );
```

This function will allocate memory by calling `malloc()`. It returns the address of allocated memory. In addition, it records the address, length, current timestamp, and location of the call in an internal data structure. If size is zero, this is not an error but should be reported on `stderr` as an unusual operation. If the input is excessively large (more than 128 MiB) then this function should report an error in `stderr` and terminate the program. The parameter `WHERE` is a string constant that records the filename and line number of the caller.

```
void slug_free ( void *addr, char *WHERE );
```

This function first checks that the `addr` is the start of a valid memory region that is currently allocated by looking through the internal data structures. If not, an errors is shown and the program terminated. If it is valid then `free()` should be called and the internal data structures updated to indicate that the address is no longer actively allocated.

```
void slug_memstats ( void );
```

This function traverses the internal data structures kept by `slug_malloc()` and `slug_free()` and prints out information about all current allocations. Each allocation report should include the size of the allocation, a timestamp for when the allocation took place, the actual address of the allocation, and file and line number in the test program where the allocation happened.

In addition, a summary of all allocations should be reported that includes the total number of allocations done, the number of current active allocations, the total amount of memory currently allocated, and the mean and standard deviation of sizes that have been allocated.

Header

You will also need to create a C header file for users to include in their programs. Start by requiring users to explicitly call `slug_malloc()`, `slug_free()`, and `slug_memstats()`. Once that is working, include C preprocessor macros that replace normal `malloc()` and `free()` calls with `slug_malloc()` and `slug_free()`.

Here is a snippet of preprocessor macros that may be helpful:

```
#define malloc(s)         slug_malloc((s))
```

To get the file and line number you might find the following macros helpful:

```
#define FUNCTIONIZE(a,b)  a(b)
#define STRINGIZE(a)      #a
#define INT2STRING(i)     FUNCTIONIZE(STRINGIZE,i)
#define FILE_POS          __FILE__ ":" INT2STRING(__LINE__)
```

You will need to add an argument to slug_malloc() and slug_free() to pass in FILE_POS.

```
#define malloc(s)         slug_malloc((s), FILE_POS)
```

When compiling test programs, you will need to link in the object file that includes your functions. You will also need to include your special header file in the source code of the test program.

Leak Detection

The idea is that users will compile their program using your header and library. After doing any allocations, your library will register a function to be called at normal program exit using `atexit()`. When the test program terminates, your handler will print out all the memory that is still allocated that was not properly freed using the `slug_memstats()` function. Memory that was allocated and not freed by the end of the program is often a memory leak.

Your library functions should keep track of whether an exit handler has been installed (starting at "no"). If any allocation function is called and the hander has not been installed you should install your custom exit handler. Your handler should only be installed once. Your handler should check whether memory is allocated and display an appropriate warning message if any memory is still allocated, then call `slug_memstats()` to show a table of allocations.

Testing

Your testing should be able to:

- Show a test program that correctly uses allocation in a non-trivial way behaves correctly.
- Show that after allocating and deallocating memory, trying to deallocate an invalid address is immediately detected.
- Show that after allocating and deallocating memory, trying to deallocate an already freed region is immediately detected.
- Show that after allocating and deallocating memory, trying to deallocate a valid region by passing in a pointer inside the region is immediately detected.
- Show that allocating memory and then exiting triggers the leak detector and shows where the leak occurred.
- Experiment with other memory allocation mistakes to see what you are able to detect.

Your testing strategy and writeup of your results should be included in your design document for this project, together with a description of the design of your implementation.

You may want to use random numbers to do random allocations and deallocations. In this case you should use a fixed constant seed so you can reproduce interesting test cases.

It may be convenient to do development in Linux; your library should be essentially the same between Linux and MINIX.

Deliverables

Turn in a `.tgz` file for your project directory into eCommons. It should include:

- Build and run instructions in `README.txt`
- Your design document (including test plan and result writeup), named `design.pdf`
- Source files (no generated files!)
- Any extra build files (e.g. `Makefile`)
- Any test files

Your project must be submitted in this format or you will be asked to fix your submission and resubmit (using grace days or getting a penalty). If we try to follow the instructions in your `README.txt` file and it doesn't work, you will be asked to resubmit (using grace days or getting a penalty). We will try your submissions in a fresh MINIX 3.1.8 (or MINIX 3.2.1 installation if you specify that in your instructions). Don't forget to include any files needed. Also, do not include files that could be generated automatically from source files (each such file included will be penalized).

For building, we would prefer to type one command to build everything. This might involve copying files from your directory into `/usr/src` and running commands such as `make install` from the `/usr/src/tools/` directory.

Coding Style

All code submitted must follow these guidelines. Deviations may be penalized. Unclear or sloppy code will always be penalized. Good coding style makes your code more readable to others and is critical for keeping complex software maintainable.

Comments

- Every file must contain a comment on top that indicates whether you created it, or it was part of MINIX and you modified it. If you created the file, put CREATED in the command and a date. If you modified an existing MINIX file, put CHANGED in the comment with a date.
- Every function must have a comment immediately before it that describes what the function does. It should describe the parameters, return value, intended action, and any assumptions the function makes.
- Code should be well commented. Trickier parts of code need more comments, simple parts need fewer comments. You should end up with about as many lines of comments as you have lines of code.
- If you changed an existing MINIX file, put comments that indicate where the change starts and ends. Put “CHANGE START” and “CHANGE END” in these comments.

Indentation

- Only use spaces, no tabs.
- Indent a fixed number of spaces at every level (e.g. 2 or 4).
- Wrap long lines manually in a visually reasonable way.

Spacing

- Every significant comment should be preceded by a blank line. Do not put a blank line after comments before the code that is being described.
- Put a blank line between chunks of code that do different things.
- Put a blank line between variable declarations and other code.
- Put a blank line after every right brace unless the next line is also a right brace.

Braces

- For function declarations, put the opening brace in column 0.
- For other structures, put the left brace on the same line as the keyword (if, while, etc.). Put the right brace to align with the opening keyword (if, while, etc.)

Identifiers

- Functions and variable names should be all lowercase, with underscores between words.
- Names should be descriptive without being too long (this is an art).
- Single letter names may only be used as trivial loop counters.

Global Variables

- Global variables may not be used unless they are the correct solution to a particular implementation issue.
- Any global variables must be explained in the design document.

(This assignment was inspired by an assignment by Barton Miller at the University of Wisconsin for CS 537.)