Assignment 3: myalloc

Date: April 28th, 2018
Deadline: May 13th, 2018, 23:59

Objectives

You must implement a low-level memory manager: the OS functions that make physical RAM available to other parts of the system.

Context

Your work will be evaluated with a simulator of physical RAM. The simulated RAM is organized in one or more modules (like memory chips). Each module is organized as an array of contiguous equally-sized banks. Each bank contains many bytes.

Each module is connected to the processor (where your code runs) at a fixed address. Your code will receive information about the list of modules and their starting address during initialization. The starting address of each module is always multiple of the bank size, which itself is always power of two.

Like in real RAM, banks must be activated before use, and deactivated when they are no longer needed. This is needed because when a bank is activated, it consumes energy, so we wish to reduce energy wastage. Your code must use the provided API to activate and deactivate banks. The challenge is that the activation and deactivation is a relatively slow operation. The goal of the assignment is to find an algorithm that both maximizes performance and decreases energy use.

Your submission must provide an API that can be integrated in the rest of an operating system. Your API must manage the banks of RAM behind an interface similar to `malloc` and `free`. See `myalloc.h` for details.

Requirements

Your code must not use an existing memory manager. For example `malloc` and `free` are forbidden. Your code must not use global variables; the only storage you can use for your management structures are the simulated memory modules themselves. You must submit your work as a tarball [1]. Your archive must also contain:

- a text file file named "`AUTHOR`" containing your name and Student ID.
- a file named `report.pdf` which contains a write-up of your results (see below).
- If you want your scores to be posted on the leaderboard without you student numbers you must include a `LEADERBOARD_NAME` file with the name you want to be displayed in your archive.

It is strongly advised you use version control.

Getting started

1. unpack the provided source code archive; then run `make`
2. Try out the generated `test-dumb` and familiarize yourself with its interface. Notice how `test-dumb` uses the implementation in `mm-dumb.c`.
3. Try out different trace files (`.t`) with `-k`.
4. Try to run `test` which uses the same interface, but the implementation in `mm.c`. What do you see?
5. Edit the file `mm.c`. This is probably where you should start implementing your memory manager.

Description of the functions to implement

`mm_alloc(st, sz)`

Allocate `sz` bytes of memory for the requesting program. Return a pointer to the start of the allocated memory, or 0 if the memory could not be allocated. The start address (pointer returned by `mm_alloc`) must be a multiple of `sizeof(intmax_t)` [2].

`mm_free(st, p)`

Release the memory pointed to by `p`.

Grading

Your grade starts from 0, and the following tests determine your grade (in no particular order):

- +0,5pt if you have submitted an archive in the right format with an `AUTHOR` file and a non-empty `report.pdf`.
- +0,5pt if your source code builds without errors and you have modified `mm.c` in any way.
- +1pt if your code supports a simple sequence of calls to `mm_alloc` without errors using only 1 memory module and returning a valid range every time.
- +1pt if your overhead per allocated byte is asymptotically constant in long sequences of calls to `mm_alloc` followed by `mm_free` using only 1 memory module (in other words, your `mm_free` causes banks to be deactivated eventually).
- +1pt if your code can optimize for performance: during sequences of `mm_alloc` then `mm_free`, `hw_activate` is called less often than there are calls to `mm_alloc` (bank reuse).
- +1pt if your code has a significantly lower number of "allocations failures despite empty space" than the `dumb` allocator (i.e. your code actively manages free space).
- +2pt if your report satisfies the requirements set below.
- -1pt if *either* `gcc -W -Wall` or `clang -W -Wall` reports warnings when compiling your code.
- -1pt if `valgrind` complains about your code.

The following extra features will be tested to obtain higher grades, but only if you have obtained a minimum of 5 points on the list above already:

- +0,5pt if your manager can satisfy multiple calls to `mm_alloc` by sharing a common bank (when the allocation size is smaller than a bank).
- +1pt if your code can optimize for energy usage by deactivating banks released by `mm_free`.
- +0,5pt if your code tries to increase locality, i.e. `mm_alloc` tries to reuse the banks most recently deallocated by `mm_free`.
- +0,5pt if your code supports multiple modules and tries to spread allocations evenly across modules (minimizes the imbalance factor).
- +0,5pt - +1,5pt for performance, see below.
- +0,5pt - +1,5pt for performance/watt, see below.
- -1pt if any if your source files contains functions of more than 30 lines of code.
- -1pt if your source files are not neatly indented.

The grade will be maximized at 10, so you do not need to implement all features in this second list to get a top grade.

Note: Your memory manager will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort".

Structure of your report

The accompanying `report.pdf` is worth 2 points. It must contain at least the following:

- your name and student ID;
- an overview of the data structures and algorithms you have used, and their asymptotic complexity;
- a list of the grading objectives you have (attempted to) reach during your work;
- if you have made any effort towards optimizing for performance or performance/watt, a table that compares the various metrics of your code with the example `dumb` implementation;
- all bibliographic references to your sources of inspiration, if any.

Your report should be between 1 and 4 A4 pages.

Performance evaluation

At the end of each successful simulation, some statistics are printed. Your submission is eligible for extra credits as follows:

1. you are only eligible if you submitted your work on time, and this is not your 2nd try.
2. we will first select all eligible submissions that have less than 50% "allocation failures despite empty space" on our reference traces, i.e. select only implementations that minimize fragmentation somehow.
3. we will rank the "performance" and "performance/watt" statistics for the selected submissions.
4. the student(s) with the highest performance will get +1,5pt, the student(s) with 2nd best will get +1pt, and the student(s) with 3rd best will get +0,5pt.
5. the student(s) with the highest performance/watt will get +1,5pt, the student(s) with 2nd best will get +1pt, and the student(s) with 3rd best will get +0,5pt.

We will cluster the performance statistics so that submissions within the same range (5%-10%) will be treated as equal.

Understanding trace files

The provided test program operates under control of a *trace file*: a text file containing orders to call `mm_alloc` and `mm_free` in a specific order.

A couple of example trace files are provided (`ls *.t`) with the example framework. You can also visualize the commands of a trace file commands using "`test -t -k ...`".

For advanced users only: You may also define your own trace files. To understand the trace format, read the source code of `run_trace` in `test.c`.

Evaluation environment

During evaluation, we will use different trace files containing more complex combinations than those already provided. Also, we will check different RAM configurations:

- Using small and large number of banks per module (`-b`).
- Using small and large bank sizes (`-s`).

Be sure to test thoroughly! **Think about automating your testing**.

1 http://lmgtfy.com/?q=how+to+make+a+tarball
2 `intmax_t` is defined in `<stdint.h>`.