

Assignment # 3: Building your own Memory Manager

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 assignment, 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](#).

Task-1 (Exploring `mmap`)

1. Write a simple C program (`T0_Test.c`) that runs for a long duration, say, by pausing for user input or by sleeping.
 - While the process is active, use the `ps` or any other similar command with suitable options, to measure the memory usage of the process.
 - Specifically, measure the virtual memory size (VSZ) of the process, and the resident set size (RSS) of the process (which includes only the physical RAM pages allocated to the process). You should also be able to see the various pieces of the memory image of the process in the Linux `proc` file system, by accessing a suitable file in the `proc` filesystem.
2. Now, add code to your simple program to memory map an empty page from the OS. For this program (and this lab, in general), it makes sense to ask the OS for an anonymous page (since it is not backed by any file on disk) and in private mode (since you are not sharing this page with other processes). Do not do anything else with the memory mapped page. Now, pause your program again and measure the virtual and physical memory consumed by your process. What has changed, and how do you explain it?
3. Finally, write some data into your memory mapped page and measure the virtual and physical memory usage again. Explain what you find.

Task-2 (Building a simple memory manager)

Goal: In this task, you will write code for a memory manager (`alloc.c`), 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.h`. 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.

Few Conditions to be careful about –

- DO 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.
- 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 $N1$ and $N2$ are adjacent to each other, a merged memory chunk of size $N1 + N2$ 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.

How to test: 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`. You can compile and run the test program using the following commands –

```
$gcc test_alloc.c alloc.c
$./a.out
```

Submission and Grading:

The assignment will be graded on following items:

1. Completeness and correctness on your programs for both tasks. (50%)
2. A report on your design process and appropriate evidence with **enough test scenarios** (in the form of screenshots) (20%)
3. Programs' readability and correctness (10%)
4. Give instruction on how to run your code (2%)
5. Describe the individual steps taken, and any hurdles faced while implementing (5%)
6. References (including links where you found some sample code. (3%)