Lab 5: Slab Memory Allocator

IUPUI CSCI 503 – Fall 2013
Assigned: November 4, 2013
Due time: 11:59PM, November 18, 2013

1. Introduction

Instead of using Libc's malloc and free, in Lab 5, you will design and develop your own memory allocator, which is called slab memory allocator. This slab memory allocator provides three functions to users:

- *void** ***alloc_mem****(****struct slab_allocator***** alloc, int num_bytes);*
  - It returns a NULL pointer if there is an error.
- *int* ***free_mem****(****struct slab_alloctor***** alloc, void* ptr);*
  - On success, it returns 0. Otherwise, it returns -1.
- void **init_slab_allocator**(**struct slab_allocator*** alloc, size_t mem_pool_size);
- Certainly you need to define struct slab_allocator { … };

Slab: A slab is a contiguous memory region. In Lab 5, the size of each slab is always 1M bytes.

Chunk: Given a slab, you chop it into a number of blocks. Each block is then called a "chunk". All chunks belonging to the same slab are of the same size (except for the last chunk if SlabSize is not divisible by ChunkSize).

Slab Class: A slab class stores a set of equal-size chunks. The slab class can be implemented as a doubly linked list. Each node is a chunk. There are a number of slab classes. Different classes store chunks with different sizes. For instance, Class A stores chunks of m bytes, Class B stores chunks of n bytes.

Slab Allocator: A slab allocator is a data structure that contains a number of N slab classes. It also stores information related to the slab allocator such as the number of classes, a memory pool, and so on.

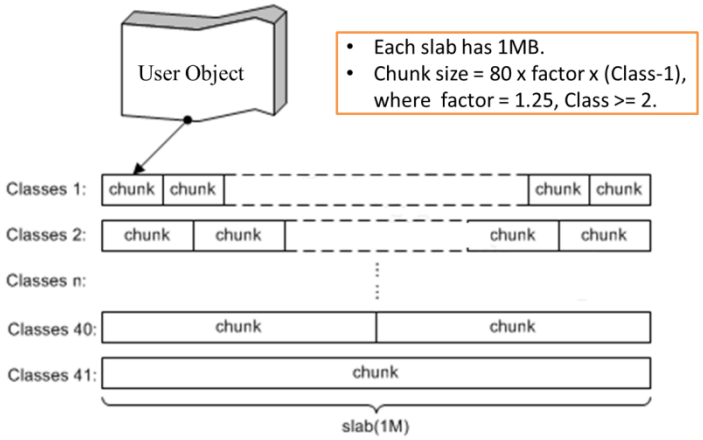

Figure 1. Slab Memory Allocator

Figure 1 shows the overall structure of the slab allocator. In this lab, **the first class has a chunk size of 80.**

How **init_slab_allocator()** works?

- First of all, the slab allocator has a pool of 1MB slabs. It is pre-allocated by calling malloc() just **once**. The size of the memory pool is specified by parameter *mem_pool_size* (e.g., 128 MB). The pool can be implemented as a doubly linked list, where each node is a 1MB slab. **You can only use malloc inside the function init_slab_allocator() and when you need to create a slab_allocator structure.**
- Next, you create a table (or array) of classes (see Figure 1). The chunk size in each class is decided by the formula in Figure 1.
- The init_slab_allocator() function needs to populate the class table by giving one slab to every class. For instance, 10 classes will use 10 slabs. Note the assigned slab must be chopped to chunks first before being linked to a linked list.
- Figure 1 shows the initial assignment of slabs for each class. Note that it is possible that more slabs can be assigned to each slab class, if the free chunks of that particular slab class are empty.

How **alloc_mem()** works?

- Based on the given #bytes, find an appropriate slab class, then remove a linked list node from the slab class. The address the function returns must be a multiple of 8. You need to think through how you can guarantee the address is divisible by 8.
- When the slab class becomes empty, go to the allocator's memory pool to fetch a new slab. Then chop the new slab into a number of chunks. With these new chunks, you are able to return space to a user. If the memory pool is empty, print an error message and exits gracefully.

How **free_mem()** works?

- Based on the given pointer, you need to find out which slab class this memory block originally came from. Then add the block to the slab class's linked list.

2. Structure of Your Source Code

There will be four files:

1) slab_allocator.h: this file stores slab_allocator related data structures and function declarations. The interface must be exactly the same as that defined in Section 1.
2) slab_allocator.c: this file stores the implementation of slab_allocator related functions.
3) dll.h: data structure for doubly linked list.
4) test.c: main function that uses your slab_allocator to build, traverse, and delete double linked lists.

Note that TA will compile his own test program with your slab_allocator.h and slab_allocator.c. So you should be able to send slab_alloctor.h and slab_alloctor.c to anybody and he or she can use your allocator directly.

3. Score Distribution

- Coding style and error checking: **10%** (as described in the "Labs Grading Policy")
- **10%**: You slab_allocator.h, slab_allocator.c can be compiled and linked with TA's code correctly.
- **30%**: Pass the tests that your slab allocator can be used to create new nodes to create a list. A list node may have different sizes.
- **20%**: Pass the tests that your slab allocator can be used to delete a list and free the list nodes correctly.
- **10%**: Pass the tests that your slab allocator can be used to new/delete nodes dynamically. That is, mixing new and free together dynamically.
- **10%**: The pointer returned by alloc_mem() is a multiple of eight.
- **10%**: If a slab class has no space left, your function should fetch a new slab from the memory pool and continue working. In case the memory pool is empty, print an error message.

If there is a "Segmentation Fault" when testing your code, then TA will deduct 20 points from your score.

The total score is 100 points.

4. Deliverable

You should submit source code in C, a Makefile, and a README. Make a tar ball and send it to the TA. The tar ball name may be "Lab5_your_name.tar". Note that only slab_allocator.h and slab_alloctor.c will be used by our TA.

5. How TA will Test Your Code

TA will use your slab memory allocator in his own doubly linked list program and test if his program still works or not. These are a few test examples:

- Create a doubly linked list. The node size can be any size between 8 and 1000 bytes. Different sizes will be tried.
- Traverse the list and assign values to each node. After assign values to a list, the program then checks whether the stored values are correct or not.
- Dynamically new (insert) nodes, and delete (free) nodes. Then traverse the list to check whether the remaining list nodes are correct or not.
- …

Hint: to test your code by yourself, you could try various list-node sizes, try to do a number of INSERT(alloc), then do a number of DELETE(free), try to mix alloc()'s and free()'s together in a for loop,

try to exhaust a slab class and trigger allocating memory from the memory pool. You may also ask your fellow graduate students to see how they test their codes.