CS:3620 — Spring 2020 — Homework 4

This homework is about the pthread API in Linux.

Please, ask general questions about the homework on ICON, so that everyone can benefit from the answers.

General Requirements Submit your homework as a *single* `tar` file. When unpacked, the `tar` file must have the following directory structure (substitute ⟨your_HawkID⟩ with your actual HawkID):

⟨`your_HawkID`⟩
    `task.c`

No other file should be present in the archive. Notice that, for this homework, you do not have to provide a compilation script. Your code will be compiled using the following command:

```
gcc -ggdb -O0 -o task task.c -lpthread -lcrypto
```

To compile correctly your code, it may be necessary (in Ubuntu 16.04 64bit) to install the package `libssl-dev`:

```
sudo apt-get install libssl-dev
```

`task.c` needs to be a modified version of the file `task.c` provided in the `tar` file of this homework, as explained below.

Main Task

Your overall goal is to change the provided `task.c` file to make it able to find solutions for the assigned problems using multiple parallel threads. I will now describe what the provided code does. Your code is not to re-implement what the provided code is already doing, but just to change it, as described in the section *Your modifications*.

The provided `task.c` code

The provided code implements a program that takes two or more command line arguments: ⟨`nthreads`⟩ ⟨`challenges...`⟩. ⟨`nthreads`⟩ specifies the number of parallel worker threads the program uses to solve the given challenges (as better explained below). ⟨`challenges...`⟩ is a list of one or

more integer numbers in base 10. Examples:
`./task 1 12`
The program will solve the challenge 12, using 1 worker thread.
`./task 1 12 1000`
The program will solve the challenges 12 and 1000, using 1 worker thread.
`./task 3 12 1000 123`
The program will solve the challenges 12, 1000, and 123, using 3 worker threads.

For every given challenge, the program output a line to stdout, with the following format:

⟨challenge⟩ ⟨solution1⟩ ⟨solution2⟩ ⟨solution3⟩ ⟨solution4⟩ ⟨solution5⟩ ⟨solution6⟩ ⟨solution7⟩ ⟨solution8⟩

For instance, when run in this way:
`./task 1 12 1000 123`
The output is:

```
12 1720 15424 184447 262378 431435 461772 526096 812453
1000 62444 91459 120212 147426 189835 281930 338887 542638
123 113731 150733 534218 618885 677519 694956 861114 944627
```

The 8 solutions outputted for each challenge satisfy the following 3 conditions:

1. For all the solutions, applying the algorithm `SHA256` to a solution returns a hash starting with the given challenge.[1] Every solution is considered as a little-endian unsigned 64bit number, and every challenge is considered as a little-endian unsigned 16bit number.

2. All the solutions have a different last digit (considering them in base 10).

3. All the solutions are not divisible for any number in the range [1000000,1500000].

The provided code generates correct solutions when running with ⟨`nthreads`⟩ equal to 1. However, since it does not use any locking mechanism, it fails when running with ⟨`nthreads`⟩ bigger than 1. The code may

[1] To solve this homework, you are not required to know the details of this algorithm. If you are interested in this topic (and how it is related with Bitcoins) you can read: `https://en.wikipedia.org/wiki/SHA-2` and `https://en.bitcoin.it/wiki/Block_hashing_algorithm`, or ask me.

fail in different ways: entering an infinite loop, returning solutions which do not follow the 3 conditions above, triggering a `Segmentation fault`, ...

Your modifications

Your goal is to modify the provided code so that it respects the following properties.

1. Your modified code outputs solutions in the same format than the original one. Do not add any `printf` statement! Your homework will be automatically graded, printing to `stdout` or `stderr` any extra text will most likely confuse the grader, potentially evaluating your homework as completely incorrect.

2. Your code needs to create and use a number of worker threads equal to ⟨`nthreads`⟩.

3. When running with ⟨`nthreads`⟩ equal to 1, your modified code must not run more than 20% slower than the original code.

4. For each challenge, your modified code has to print 8 solutions, still respecting the 3 conditions explained above. This property must be true when your code is run with any number of ⟨`nthreads`⟩ between 1 and 100 (included).

5. Assuming that you are using a machine with at least 4 vCPUs[2], when the value of ⟨`nthreads`⟩ is 2, 3, 4, or 5, your code needs to run at least 25% faster than when the value of ⟨`nthreads`⟩ is 1.

The provided `test.py` script tries to automatically verify property 1, property 4, and property 5. You can run it with the following comand:
`python test.py ./task`

This script prints to `stdout ERROR` when a test fails, and `SUCCESS` when it succeeds, alongside debugging output.

If you run `test.py` with the original provided code for `task`, some tests will succeed, but any test involving a value of ⟨`nthreads`⟩ bigger than 1 will likely fail.

The script `test.py` assumes that your machine has at least 4 vCPUs and it is not under heavy load.

[2]You can check this using `htop`, as explained in class.

Hints and Suggestions

Do not modify any provided code verifying the 3 conditions mentioned above. Your goal is just to make the code working with multiple worker threads (i.e., values of ⟨`nthreads`⟩ bigger than 1), not to implements numeric checks.

You can solve this homework by adding/modifying less than 20 lines of code.

Before starting writing any code, spend your time understanding the provided code and the homework description. Also, spend some time testing your code, since, when working with threads, code can work correctly sometimes and incorrectly other times, in unexpected ways.

There are 3 main aspects that you need to consider to fix the provided code.

First of all, in the provided code, all the worker threads start looking for solutions from the value 0. This needs to be changed so that every worker thread checks different candidate solutions.

Then, two global variables are read/written by different worker threads: `found_solutions` and `solutions`. Accessing them should be *protected* using an appropriate locking mechanism. I suggest to use a `read-write lock`, but there may be other ways to solve this homework. The lock can be acquired for reading when just checking the value of one of these variables, and acquired for writing when modifying it.

Finally, any worker thread should check if all the required 8 solutions have been found, and, if this is the case, terminate. This can happen at any time during the execution of the code in `worker_thread_function`. You may need to add instructions checking this.

Remember always to unlock an acquired lock. It is easy to make mistakes and write code that, under certain conditions, forgets to unlock a lock. This will most likely cause your code to stall.