CS-350 - Fundamentals of Computing Systems
Homework Assignment #7 - BUILD

Due on November 7, 2024 — Late deadline: November 9, 2024 EoD at 11:59 pm

Prof. Renato Mancuso, Prof. Anna Arpaci-Dusseau

Renato Mancuso, Anna Arpaci-Dusseau

BUILD Problem 1

Now that we have a nice running image server, we will now look to understand how our code interact with key system resources, in particular the CPU, the Level-1 Cache (L1 Cache), and the Last-Level Cache (LLC). In order to do this we will leverage the Unix "perf" infrastructure. The perf subsystem gives us access to several hardware counters in our machine, allowing us to profile instruction counts and memory behavior such as cache hits/misses. As we have seen in class, CPU-bound operations will spend most of their time on the CPU, without requiring additional resource. Operations that require access to a lot of data, on the other hand, will demand more service from the memory subsystem. Whenever a process needs data, L1 cache lookup is performed. If the data is not in L1, an LLC lookup is performed. A miss in LLC means that an access in main memory (DRAM) needs to be performed.

Output File: `server_img_perf.c`

Overview. Once again, we are not going to scrap everything we have build together so far. On the contrary, we are building on top of it once again. This time around, we have provided you with a new library called `perflib` (see files `perflib.c` and `perflib.h`). With this assignment, we will enable profiling of operations in our image processing server!

First, the type of operations from the client are the same as before, and the same goes for the structure of the client-originated requests and expected responses. Your goal is to use the provided `perflib` to sample the various hardware performance counters and determine how many *hardware events* are caused by an image operation carried out by the server.

So what are *hardware events* anyway? As your code executes, it completes assembly instructions[1]. The completion of a CPU instruction is an example of one such events. Say that while executing an instruction, data is looked up into the L1 Cache and not found (L1 cache miss), that's another type of *event*. The same goes for a lookup that causes a miss in LLC. Thus, as you execute any code, one can count the number of events of different type caused by your code. And "perf" can be used precisely for this.

Design. Please take a look at the code of the new `perflib` library. The library provides useful helper functions to interact with "perf" and sample the various hardware counters of interest.

For this assignment, once again, we will ask you to implement **single-threaded** implementation of the image server. The main difference is that every image processing operation carried out by the server will be profiled to understand the number of *hardware events* generated by the corresponding code. For each run of the server, we will focus on a specific type of event among three events of interest:

1. Number of completed CPU instructions;
2. Number of L1 cache misses;
3. Number of LLC cache misses.

To do so, first, the server should accept an additional flag as one of the input parameters, namely `-h`. This flag will be used to specify what type of hardware event we wish to count when the server is launched from the command line. The `-h` parameter must be followed by a string: (1) "INSTR", (2) "L1MISS", or (3) "LLCMISS".

In your server, once you detect the argument of the `-h` parameter, you should pass this information to your worker thread which will use the provided `perlib` functions to setup and sample hardware event counters.

The provided `perflib` provides two main operations: (1) `setup_perf_counter(...)` and (2) `read_perf_counter(...)`.

[1] CS210 anyone?

Problem 1 continued on next page...

First, the `int setup_perf_counter(uint64_t type, uint64_t config)` function allows you to setup event counting for the calling thread. If event setup is successful, it returns an integer that can be considered as a *handle* (actually, a file descriptor) to access the counters. Say we call this handle `evt_fd`. The function takes two parameters to specify the event to be monitored via its `type` and `config`.
When we are setting up our performance counter, the `type` parameter and `config` parameter must be set in correspondence with the event we intend to monitor.

1. Instruction: `type = PERF_TYPE_HARDWARE` and `config = PERF_COUNT_HW_INSTRUCTIONS`.
2. L1 cache misses: `type = PERF_TYPE_HW_CACHE` and

```
config = (PERF_COUNT_HW_CACHE_L1D) | (PERF_COUNT_HW_CACHE_OP_READ << 8)
            | (PERF_COUNT_HW_CACHE_RESULT_MISS << 16)
```

3. LLC cache misses: `type = PERF_TYPE_HW_CACHE` and

```
config = (PERF_COUNT_HW_CACHE_LL) | (PERF_COUNT_HW_CACHE_OP_READ << 8)
            | (PERF_COUNT_HW_CACHE_RESULT_MISS << 16)
```

You can read `https://man7.org/linux/man-pages/man2/perf_event_open.2.html` to find out about all the different types of hardware counters and how to configure perflib to count different events!
Once the counters have been setup, you can enable counting with the system call:
`ioctl(evt_fd, PERF_EVENT_IOC_ENABLE, 0)`.
You can also reset the current value of the counters with using:
`ioctl(evt_fd, PERF_EVENT_IOC_RESET, 0)`.

Most importantly, you can always read the current value of the counter you have setup for observation with the provided `uint64_t read_perf_counter(int evt_fd)` function. This function takes in input the aforementioned handle `evt_fd` and returns the current value of the number of sampled hardware events since the last counter reset.
To recap, (1) extend the `main` to recognize the new `-h` paremeter; (2) perform the setup of the event under consideration, according to what has been passed using the `-h` parameter. This needs to be done during the initialization of the same (!) thread that will sample the hardware events. Next, (3) every time an image operation is performed by the worker thread, sample how many hardware events (of the appropriate type) are caused by said operation. Finally, (4) print out the sampled number of hardware events together with the other statistics about the completed operation—see output format below.

Desired Output. The output that should be produced every time an image processing request is completed matches for the most part the format we have seen in HW6. Specifically, at every request completion, your server must output something of the form:

```
T<thread ID> R<req. ID>:<sent ts>,<img_op>,<overwrite>,<client img_id>,<server img_id>,<receipt ts>,
<start ts>,<compl. ts>,<event name>,<event count>
```

Here, `<img_op>` is a string representing the requested operation over an image. For instance, if the operation was `IMG_REGISTER`, then the server should output the string "IMG_REGISTER" (no quotes) for this field. `<overwrite>` should just be 0 or 1, depending on what the client requested. `<client img_id>` should be the image ID for which the client has requested an operation.
If the server is ignoring any of these values in the response, set these fields to 0. Finally, `<server img_id>` should report the image ID on which the server has performed the operation requested by the client. Recall

that this might be different from what sent by the client if `overwrite = 0` in the client's request, but it must be the same if `overwrite = 1`.
Next, `<event name>` must be the same string as what was passed to the server with the `-h` parameter—thus "INSTR", "L1MISS", or "LLCMISS". Finally, `<event count>` must be the sampled number of hardware events of the considered type occured while processing the request with ID `<req. ID>`. NOTE: these last two fields must be omitted ONLY in the case of an `IMG_REGISTER` operation.

Submission Instructions: in order to submit the code produced as part of the solution for this homework assignment, please follow the instructions below.
You should submit your solution in the form of C source code. To submit your code, place all the `.c` and `.h` files inside a compressed folder named `hw7`.zip. Make sure they compile and run correctly according to the provided instructions. The first round of grading will be done by running your code. Use CodeBuddy to submit the entire `hw7`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa23/codebuddy.php?hw=hw7`. You can submit your homework multiple times until the deadline. Only your most recently updated version will be graded. You will be given instructions on Piazza on how to interpret the feedback on the correctness of your code before the deadline.