IC221 pthreads lab Name Ian Coffey

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?

|  |
| --- |
| Threads do not always execute in strictly numerical order. This is because each thread is scheduled individually , and the user application often does not have an explicit way of controlling how these threads are scheduled. It is whatever order the processor receives the thread. So as a result, different threads may execute at different times, and not do so in numerical order. |
|  |

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

|  |
| --- |
| The order of the threads changes on each run. Again this is because the user application does not have control on how each thread is individually scheduled, so each thread executes at a differing time than the others, and seems pseudo random |

(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.

|  |
| --- |
| The others are not blocked even if one blocks. This is because threads run independently of. eachother, and other instances of threads to not care if one just so happened to request input. |

**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: 22b57308c9bd8736b4f9edb78cd7762b  Thread 5 Crack digest: 2c246002a7a30057f30345b8b0a26c78  Thread 2 Crack digest: 137d480e4b8cc9c4d0d74bfbac2b5372  Thread 1 Crack digest: 5e9a034c3bc02beef3331434e7605a40  Thread 3 Crack digest: f325365b5c2d6fe4e9bfbaf47a008e1a  Thread 4 Crack digest: 4d34c4b371f877143fd7c2ede6953a64  Thread 6 Crack digest: 2dfadf1c87039ffa7beca3f732d544c4  Thread 7 Crack digest: 5a5449b040bffee33b263dd12b4ae20e  Thread 8 Crack digest: 12bc88de284dae7d924c18e7fabf856d  Thread 10 Crack digest: 7ddc153d200f3aed4a40985c0f1246f7  Thread 1 Match: bandy 5e9a034c3bc02beef3331434e7605a40  Thread 10 Match: barde 7ddc153d200f3aed4a40985c0f1246f7  Thread 8 Match: droop 12bc88de284dae7d924c18e7fabf856d  Thread 4 Match: platy 4d34c4b371f877143fd7c2ede6953a64  Thread 7 Match: nunch 5a5449b040bffee33b263dd12b4ae20e  Thread 3 Match: turgy f325365b5c2d6fe4e9bfbaf47a008e1a  Thread 5 Match: unrow 2c246002a7a30057f30345b8b0a26c78  Thread 2 Match: trave 137d480e4b8cc9c4d0d74bfbac2b5372  Thread 0 Match: sizal 22b57308c9bd8736b4f9edb78cd7762b  Thread 6 Match: sonam 2dfadf1c87039ffa7beca3f732d544c4 |

Alpha Code: 253672

(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 0m13.657s |

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 | 62.354 Seconds |
| 2 | 33.525 Seconds |
| 4 | 18.276 Seconds |
| 8 | 13.766 Seconds |
| 16 | 14.159 Seconds |

(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:

|  |
| --- |
| 6 cores \* 2 threads = 12 total Threads |

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

|  |
| --- |
| The performance got exponentially better until it stopped getting better at 16 threads |

(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?

|  |
| --- |
| After exceeding 12 threads, the improvement in efficiency plateaued. This is because in the system, there are 6 cores, and each core can handle 2 threads, so there can only be a maximum of 12 threads. It wont throw an error if that number is exceeded, but the performance will stay the same |