The George Washington University

Advanced Operating Systems - Spring 2015

Programming Exercise - Slab-based Memory Management

Due January 26th, 2015 at midnight.

1 Objective

You will implement a slab allocator for memory management. You will learn how it organizes data, and optimizes for apriori-known allocation sizes. You will optimize your implementation to be lightning-fast. Please note that you will *have to learn* about the slab allocator and how it works before you will be able to do this assignment. This will mainly be done by reading the original slab allocator paper linked on Piazza. This course requires you to learn quite a bit on your own. Please feel free to use the message board to understand any generic issues about these APIs.

This assignment is hard, and time consuming. Please start early.

Assignments *after* this one will be comparably difficult if not more difficult as there will be no online references, and they are more intensive. If you find this assignment unreasonably challenging, please consider that the drop date is a week from the first day of classes.

If you have trouble with `C`, please see the tutorials on the course webpage.

This assignment is not to be done in groups. You can discuss the ideas with others, but you cannot exchange or see each other's code. See the Academic Honesty policy below.

2 Assignment

For this homework, you will implement a slab allocator. Find the original paper linked to on Piazza. This paper is not very clear, and you will want to seek out other resources to learn more about slab allocators, and ask questions on Piazza. However, you must still implement the API and general design introduced in this paper. Through this homework, you will learn: The importance of *slowly* writing and thinking about code. The utility of assertion statements. The importance of paying attention to performance in code design. How to write systems code in C, if you didn't already have experience with it.
Please see the academic honest section below. You are *not allowed to copy other's code* either from classmates or from the Internet. All of the code must be your own. For this assignment, you are *not* allowed to work in groups. You must implement this in C, and provide your own thorough test cases. You must use proper structuring of your project into `*.c` and `*.h` files. The quality of your code matters. Please focus on simplicity, and if a section of your code is becoming complex, do back up and simplify. That said, the fundamental operations of allocation and deallocation should not take more than around 100 cycles, so performance matters too.
Details:

- Implement the API from Section 2.3 with the following exception: You don't need to implement constructors and destructors, so they don't need to appear in you cache allocation function. `kmem_cache_grow` can simply use `malloc`, and `kmem_cache_reap` can use `free` (described in 3.1).
- You are free to ignore what the paper calls "memory pressure" which is essentially the system asking for memory back. Instead, and unlike what section 3.4 suggests, you can free a slab immediately when no objects in it are allocated.
- It is essential to understand what a freelist is, and how it is used in the slab allocator. For example, you should understand that in the common case where there are cached objects available, allocation and free should be O(1) operations.
- You should only use malloc and free when allocating new slab, and *not* when you allocate or free a new object. To do this, you must understand that the freelists can be held within object memory itself, as, by definition, that object memory is unused if it is on a freelist.
- You can ignore section 4.
- It will help to read about malloc implementations, and understand how memory allocators are implemented in general. One reasonable implementation is dlmalloc, that is quite googleable.

A concise, fully functional implementation is possible in around 200 lines of code. If you are going much beyond that, then ask if you are making it more difficult than it needs to be.

3 Testing

You must write your own battery of tests in a `main.c` file. Please test all edge cases. You *are* allowed to share these unit tests with other students, but not any of the slab implementation.

4 Submission

You must create a `Makefile` to build you code, and you must include the `clean` directive to clean up your directory (remove temp files and `.o` files). You will create a **tarball** (.tar.gz, or .tgz) archive containing **your entire slab directory** (please run a `make clean` before zipping the directory to remove any build products) and you will submit this tarball file to Blackboard in the folder for Programming Assignment 1 by the appointed time indicated at the top of this document. The tarball file will be named with the following naming convention:

```
csci_6411_ws_<your GWNet Id>.tgz
```

Please replace `<your GWNet Id>` with your user name that you use to log into Blackboard.

5 Evaluation and Grading

You are obligated to complete this exercise on your own by the Code of Academic Integrity (`http://www.gwu.edu/~ntegrity/code.html`). You **cannot use code that you find online.**

You cannot look at the source code of another student, or show them yours. Any sources that help you must be acknowledged in your `README` file (e.g. `man` pages).

The conciseness and simplicity of your code will affect your evaluation. Find a balance between the number of lines of code and the simplicity of each line.

Your code will be evaluated in terms of the comprehensiveness and descriptiveness of comments. A block comment preceding the function definitions you implement should describe the function including any special information about the input parameters and output result. Line commments should clarify variable usage and complex logic. Do not assume that either you or others will understand code that you have written.

Your code will be tested on Linux, and it should not use specialized APIs that are only present in Linux versions after 2.6.33.