BCIT

COMP 4736 **Lab 06: Dynamic Memory Management**

In this lab, you will understand the principles of memory management by building a custom memory manager to allocate memory dynamically in a program. Specifically, you will implement functions to allocate and free memory, that act as replacements for C library functions like `malloc` and `free`.

Before You Begin

Understand how the `mmap` and `munmap` system calls work. In this lab, you will use `mmap` to obtain pages of memory from the OS, and allocate smaller chunks from these pages dynamically when requested. Familiarize yourself with the various arguments to the `mmap` system call.

Building a Simple Memory Manager

In this part of the lab, you will write code for a memory manager, to allocate and deallocate memory dynamically. Your memory manager must manage 4KB of memory, by requesting a 4KB page via `mmap` from the OS. You must support allocations and deallocations in sizes that are multiples of 8 bytes. The header file `alloc.h` defines the functions you must implement. You must fill in your code in `alloc.c`. The functions you must implement are described below.

- The function `init_alloc()` must initialize the memory manager, including allocating a 4KB page from the OS via `mmap`, and initializing any other data structures required. This function will be invoked by the user before requesting any memory from your memory manager. This function must return 0 on success and a non-zero error code otherwise.

- The function `cleanup()` must cleanup state of your manager, and return the memory mapped page back to the OS. This function must return 0 on success and a non-zero error code otherwise.

- The function `alloc(int)` takes an integer buffer size that must be allocated, and returns a `char*` pointer to the buffer on a success. This function returns a NULL on failure (e.g., requested size is not a multiple of 8 bytes, or insufficient free space). When successful, the returned pointer should point to a valid memory address within the 4KB page of the memory manager.

- The function `dealloc(char *)` takes a pointer to a previously allocated memory chunk, and frees up the entire chunk.

It is important to note that you must NOT use C library functions like `malloc` to implement the `alloc` function; instead, you must get a page from the OS via `mmap`, and implement a functionality like `malloc` yourself. The memory manager can be implemented in many ways. So feel free to design and implement it in any way you see fit, subject to the following constraints.

- Your memory manager must make the entire 4KB available for allocations to the user via the alloc function. That is, you must not store any headers or metadata information within the page itself, that may reduce the amount of usable memory. Any metadata required to keep track of allocation sizes should be within data structures defined in your code, and should not be embedded within the memory mapped 4KB page itself.

- A memory region once allocated should not be available for future allocations until it is freed up by the user. That is, do not double-book your memory, as this can destroy the integrity of the data written into it.

- Once a memory chunk of size N_1 bytes has been deallocated, it must be available for memory allocations of size N_2 in the future, where $N_2 \leq N_1$. Further, if $N_2 < N_1$, the leftover chunk of size $N_1 - N_2$ must be available for future allocations. That is, your memory manager must have the ability to split a bigger free chunk into smaller chunks for allocations.

- If two free memory chunks of size N_1 and N_2 are adjacent to each other, a merged memory chunk of size $N_1 + N_2$ should be available for allocation. That is, you must merge adjacent memory chunks and make them available for allocating a larger chunk.

- After a few allocations and deallocations, your 4KB page may contain allocated and free chunks interspersed with each other. When the next request to allocate a chunk arrives, you may use any heuristic (e.g., best fit, first fit, worst fit, etc.) to allocate a free chunk, as long as the heuristic correctly returns a free chunk if one exists.

We have provided a sample test program `test_alloc.c` to test your implementation. This program runs several tests which initialize your memory manager, and invoke the `alloc` and `dealloc` functions implemented by you. Note that we will be evaluating your code not just with this test program, but with other ones as well. Therefore, feel free to write more such test programs to test your code comprehensively. It is important to note that none of the functionality or data structures required by your memory manager must be embedded within the test program itself. Your entire memory management code should only be contained within `alloc.c`.

Reminders

The header file `alloc.h` defines the functions that you need to implement. The test program `test_alloc.c` has included `alloc.h` so that it can call those functions. Your `alloc.c` will need to include the same header file, and implement the functions for the test program to call.

If you read the test program, you will see two types of includes.

```
// Compiler will search the <header file> from system path.
#include <stdio.h>

// Compiler will search the "header file" from current local path.
#include "alloc.h"
```

Note that to compile multiple source files into a single executable, you can use command like the following:

```
gcc source_1.c source_2.c -o executable -Wall
```

Header files (`.h`) only needs to be included in the source file (`.c`); it does not appear in the compilation command.

Submission Instructions

You should submit a single source file as `alloc.c`. You need not submit the testing code.

— End of Lab 06 —