Project: Memory Allocator in C

CS 620: Operating Systems Fundamentals

March 2024

1 Introduction

This project implements a simple memory allocation library in C, similar to malloc. Please review Chapter 17 of the book for more details on free space management. Our library will have two main APIs: `void * mem_allocate(int size)` and `mem_free(void * pointer_to_free)`. You are free to implement the free space management using any data structure(s) you want, however, I'd carefully read the book and stick with something from the text. Also remember, in your memory allocation library, you are not allowed to use `malloc()` or `calloc()` functions since you are implementing them.

2 Requirements

2.1 API

The memory allocator must follow the following API:

- `int init(int heap_size)` – this function initializes the allocator with a heap of `heap_size` bytes. Note that the actual heap size may be different than the `heap_size` specified, as we can only ask OS to give memory in pages, so the actual heap size is a multiple of a page size. If `heap_size` is not a multiple of a page size, round up the heap size to the nearest multiple of the page size. Repeated calls are not allowed, and the `init()` function should not re-initialize the heap. The function should return 1 if successfully initialized, and 0 if called after a previously successfully initialization and -1 in case of any other errors.
- `void * mem_allocate(int size)` – this function returns a pointer to the newly allocated memory block of a given size. If the allocator cannot find enough memory, it will return NULL.
- `void mem_free(void * pointer_to_free)` – This function frees the memory block pointed to by the `pointer_to_free`. Just like with a regular malloc library's `free()` call, the `pointer_to_free` must be a correct pointer previously given by the `mem_allocate()`, otherwise the call to `mem_free()` does not guarantee correct behavior (in other words, if passed a bad pointer, you library is allowed to mess up).

Your library should compile to a dynamic library file for Linux environment.

2.2 Implementation

For the allocator library to work, it needs to have an initial free memory block given to it by the OS. We will use mmap system call to get that initial memory block. For our allocator, **we will not support** growing allocations – if more memory is needed than initially initialized, the allocator should return NULL in the `mem_allocate()` call.

The `mmap` system call provides the functionality to map files to memory. In our case, however, we will use the file-less `mmap`, so no copy of the data gets written to storage. Below is an example of how to use `mmap` system call:

```
...
int fd = open("/dev/zero", O_RDWR);

// size (in bytes) needs to be evenly divisible by the page size
int s = numPages * pgSize;
void *ptr = mmap(NULL, s, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (ptr == MAP_FAILED) {
    perror("mmap");
    return -1;
}
...
```

The pointer returned by `mmap` points to the block of free memory for your allocator. After this, you are using this block to perform all allocations. Note, this is just a block of memory, and initially it constitutes all of your free space. However, you need to track your free spaces as allocations and de-allocations occur. Please refer to the Chapter 17 in the book on more info on free-space management. It is the core of this project to implement such free-space management for your allocator. One important caveat here is that **you are not allowed** to use `malloc` or `calloc` or any other memory allocation utility to create objects on the heap for the data structures you need to manage the free-spaces. **All of the data structures must be implemented within the confines of your heap created in the `init()` call.**

2.3 Finding the Appropriate Free Space

You allocator should find a suitable block of memory for the allocation. If it does not have a suitable free space, the `mem_allocate` should return NULL. As your library allocates and frees the memory, the free-list may become fragmented. You may need to coalesce the free spaces whenever possible to avoid the fragmentation.

3 Testing

Make sure your library can be used as a dynamic library. You must provide a make file to compile your code into a dynamic library. If make file is missing or does not compile your code, you will receive a 30% deduction on your grade. I will test your library against my testing code. Your grade will largely depend on how well your allocator passes my tests. I am not disclosing the entire test suit, however, below you can find a hint of some testing scenarios:

- init() the allocator and allocate an object with a size of half of the initialized heap. Free the memory.
- init() the allocator and allocate N small objects. Free $\frac{N}{2}$ objects, for some random N within a reasonable range. Repeat until heap is full and allocation fails while changing N in each repetition
- init() the allocator and allocate N small objects. Free all. Allocate n large objects, for some reasonable numbers $N >> n$.

Remember, failing to allocate memory may be ok, as it may indicate running out of heap space. However, bad free space management can lead to inability to allocate memory sooner than in the ideal/optimal allocator.

4 Grading

1. Correctness – 50%
    - Whether you allocator is correct and returns usable memory when allocated, frees memory when deallocating and allows the reuse of freed memory.

2. Performance – 30% – If your code is not correct, you will automatically get 0 on performance aspect of the grade
    - Whether your allocator is reasonably fast (10%). This is a somewhat subjective metric, but I do not want your allocations to take forever.
    - Whether your allocator can manage free spaces well and avoid (excessive) fragmentation (20%). This one is more concrete – I will have tests that evaluate this metric.
3. Report – 20%
    - You are to write a report describing your implementation, data-structures and algorithms you used. Please also describe your testing strategy and (interesting) testing cases you have used. Please keep the report between 2 and 4 pages of 12-point font and 1-inch margins.