ITP 30002-{01, 02} Operating System, Spring 2021

Homework 4. `rmalloc`: reconfigurable memory allocation library

Submission deadline: 11 PM, 15 May (Sat), 20211

1. Overview

This homework asks you to complete a library program for reconfigurable dynamic memory management called `rmalloc`. As similar with the standard C library (e.g., `malloc`), `rmalloc` acquires memory space via `mmap`, maintains linked lists to manage used and unused memory slots (Section 2.2), and serves memory allocation or deallocation requests from a user. A unique aspect of `rmalloc` is that it provides a function by which a user can chance a memory allocation policy in runtime. In addition, `rmalloc` has a library function to print out the snapshot of the linked list of the memory slots to display the internal status of memory management.

You can find the skeleton code of `rmalloc` at the following link: https://github.com/hongshin/OperatingSystem/tree/hw4. The submission deadline of this homework is **11 PM, 15 May** by when you must turn in all source code files and a report on the results.

2. Program Design

Basically, `rmalloc` implements the ideas presented at Chapter 17 *Free Space Management* in the OSTEP book. Upon user requests, `rmalloc` acquires empty pages from the operating system by `mmap`, and then embed a free list to the acquired memory to maintain available space.

2.1. Interfaces

As declared in `rmalloc.h`, there are six functions provided to a user for managing dynamic memory:

- `void * rmalloc (size_t s)`: Allocate a memory region of `s` bytes and returns the start address. This function must return null if it failed to allocate a memory area.
- `void rfree (void * p)`: Deallocate a memory region previously allocated by `rmalloc`. This function aborts program execution (i.e., `assert`) if `p` is not a valid address or `p` was already freed before.
- `void * rrealloc (void * p, size_t s)`: Resize the allocated memory area into `s` bytes. As the result of this operation, the data may be immigrated to a different address, as like `realloc` possibly does.
- `void rmshrink ()`: Reclaim unused memory to reduce the amount of allocated memory as much as possible.
- `void rmconfig (rm_option opt)`: Set the space management scheme as `BestFit`, `WorstFit`, or `FirstFit`.
- `void rmprint ()`: Print out the internal status of the free list and the used list (Section 2.3) to the standard output.

2.2. Operations

This library maintains two linked lists of memory regions, `rm_free_list` and `rm_used_list` to serve memory allocation requests and manage free spaces [1]. A memory region is a continuous chunk of memory containing a header structure (i.e., `struct rm_header`) and a payload. `rm_free_list` contains all unused memory regions, and `rm_used_list` does all memory regions given to the user (i.e., used).

For a user's request of allocating consecutive `m` bytes, function `rmalloc` first finds a memory region in `rm_free_list` feasible to accommodate the request. If the found memory region has exactly `m` bytes in its payload, `rmalloc` moves the memory region to `rm_used_list` and returns the starting address of the payload. If the found memory region has more than `m` bytes in its payload, `rmalloc` splits the found memory region into two pieces such that the first one is used for serving the user's request of `m` bytes (thus, added to `rm_used_list`), and the second one is to represent the remaining free space (thus, added to `rm_free_list`).

Selection of an unused memory region for serving a memory allocation request is determined by which of the three space management schemes is set to be used. `rmalloc` by default makes a decision with the First-fit scheme (i.e., `FirstFit`). A user can change the scheme later by calling `rmconfig`, thus `rmalloc` must comprise three different space management schemesl

If no memory regions in `rm_free_list` is large enough for splitting, `rmalloc` must acquire more memory spaces via `mmap`[2]. Although its main purpose is to set up a memory area for memory mapped I/O, `mmap` can be used also for acquiring a new free memory area without any connection to a file when it is invoked with an option `MAP_ANON`[3]. `rmalloc` must call `mmap` to allocate new memory if it is needed. For efficient memory uses, `mmap` must be called to allocate new memory in page size units. Note that memory allocation by `mmap` is properly aligned with pages. The size of a page in a system can be queried by `sysconf` or `getpagesize`.

For a memory free request from a user, `rfree` reclaims the used memory by moving the corresponding memory region from `rm_used_list` to `rm_free_list`. If `rfree` receives an invalid request, it should abort the program execution. In addition, `rfree` must conduct as much coalescing as if possible.

When `rmshrink` is invoked, it tries to reduce the amount of

---

[1] Unlike the example of OSTEP: Ch. 17.2, this library maintains not only a free list, but also another linked list of *used* memory regions. For this reason, there is no need to mark magic number for used memory regions.

[2] Even though multiple ways exist to acquire memory resource from an operating system, you are restricted to use `mmap` in this homework,

[3] See the `mmap` manual: https://man7.org/linux/man-pages/man2/mmap.2.html

the acquired memory as much as possible by releasing unnecessary memory areas. `rmshink` should call `munmap` to reclaim a memory area if the whole part of it is not in used.

`rmprint` is to display the status of `rm_free_list` and `rm_used_list` to show how heap memory is allocated at a moment in a program execution.

3. Your Tasks

Before getting started working on this homework, it is highly recommended to study OSTEP: Chapter 17 thoroughly (even some of the program design is not the same) and look into `mmap` in details.

After the background study, you need to write the missing definitions of the functions in `rmalloc.c` to fulfill all functionalities given at Section 2.2. In the implementation, it is strictly prohibited to modify `rmalloc.h` and the definition of the `rmprint` function. Note that your program will be desk rejected at evaluation if any content of these immutable parts were changed.

In this homework, you are also asked to write a report on your homework results. Specifically, your report must include the followings:

- Explanations on the logics of important operations,
- Demonstration of multiple test scenarios that different aspects of your program work correctly
- Discussion on your results including challenges you have faced and/or questions you have bear in in doing this homework, new ideas for improvements, etc.

Evaluation will be primary based on your report. Thus, please try to best to deliver your results in your report. Especially, the comprehensiveness of your demonstration will be carefully checked to see how many requirements were properly addressed in your solution. In addition, your program will be run against new test cases to see whether it behaves as expected.

4. Submission

The submission deadline is **11 PM, 15 May (Sat)**. Submit an archive (e.g., as a `tar` or `zip` file) of all source code files and a PDF file of your report at the homework submission form entitled with “`Homework 4`” in Hisnet. Your report must use the given template and must not exceed 2 pages in the template. Please convert your report to a PDF file before submission.