PA4: Password Cracker (Due 10/18 @ 11:59PM)

The GitHub classroom link for this assignment is: https://classroom.github.com/a/nlw6Gph8

The focus of this assignment is writing a program that tries to decrypt passwords which were “encrypted” with the very weak SHA-1 hashing function.

Installing Thread Building Blocks

For this assignment, we use thread building blocks for parallelization. You’ll have to install it.

I gave the direct links to the installers below, but if for some reason these do not work, go to https://software.intel.com/en-us/intel-tbb and follow the instructions to download them.

Windows

- Download and run this:http://registrationcenter-download.intel.com/akdlm/irc_nas/tec/11993/w_tbb_2018.0.124.exe
- Make sure you install it into the default directory. You don’t have to install the Visual Studio plug-ins (you select “customize” and just uncheck them)

Mac

- Download this image, and run the contained app: http://registrationcenter-download.intel.com/akdlm/irc_nas/tec/11992/m_tbb_2018.0.104.dmg
- Make sure you install into the default directory. You don’t have to install the “for Android” version of the libraries if you don’t want to

Linux

- apt-get libtbb-dev

As with PA3, you’re expected to create new files in the src directory as needed, and update CMakeLists.txt as needed.

Input files are given in the input subdirectory. There’s pass files which contain a list of passwords “encrypted” with the very weak SHA-1 hashing scheme. The d files are plaintext dictionary files (d2.txt is a small one, d8.txt is the full one)

SHA1 Hashing

To compute the SHA-1 hashes, we will be using a simple open source library for this purpose. This is declared in Sha1.h and implemented in Sha1.cpp.

There are only two functions in this library. To calculate the hash, you use `sha1::Calc`, which takes three parameters (pointer to data, number of bytes of the data, and the pointer to where you want to store the data). For example, if you wanted to hash the string `"abcd"` you would do:

```cpp
unsigned char hash[20];
sha1::Calc("abcd", 4, hash);
```

This will store the 20 unsigned chars into hash. But if we want to display this as a hexadecimal string we can view, you can use the following code:

```
char hexStr[41];
sha1::ToHexString(hash, hexStr);
```

This would store the string "81fe8bfe87576c3ecb22426f8e57847382917acf" into `hexStr`.

Note that the size of `hexStr` is 41. That's because even though the SHA-1 hash is only 40 characters, you need one more byte to store the `'\0'` to mark the end of a C-style string.

Dictionary Lookup

In ProcessCommandArgs, if argc == 3, argv[1] is the name of the dictionary file, and argv[2] is the name of the password file.

Now that you know the basics of how to hash, add code that loads in the dictionary file, and creates an unordered_map of the dictionary. The key is the hashed password text, and the value is the unencrypted password text.

Decrypting with the Dictionary

After loading the dictionary, read in the password file. Each line has the hash of a password. Do a lookup into your map to find out if you know what the plaintext password is.

Output the all the solved/unsolved passwords to solved.txt. If you successfully find a password in the dictionary, you should output the original hexadecimal hash, followed by a comma, followed by the plain text password. If you fail to find a password in the dictionary, to the right of the comma simply output a couple of question marks.

At this point, you should pass the "dictionary only" test.

Brute Force

Now add code so that when the dictionary look-up finishes, you attempt to brute force the remaining unsolved passwords. Note that you should **NOT** run the brute force algorithm on every single password in the password file. Only run it on the password hashes which were not in the dictionary.

For us to use TBB effectively, we need to separate the passwords we need to brute force from those we have already solved. And since ***it's a requirement*** that the order of the password hashes in solved.txt is identical to that of the original pass.txt, we can't just immediately output the text in solved.txt as in Part 2.

You'll want to store the decrypted information somehow. As you attempt to solve passwords using the dictionary lookup, you should separate the ones that weren't found from the dictionary from the ones that were.

As a reminder, you want to brute force passwords up to and including a length of 4. So you need to first do length of 1, then 2, then 3, and finally 4.

You want to use the counting machine approach I talked about in lecture (with carrying) – don't just use a quadruple for loop.

Once you have the string you're testing to see if it's the password, you can hash it with SHA-1 and compare it against the uncracked password. If it's unsuccessful, then you increment by one and try the next password. This way you guarantee that you'll go through every permutation.

First try implementing this in a serial manner – just loop through all the passwords and try to brute force them, one at a time. This will be relatively slow, but at least you can verify that your basic brute forcing works. Try the "brute force only" test case, for example.

Thread Building Blocks

Now parallelize your code using the `tbb::parallel_invoke` function to split the brute forcing search space into parts.

Right now, the serial approach incrementally tests every password in order like `"aaaa"`, `"aaab"`, all the way through `"9999"`, and goes through this range for every password! However, it makes much more sense to split up the search space. For example, you could split it in half by having one of the lambda expressions passed to `parallel_invoke` brute force from `"aaaa"` to `"q999"` while the other lambda could brute force from `"raaa"` to `"9999"`. Then as you go through each permutation one by one, you can also test every unsolved password against the permutation.

In our case, we want to split it up into a smaller grain size than just half. Since there are a total of 36 characters, you should split up the search into 9 separate lambda expressions invoked simultaneously, each doing 4 different characters. For example, the first lambda would do `"aaaa"` to `"d999"` and the second lambda would cover the range `"eaaa"` to `"h999"`.

Remember that `parallel_invoke` requires including `<tbb/parallel_invoke.h>`.

Now make sure your parallelized version still works on "Brute force only."

At this point, you need to switch to Release mode and try the two "full" test cases. As with the previous assignment, you have a time limit for completion. This time, the time limit is 2.5s (which is super high).

Grading Rubric

Part	**Points**
Part 1 - Basic Hashing	
Graded Unit Tests	75
Your Test Cases Quality/Variety	15
Code quality (incl. following the style guide)	10
Total	**100**