In this MP, you will be creating a program that can recover lost passwords. For security reasons, passwords are usually never stored in plain text. Instead, the hashed versions of passwords are stored. For an example of this, take a look at the /etc/shadow file on any modern linux machine.

When a user tries to log in, the password they enter is hashed and compared with the stored hash. This way, there’s no need to store your actual password.

Given the output of a good hash function, it is hard or impossible to reconstruct the input using the hashed value. However, if you are willing to burn some CPU time, it is possible to try every possible password (brute force attack) until you find one such that hashes to the target hash.

We will be using crypt\_r() (a reentrant/thread safe version) of the crypt() function, as our hashing function.crypt\_r() takes three arguments: the string to hash, a salt string, a struct crypt\_data. Make sure to set theinitialized member of your struct crypt\_data before using it for the first time. For example:

struct crypt\_data cdata;

cdata.initialized = 0;

const char \*hashed = crypt\_r("example1", "xx", &cdata);

printf("hash of 'example1' = %s\n", hashed);

hashed = crypt\_r("example2", "xx", &cdata);

printf("hash of 'example2' = %s\n", hashed);

This code outputs the following:

hash of 'example1' = xxPXiOTQGgNxc

hash of 'example2' = xx96r6/l1aOi.

The struct crypt\_data is necessary for crypt\_r(). This is because crypt() needs to store information between invocations, so calling crypt() on multiple threads will cause this information to be inaccurate. crypt\_r()gets around this by storing information in a struct crypt\_data instead. You should check the man page forcrypt\_r, do you need to free the string it returns?

The salt argument ‘flavors’ the string so that when you hash the same password twice, you’ll get a different result (if you use a different salt). For example, we can use part of a user’s username to salt their password before hashing. This way, two users with the same password will not have the same hash stored in the database.

**For the sake of this assignment, always use "xx" for the salt argument.**

You will be given a list of password hashes which you must recover. Each password is a password partial, meaning that we have these two pieces of information:

* The first few letters in their password
* The total length of the password

**All the passwords only contain lowercase letters!**

For example, we may say that a password begins with "hello" and has a total of 8 letters. We know the hashed value associated with this password is "xxsczBXm6z4zA", so we simply have to try hashing each possible password (staring with the prefix provided) until we find one that hashes to the desired value.

Your input will be a file with one line for each password to recover. Each line will contain:

* Username (1-8 characters)
* Password (13 characters)
* Known part of password (plus periods representing unknown characters) (1-8 characters, contains 0-8 lowercase letters followed by 0-8 periods)

These three fields are separated by a single space. Don’t worry about duplicate usernames, duplicate password hashes, or duplicate prefixes. Each line is an independent task.

We will not provide input not in this format.

Example input:

donna xxC4UjY9eNri6 hello...

eric xxqJ7cKzV3v4E zip.....

francie xxGGPN89YLcGY cham....

george xxq5aBqiB66j2 xz....

helen xxhx0AsVpMTMU sysx....

inigo xxHUf9zUctXNA miss....

Version 1

**We will not grade any output which is not the result of a call to a function in format.h**

You will not be graded on a single threaded implementation, but it is **always** a good idea to write a single threaded version of your code before trying to parallelize!

Use multiple threads to speed up the password processing. The main thread will start up a pool of worker threads, then read lines of input from standard input. Each line will be converted to a task which is added to a task queue. The task queue is provided in libprovided.a and queue.h. It is very similar to the synchronized queue you’ve implemented in lab. The worker threads will pull one task from the task queue, then process the task. When a worker thread starts processing a task, it will print the username of the task (use format.h).

When a worker thread finishes a task, use format.h to print the cracked password, along with the index of the thread (starting with index 1), and the amount of CPU time spent working on the password (usegetThreadCPUTime()).

When your main thread finishes reading in lines from the input, we can’t shut down immediately. The worker threads may still cracking some passwords. You need to decide how to cleanly shut all the threads down when there are no more passwords to crack. Every thread must join with the main thread!

After all the worker threads have exited, the main thread will print (this is provided):

* Number of successful and unsuccessful password cracks
* Wall clock time since the program was started (via getTime() in utils.h)
* CPU time used (a sum of the CPU time used in all threads).
* Proportion of CPU time to wall clock time.

By default, the provided code creates 4 worker threads. If a command line argument is supplied to the program, it will use that as the number of worker threads rather than the default.

Example:

$ cat password\_file.txt

donna xxC4UjY9eNri6 hello...

eric xxqJ7cKzV3v4E zip.....

francie xxGGPN89YLcGY cham....

george xxq5aBqiB66j2 xz....

helen xxhx0AsVpMTMU sysx....

inigo xxHUf9zUctXNA miss....

