IC221 pthreads lab Name: Patrick Catren

Spring AY22

**Learning Objectives**:

- Understand how POSIX threads are created and terminated (rejoined)

**Estimated Completion Time**: 1.5 hours **Lab Total**: 100 points

**Submission**: This completed document to the online submission site. No need to submit the C files.

**Example Files**: counters.c crack.c

**Task 1**: counters.c

Review the contents of counters.c. Examine the pthreads code. Understand how threads are created and joined.

Compile and run counters.c

gcc -o counters counters.c -lpthread

./counters

Note how the lpthread flag tells the compiler to link against the pthread library. This is necessary because it is not a 'standard' library. If you include this flag at the beginning, as in:

gcc -lpthread -o counters counters.c

it will cause a compiler error, because the source/target arguments are expected first.

(5) Do the counter threads *execute* in strictly numerical order? Why or why not?

|  |
| --- |
| No they do not, because the OS schedules them under multiple cores so we cannot predict the order in which they will execute. |

(5) If you run the program repeatedly, does the order of thread execution sometimes change? Why or why not?

|  |
| --- |
| Yes, depending on what other process are running the OS may allocate resources differently than it did previous times. |

(10) If one of these threads happened to block, would the other threads block? Why or why not? If you're not sure, uncomment the "lucky number" portion of thread\_function to experiment.

|  |
| --- |
| No, because they are executed independently. |

**Task 2**: crack.c

Complete the crack.c code

The crack.c code is a password cracking program. It uses a brute-force approach to try to find the password that matches a given MD5 hash. The MD5 hash, written in hex, is called a *digest*. These are unsalted hashes. To keep it simple, we'll only uses password characters between 'a' and 'z', and we will crack passwords only of length 5. **Note**: if running on WSL or a VM, ensure you have the ssl library: sudo apt install libssl-dev

Each of you will receive a unique batch of digests to crack, randomly selected based on your alpha code. All passwords are real user passwords, randomly chosen from a published batch of compromised passwords. Your digest file will have the form m999999.digest

In the program's main() function, it opens the file containing the digests and reads them into memory. Edit the appropriate line so the program opens *your* unique digest file, associated with your alpha code.

Compile the program as follows: gcc crack.c -o crack -lpthread -lcrypto. Notice the addition of both the pthreads library and the crypto library that we are linking against.

Run the program: ./crack

The program should take less than a minute to crack all the hashes, but way of brute force. It starts at 'aaaaa' and tries every possible password up to 'zzzzz'. This is not optimally efficient, but that's okay.

(10) Enter the output of the program, showing that it broke all the passwords in your batch:

|  |
| --- |
| Thread 0 Crack digest: ab8165881cd720178aa99eac33d9951d  Thread 1 Crack digest: 94e0b4c7eec58f8aff75c00a131b191f  Thread 2 Crack digest: b843472d0a72b42a68a04adb1f2a75cf  Thread 3 Crack digest: 4768b3fc7ac751e03a614e2349abf3bf  Thread 4 Crack digest: 244a8648a06fdf447cef78d9ee4fee25  Thread 5 Crack digest: 86bfa6b4dd02e3a910db6a0c38ac0c90  Thread 6 Crack digest: e5c316da9475579ef204ab0b370e10a1  Thread 7 Crack digest: f436371de5cdc3cfd1d28396440fa32b  Thread 9 Crack digest: 786563bc11bccf5794051b5779a876d5  Thread 10 Crack digest: 31e10abc4f418ed16ed9e646f5c0bee7  Thread 0 Match: bahan ab8165881cd720178aa99eac33d9951d  Thread 7 Match: calas f436371de5cdc3cfd1d28396440fa32b  Thread 6 Match: desto e5c316da9475579ef204ab0b370e10a1  Thread 10 Match: eliza 31e10abc4f418ed16ed9e646f5c0bee7  Thread 2 Match: jhane b843472d0a72b42a68a04adb1f2a75cf  Thread 1 Match: trant 94e0b4c7eec58f8aff75c00a131b191f  Thread 4 Match: scaup 244a8648a06fdf447cef78d9ee4fee25  Thread 5 Match: ricki 86bfa6b4dd02e3a910db6a0c38ac0c90  Thread 3 Match: tobie 4768b3fc7ac751e03a614e2349abf3bf  Thread 9 Match: yhang 786563bc11bccf5794051b5779a876d5 |

Alpha Code: \_240954\_\_\_\_\_\_\_\_\_\_\_\_\_

(10) Use the time command to time the execution, as follows: time ./crack

How much does the OS report as the *real* time needed to break your digests? Ignore the *user* and *sys* times.

|  |
| --- |
| real 0m56.089s |

Multithreading. Modify the code as follows:

- Add a loop in main() to create NTHREADS pthreads. Use the array statics[i] as an argument, so they will get assigned a meaningful thread number.

- Add another loop in main() that will join them all when complete.

- Comment out the last two lines of main(), which invoke the cracking function using a single thread.

(20) Compile and re-run your code using NTHREADS values of 2, 4, and 8, respectively, and complete the table below. If running a Linux VM, I encourage you to assign the VM at least 4 CPUs for this exercise. If the program hangs, just break out with CTRL-C and note that.

|  |  |
| --- | --- |
| Number of Threads | "Real" time to crack all digests (seconds) |
| 1 | 56.089s |
| 2 | 32.773s |
| 4 | 22.542s |
| 8 | 13.812s |
| 16 | 14.706s |

(10) Run the command lscpu from the command shell to determine the number of cores on the Linux machine you are using. Look at the entry "On-line CPU(s) list" for a range, or multiply "Cores per socket" times "Threads per core". Enter the number you calculated:

|  |
| --- |
| 12 |

(10) What performance trend did you observe, after implementing multi-threading?

|  |
| --- |
| The time decreased significantly as we increased the number of threads until 16. |

(20) Was there a certain number of threads above which no notable performance change was observed? If so, what was the number, and why do you suppose that is?

|  |
| --- |
| Yes, it was 16, which makes sense because we only have 12 cores, so it can only run the 12 at a time and then the four before it can join. |