Lecture Notes on Operating Systems

Lab: 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 chunks from these pages dynamically when requested. Familiarize yourself with the various arguments to the `mmap` system call.
- Write a simple C/C++ program 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.
- 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?
- Finally, write some data into your memory mapped page and measure the virtual and physical memory usage again. Explain what you find.

Part A: 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` or `alloc.cpp`. 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`.

You can compile and run the test program using the following commands (use g++ for C++).

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

Part B: Expandable heap

In this question, you will build a custom memory allocator over memory mapped pages, much like you did in the previous part of the lab. However, now your memory allocator should be "elastic", i.e., it should memory map pages from the OS only on demand, as described below. You are given the header file `ealloc.h` that defines 4 functions that your elastic memory allocator should support. You must implement these functions in the file `ealloc.c` or `ealloc.cpp`.

- The function `init_alloc()` should initialize your memory manager. You can initialize any datastructures you may require in this function. However, you must NOT memory map any pages from the OS yet, because you are supposed to allocate memory only on demand.

- The function `cleanup()` should clean up any state of your memory manager. It is NOT required to unmap any pages you memory-mapped from the OS here. We assume in this question that your elastic memory allocator expands by invoking the `mmap` system call when allocations are made, but does not return memory back to the OS via `munmap`.

- The function `alloc(int)` should take an integer buffer size that must be allocated, and must return a `char *` pointer to the buffer on a success. This function should return NULL on failure. You can make the following assumptions to simplify the problem. Buffer sizes requested are multiples of 256 bytes, and never longer than 4KB (page size). The total allocated memory will not exceed 4 pages, i.e., 16KB. You need not worry about allocating chunks across page boundaries, i.e., you can assume that every allocated chunk fully resides in one of the 4 pages.

  Upon receiving the `alloc` request, your memory allocator should check if it has a free chunk of memory to satisfy this request amongst its existing pages. If not, it must call `mmap` to allocate an anynomous private page from the OS, and use this to satisfy the allocation request. Memory must be requested from the OS on demand, and in the granularity of 4KB pages. However, the allocator should not memory map more pages than required from the OS, and should only request as many pages as required to satisfy the allocation request at hand. For example, suppose your memory allocator has been initialized, and the user of your memory allocator has invoked `alloc(1024)` to allocate 1024 bytes. Your allocator should make its first `mmap` system call at this point, to memory map one 4KB page only. The next `mmap` system call to allocate a second page must happen only when there is no free space within the first memory mapped page to satisfy a subsequent allocation request. It is very important to note that successive calls to `mmap` may not return contiguous portions of virtual address spaces on all systems. Your code must not rely on this assumption, in order to be portable across systems. Therefore, please do not attempt to allocate chunks that cross page boundaries, and ensure that an allocated chunk is always fully within a page.

- The function `dealloc(char *)` takes a pointer to a previously allocated memory chunk (that was returned by an earlier call to `alloc`), and frees up the entire chunk. There is no requirement for your heap to shrink on deallocations, i.e., you need not ever give back freed up empty pages to the OS via the `munmap` system call.

Requirements of merging and splitting free chunks remain the same as in part A. We have provided a simple test program `test_ealloc.c` to check your implementation. This program performs multiple allocations and deallocations using your custom memory allocator, and checks the sanity of the allocated memory. The test script also checks that you are correctly splitting and merging existing free chunks to satisfy allocation requests. To check that you are only memory mapping pages from the OS on demand, as specified in the problem statement, the program also prints out the virtual memory size (VSZ) of the process periodically. The comments printed out by the test program should help you figure out how the VSZ of your program is expected to grow in a correct implementation.

(Note that we can only check correctness of the values of VSZ and not of the actual physical memory used by your process, because the physical memory allocation is out of your control and is fully handled by the OS demand paging policies.)

Submission instructions

- You must submit the files `alloc.c`/`alloc.cpp` in part A, and `ealloc.c`/`ealloc.cpp` in part B. You need not submit the testing code.
- Place these files 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.

Grading

We will use test scripts (with possibly new testcases than those provided to you) to test the correctness of your code. We will also read your code to ensure that you have adhered to the problem specification.