$ ./cracker1 [thread\_pool\_size] < password\_file.txt

Example output:

Thread 1: Start donna

Thread 4: Start francie

Thread 3: Start george

Thread 1: Password for donna is helloaac (3 hashes in 0.00 seconds)

Thread 1: Start helen

Thread 2: Start eric

Thread 2: Password for eric is zipaaazz (676 hashes in 0.00 seconds)

Thread 2: Start inigo

Thread 1: Password for helen is sysxpert (266806 hashes in 1.05 seconds)

Thread 2: Password for inigo is missudad (353552 hashes in 1.32 seconds)

Thread 4: Password for francie not found (456976 hashes in 1.65 seconds)

Thread 3: Password for george is xzzzzy (456975 hashes in 1.75 seconds)

5 passwords recovered, 1 failed.

Total time: 1.74 seconds.

Total CPU time: 5.77 seconds.

CPU usage: 3.31x

The times and order may vary slightly.

**Remember to use appropriate synchronization, and make sure to use crypt\_r.** If you create a new thread for each task (instead of keeping the threads in the thread pool running), you will loose points! (and your implementation will be very slow)

Version 2

**We will not grade any output which is not the result of a call to a function in format.h**

Version 1 works great when there is a long list of passwords that need cracking in parallel, but it’s no faster than a single threaded version when there’s one really hard password that needs cracking. For version 2, you’ll still have a pool of threads, but rather than assigning one thread to each password task, all the threads will work in parallel on each password task.

Example input:

maude xxEe0WApyDMcg a......

jesse xxsJNywggi0lA za......

francie xxGGPN89YLcGY cham....

Example output:

Start maude

Thread 2: Start maude at 77228944 (agnaaaa)

Thread 4: Start maude at 231686832 (atnaaaa)

Thread 1: Start maude at 0 (aaaaaaa)

Thread 3: Start maude at 154457888 (anaaaaa)

Thread 1: Stop after 308332 iterations (found)

Thread 3: Stop after 327747 iterations (cancelled)

Thread 4: Stop after 337977 iterations (cancelled)

Thread 2: Stop after 318970 iterations (cancelled)

Password for maude is aaarocx (1293026 hashes in 1.72 seconds)

Total CPU time: 6.41 seconds.

CPU usage: 3.72x

Start jesse

Thread 3: Start jesse at 154457888 (zanaaaaa)

Thread 4: Start jesse at 231686832 (zatnaaaa)

Thread 1: Start jesse at 0 (zaaaaaaa)

Thread 2: Start jesse at 77228944 (zagnaaaa)

Thread 2: Stop after 945682 iterations (found)

Thread 4: Stop after 968171 iterations (cancelled)

Thread 3: Stop after 966968 iterations (cancelled)

Thread 1: Stop after 911765 iterations (cancelled)

Password for jesse is zagpbuyj (3792586 hashes in 3.24 seconds)

Total CPU time: 12.77 seconds.

CPU usage: 3.94x

Start francie

Thread 4: Start francie at 342732 (chamtnaa)

Thread 3: Start francie at 228488 (chamnaaa)

Thread 1: Start francie at 0 (chamaaaa)

Thread 2: Start francie at 114244 (chamgnaa)

Thread 4: Stop after 114244 iterations (end)

Thread 3: Stop after 114244 iterations (end)

Thread 2: Stop after 114244 iterations (end)

Thread 1: Stop after 114244 iterations (end)

Password for francie not found (456976 hashes in 0.40 seconds)

Total CPU time: 1.53 seconds.

CPU usage: 3.81x

Distribute the work by splitting the search space into equal-sized chunks, one for each worker thread. For example, if there are 3 unknown characters, then there are 26^3 = 17576 possible passwords that need to be tested. With 4 worker threads, you would split the work up like this:

* Thread 1: 0..4393 (aaa..gmz)
* Thread 2: 4394..8787 (gna..mzz)
* Thread 3: 8788..13181 (naa..tmz)
* Thread 4: 13182..17575 (tna..zzz)

When the number of threads doesn’t divide the search space evenly, it’s easy to get off-by-one errors due to integer rounding. We’ve provided functions getSubrange() and setStringPosition() to help you with this.

With all the threads working on the same task, you may want to restructure your thread synchronization a little. Rather than a queue, you may wish to use a barrier.

Startup Task 0.............................. Task 1..............................

main thread: read task | idle | print results, read next | idle | print results, read next

worker threads: idle | computing | idle | computing | idle

↑

barrier

Like with version 1, you may not create new threads for each task. The threads you create at the beginning of the program must be the same threads that compute the last task.

