COSC 420 - High-Performance Computing
Lab 4

Dr. Joe Anderson

Due: 25 October 2019

1 Preliminaries

In modern Linux systems, the `crypt` library (see `info crypt` or the man page, e.g. http://man7.org/linux/man-pages/man3/crypt.3.html for some details) to encrypt and store user passwords. This data is stored typically in the file named `/etc/shadow` and is only readable by the system root. Different systems use various encryption algorithms, and when using the `glibc` version (which we are on the COSC server), you can additionally specify which algorithms are employed. As a technical note, you will have to have the pre-compiler macro `#define _XOPEN_SOURCE` above your `#include` directives, as well as compile with the `-lcrypt` linker argument.

In this lab, you will play the role of a "hacker" (possibly hired by the company to do a test of their security) who has obtained access to a system and made a copy of the shadow file. Back on your own system, you aim to find out which users have chosen an unsecure password, using what is known as a "dictionary attack"; in short, you will try and guess which password is correct, by brute force.

In the shadow file you will find user data in the following format:

```
<username>:$<algorithm id>$<salt string>$<encrypted password>
```

On this line, the `<username>` segment is the login name of the user. The `<algorithm id>` is an integer, usually 1, 5, or 6 to indicate which hashing algorithm was used to encrypt the password string (see the `crypt` documentation for a list of which algorithms use which id). The `<salt string>` is a string of nonsense characters that is appended to the user's password in order to obfuscate the encrypted strings against hackers like you; how this works is that, if there were no salt, you could pre-encrypt all the possible passwords beforehand, and simply look up the encrypted version in your pre-computed database. Because the salt is appended to the plaintext password, this means your pre-computed dictionary encryption is invalid! For example, if the salt is "abc" then when a user's password is "fluffybunny", what is *actually* encrypted is "abcfluffybunny". The `<encrypted password>` section is *part of* the output of the `crypt` function, when passed the salt, algorithm id, and unencrypted password string. The full output is `$id$salt$cypertext`.

The sequential version of your brute force password-guessing algorithm will go as follows:

```
1: for Every line in the shadow file do
2:     Parse the line into username, id, salt, and cyphertext
3:     for Every word, w, in the dictionary do
4:         for Every numerical prefix p and suffix s of integer strings (including empty string) do
5:             Let guess := ws or pw depending on if you are testing a prefix or suffix
6:             Calculate result := crypt(guess, "$id$salt$")
7:             Compare result to the second field of the parsed line. If they match, you've cracked it!
8:         end for
9:     end for
10: end for
```

2 Objectives

In this lab you will focus on the following objectives:

1. Develop familiarity with `c` programming language
2. Develop familiarity with parallel computing tools `MPICH` and `OpenMPI`
3. Develop familiarity with parallel file I/O

3 Tasks

1. You may work in groups of one or two to complete this lab. Be sure to document in comments and your `README` who the group members are.
2. You will read in the dummy `/etc/shadow` file and use MPI to try brute forcing users' passwords.
   (a) Make sure you download the dictionary file (words) from the course webpage
      i. Fun fact: this is just a copy of a standard `/usr/share/dict/words` file used on GNU/Linux for spell checkers!
   (b) Download the sample `shadow` file from the course webpage: shadow.
3. Use MPI and use multiple nodes to find which users have passwords of dictionary words, perhaps followed or preceeded by short strings of integers, e.g. "99balloon", "1337dude", or "password123".
   (a) You may assume that the prefix or suffix integer does not exceed 10 characters.
   (b) You may assume that each dictionary password has a prefix *or* a suffix number, but not both.
   (c) You may assume standard grammatical capitalization, i.e. proper nouns start with a capital. Note that the dictionary file *does* include many proper nouns like common names!
   (d) You may assume that no special characters appear in any password.
   (e) You may hard-code the number of lines in the `words` file: 235886. You can check this with the command-line utility `wc -l words` to count the lines. This will save you having to loop the file and count manually in the program.
   (f) **Not all users in the system use this strict dictionary word scheme!** But you must find out which do and which do not.
4. Use a shared output file where the cracked passwords can be stored in a user-readable format.
5. Make sure you use MPI's shared file access functionality to define "windows" from which each node can access the data files.
   (a) Created some derived `MPI_Datatype`'s will likely be a good idea to simplify logic
   (b) You may want some message-passing to "abort" the brute forcing of specific passwords, once one process has cracked it.
6. Note that for debugging output, it may be helpful to have each node write its output to a file, unique to that process. This will be how we will gather logs when submitting to larger clusters.
7. Use standard timing tools (c `time_t` or `MPI_Wtime`) to record the time it takes to complete your password crack. Compare using different numbers of cores (as practical): e.g., 10, 20, 50, 100
8. Include a `README` file to document your code, any interesting design choices you made, and answer the following questions:

(a) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the input size?

(b) According to the data, does adding more nodes perfectly divide the time taken by the program?

(c) Consider the problem of brute-forcing passwords under *only* maximum string length. How much time would it take to complete this hack, extrapolating from your measurements?

(d) What are some real-world software examples that would need the above routines? Why? Would they benefit greatly from using your distributed code?

(e) How could the code be improved in terms of usability, efficiency, and robustness?

4 Submission

All submitted labs must compile with `mpicc` and run on the COSC Linux environment. Include a `Makefile` to build your code. Upload your project files to MyClasses in a single `.zip` file. Finally, turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code and constituent subroutines. Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.

5 Bonus

(10 pts each) There are several users who have made their passwords *slightly* more secure, by using multiple words, character substitutions, and special characters. Each one of these that you crack will be worth extra points!