CS-350 - Fundamentals of Computing Systems
Homework Assignment #1 - BUILD

Due on September 19, 2024 — Late deadline: September 21, 2024 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

BUILD Problem 1

In this BUILD problem, you are tasked with the first code deliverable to shine light on the temporal behavior of a real system. Specifically, you will learn how to measure the clock frequency of your CPU.

Output File: `clock.c`

Write the logic to compute the CPU frequency of the CPU on which this code will be executed by measuring the number of clock cycles elapsed in a window length of known size. We will approach this measurement in TWO alternative ways to perform a comparison. The first is *sleep-based*, the second is by performing *busy-waiting*. To do that, proceed as follows.
First, implement the C macro that uses the `RDTSC` assembly instruction to read (RD) the current timestamp counter (TSC) from the CPU. The TSC is a counter maintained in hardware that increments by one at every CPU clock cycle from the time the CPU was powered on. Call the macro that reads the current TSC value as `get_clocks(...)` and implement it inside the `timelib.h`, following the template.
Next, implement the function called `get_elapsed_sleep(long sec, long nsec)` inside `timelib.c` that uses **sleep-based** waiting to delay execution on the processor for a given amount of time, passed as two parameters: (1) number of seconds, (2) number of nanoseconds. The function, will use `get_clocks(...)` to snapshot the TSC, then use `nanosleep(...)` to sleep for the given amount of time, and finally re-snapshot the current TSC value. The return value of this function should be the difference between the second (after the sleep) and the first (before the sleep) TSC values.
Next, implement the function called `get_elapsed_busywait(long sec, long nsec)` inside `timelib.c` that uses a **busy-waiting** based procedure to measure the number of CPU clock cycles elapsed in a fixed time window. The function accepts the same parameters as before, i.e. the seconds and nanoseconds specifying the length of the waiting time window. In this function, first retrieve from the system the current system time, call it `begin_timestamp`, using the library function `clock_gettime(CLOCK_MONOTONIC, ...)`. Next, snapshot the current TSC value. Next, enter a `while loop` during which the `clock_gettime(CLOCK_MONOTONIC, ...)` function is used to monitor if enough time has elapsed since `begin_timestamp`. If enough time has elapsed, exit the loop and re-snapshot the TSC. As before, return the difference in values between the second and first TSC values.
In the `main`, you should accept 3 parameters. (1) the number of seconds to sleep for; (2) the number of nanoseconds to sleep for; and (3) a single character 's' or 'b' to use sleep-based or busy-waiting procedure to measure the number of clock cycles elapsed in a fixed time window. Using these parameters, put the CPU to busywait (or sleep) for the specified amount of time and measure how many clock cycles elapsed during that lapse of time. Finally, compute the CPU clock speed as the ration between elapsed clock cycles and wait time.
Your code should produce an output that follows the format below:

```
WaitMethod: <method>
WaitTime: <seconds> <nanoseconds>
ClocksElapsed: <elapsed clock cycles>
ClockSpeed: <measured clock speed>
```

In the output, `<method>` will either be `SLEEP` or `BUSYWAIT` (depending on the third input parameter); `<seconds>` and `<nanoseconds>` will be the amount of seconds and nanoseconds to wait for (1st and 2nd parameter); `<elapsed clock cycles>` and `<measured clock speed>` should be the number of elapsed CPU clock cycles (as an integer) and the measured CPU clock speed printed as a double-precision decimal number in MHz, respecively.

BUILD Problem 2

In this BUILD problem, you will start writing the scaffolding of a networked server application capable of replying to simple incoming requests by using time measurements to create workload with known temporal properties.

Output File: `server.c`

Start with the provided `server.c` template file. Take a look at its structure. The main logic of the code deals with establishing a TCP/IP socket on a port specified as the first and only parameter of the server. Familiarize with how this is accomplished by following the flow of system calls: (1) `socket(...)`, (2) `bind(...)`, (3) `listen(...)`, and (4) `accept(...)`.
Next, identify the function called after a successfully accepted connection, namely `handle_connection(...)`. This function should implement a loop that continues so long as the socket connection is alive (i.e., the client has not interrupted the connection).
At each iteration of the loop, the server must block waiting for a new request to be received on the socket. A request sent by the client is an object comprised by the following fields (in this exact order).

1. An unsigned 64-bit integer encoding the request ID;
2. A `timespec` structure encoding the timestamp (in seconds and nanoseconds) at which the client sent the request (set by the client);
3. A `timespec` structure encoding the length (in seconds and nanoseconds) of the request (set by the client).

Upon receiving a new request, the server must perform a busy wait for the amount of time specified by the client as the request length. It is important that you perform a busy wait instead of using sleep-based functions. You can reuse the `get_elapsed_busywait(long sec, long nsec)` procedure developed in the previous problem.
After the wait is completed for the request at hand, the server must reply to the client with a response. The response MUST have three fields: (1) the 64-bit ID of the handled request, (2) a reserved 64-bit field that you can simply set as 0 for now—it will be ignored by the client at this stage and will be used later; (3) an 8-bit acknowledgement value that needs to be set to 0 for now, to indicate that the request has been correctly handled.
For each handled request, the server should output the following information, in the exact format reported below.

```
R<request ID>:<sent timestamp>,<request length>,<receipt timestamp>,<completion timestamp>
```

Here, `<request ID>` is the ID of the request as sent by the client; `<sent timestamp>` is the timestamp at which the request was sent by the client; `<request length>` is the length of the request as sent by the client; `<receipt timestamp>` is the timestamp at which the server received the request; and `<completion timestamp>` is the timestamp at which the server completed processing of the request and sent a response back to the client. All the timestamps should be expressed in seconds with decimal digits with enough precision to capture microseconds.

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 `hw1`.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 `hw1`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa24/codebuddy.php?hw=hw1`. 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.