When the main thread reads a task, it should print "Start <username>". When a thread starts processing a task, it should print its index and starting position. As usual, make sure to use format.h. For example:

% echo eric xxqJ7cKzV3v4E zip..... | ./cracker2

Start eric

Thread 1: Start eric at 0 (zipaaaaa)

Thread 2: Start eric at 2970344 (zipgnaaa)

Thread 4: Start eric at 8911032 (ziptnaaa)

Thread 3: Start eric at 5940688 (zipnaaaa)

When a worker thread finds the correct password, it should tell all the other threads to stop working on the task. You can implement this with a simple flag variable that each thread checks on each iteration. Since all the threads are reading this variable and any thread may write to it, you’ll need to properly synchronize access to it.

When the worker threads finish a task, each thread will print the number of passwords it tried and a word describing how its run finished:

* (found) - this thread found the password
* (cancelled) - stopped early because another thread found the password
* (end) - finished with no password found. Note: this can happen if the password was found, but this thread finished its chunk before another thread found the password.

After all worker threads finish each task, the main thread will print the password (if found), the total number of hashes, the wall clock and CPU time spent on that task, and the ratio of CPU time to wall clock time. Note that we have not provided any of the timing print statements in cracker2.

As usual, we have provided a Makefile which can build a release and a debug version of your code. Running makewill compile cracker1 and cracker2 in release mode, as well as a tool called create\_examples (more on this in the next section). Running make debug will compile cracker1 and cracker2 in debug mode, and will also compilecreate\_examples.

ThreadSanitizer

We have also included the target make tsan, which compiles your code with Thread Sanitizer (run cracker1-tsanand cracker2-tsan)

ThreadSantizer is a race condition detection tool. See [this page](http://illinois-cs.github.io/tsan) for more information.

**We will be using ThreadSanitizer to grade your code! If the autograder detects a data race, you won’t automatically get 0 points, but a few points will be deducted.**

Thread status hook

We’ve provided a simple tool to help you when debugging your program. See thread\_status.h andthread\_status.c. We’ve install threadStatusPrint() as a handler for SIGINT. It will print a brief summary of what each thread is currently doing any time you hit ctrl-c. For example:

% ./cracker2 cracker2.in 2 100000 2

Start u0000000

Start u0000001

^C

\*\* Thread 0: semaphore wait at cracker2.c:219

\*\* Thread 1: processing at cracker2.c:269

\*\* Thread 2: processing at cracker2.c:269

To use it:

* #include "thread\_status.h"
* Call threadStatusSet() to describe what the thread is currently doing. The argument to threadStatusSet()should be a string constant. For example:

threadStatusSet("initializing");

...

while (!done) {

threadStatusSet("waiting for task");

task = queue\_pull(task\_queue);

threadStatusSet("processing");

...

}

When threadStatusPrint() is called, it doesn’t print the exact line number that each thread is at. It just prints the line number of the most recent call to threadStatusSet(). So, for more precise reporting, add more calls tothreadStatusSet() to your code.

thread\_status.h contains macros that will redefine calls to common thread synchronization functions so that when a thread is blocking on one of them, its status will represent that (like the “semaphore wait” on line 219 in the example above).

**Note: Since Thread Status is hooked to Ctrl-C, you might need to use Ctrl-D (EOF) or Ctrl-\ (SIGQUIT) to shutdown a running password cracker**

You’re not required to use the thread status tool as part of the assignment, we just thought it might make your debugging easier.

create\_examples

We’ve also provided a small program to create example input files, to help you with your testing. To build thecreate\_examples program, run make create\_examples. To use the program, write it’s output to a file, then use the file as input to a cracker program. For example:

./create\_examples 10 100 150 > my\_examples.in # write the output to a file

./cracker1 < my\_examples.in

To see what the cracked passwords should be, use the -soln flag when running create\_examples (see the usage documentation given when running the program with no arguments).

timing example

CPU time and so called “wall clock” time are not always the same thing. CPU time is defined as “the amount of time your program spends running on a CPU,” and wall clock time is quite literally, the amount of time that would pass on a wall clock (the kind of clock on a wall) between the time a program starts and a program finishes running. These numbers are often not the same!

If your program makes a large number of blocking system calls, it may take 10 seconds to run, but only actually consume 5 seconds of CPU time. In this case, the kernel spent time reading from a file, or writing packets to the network, while your program sat idle.

CPU time can also be much larger than wall clock time. If a program runs in multiple threads, it may use 40 seconds of CPU time, but only take 10 seconds of wall clock time (4 threads, each ran for 10 seconds).

To demonstrate these differences, we’ve provided a program in tools/timing.c which shows an example of both kinds cases.

To compile this program, run make timing, then run ./timing. You should see output like this:

sleep(1): 1.00 seconds wall time, 0.00 seconds CPU time

single threaded cpu work: 0.14 seconds wall time, 0.14 seconds CPU time

multi threaded cpu work: 0.14 seconds wall time, 0.28 seconds CPU time