Lecture Notes on Operating Systems

Lab: Dynamic memory management

Goal

In this lab, you will build 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. You will use `mmap` to obtain a page of memory from the OS, and allocate chunks from this page dynamically when requested. Familiarize yourself with the various arguments to the `mmap` system call.
- Read up on simple memory management strategies, e.g., how `malloc` manages free space on the heap, and so on.

Building a memory manager

In this 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. You must fill in your code in the files `alloc.h` and `alloc.c` provided to you. You may define any global data structures you need to keep track of memory information in the file `alloc.h`. This header file also defines the following four functions, which you must implement in `alloc.c`.

- The function `init()` 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. However, note that we do not expect you to perform optimizations like coalescing adjacent free chunks, in order to make a free chunk of size N_1 bytes available for future allocations of size greater than N_1 bytes.
- 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.

Testing your memory manager

You are provided two sample test programs `test_alloc1.c` and `test_alloc2.c`, along with a simple script `run.sh` to compile your code and run these tests. These test programs initialize your memory manager, and invoke the `alloc` and `dealloc` functions implemented by you. The first test program performs a few simple sanity checks on your memory manager, e.g., checking that it can perform simple allocations and deallocations, writing a string into the memory region allocated by your memory manager and reading it back to ensure its integrity, and so on. The second test program runs a few more complex test scenarios, including checking if your program can effectively reuse a freed up chunk and if it can split a free chunk into smaller chunks for future allocations.

Note that we will be evaluating your code not just with these test programs, 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 the files `alloc.h` and `alloc.c`.

Submission instructions

- You must submit the files `alloc.h` and `alloc.c`. You need not submit the testing code, as we may use different test programs during evaluation.
- Place this file and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.