IC221 pthreads lab Name Brendan Lewis

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. This computer has a multi core processor, so it schedules different processes at the same time. Therefore while it might seem at first glance they will be procedural and in numerical order, they don’t execute strictly in numerical order. |

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

|  |
| --- |
| Yes the thread execution does sometimes change. It changes because the CPU does not always assign resources the exact same way, and some of the threads finish before the other threads. |

(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 the other threads don’t block because each thread is running separately. So the other threads will continue and finish while the one thread waits. |

**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: cd .

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 1 Crack digest: 082d19ec6e4aa3be445e7840865c3603  Thread 1 Match: sexxi 082d19ec6e4aa3be445e7840865c3603  Thread 1 Crack digest: 02ebe446bd8e8c49173b6fbdddf10b63  Thread 1 Match: henny 02ebe446bd8e8c49173b6fbdddf10b63  Thread 1 Crack digest: f679ca1293f318860df9d5b966e6fac0  Thread 1 Match: allie f679ca1293f318860df9d5b966e6fac0  Thread 1 Crack digest: 7158513bcdad9fbdf2409e8ef89b878a  Thread 1 Match: wolff 7158513bcdad9fbdf2409e8ef89b878a  Thread 1 Crack digest: 86449dd1ea0745feb632b6e70598b133  Thread 1 Match: sabik 86449dd1ea0745feb632b6e70598b133  Thread 1 Crack digest: 927e21dd48ee865ab5490a492caa5e94  Thread 1 Match: senor 927e21dd48ee865ab5490a492caa5e94  Thread 1 Crack digest: 9c01d3aec873ea8a53858875e9a4dda3  Thread 1 Match: nursy 9c01d3aec873ea8a53858875e9a4dda3  Thread 1 Crack digest: 08ea4a7dd7b9b0edcc9d259c164519ad  Thread 1 Match: saidi 08ea4a7dd7b9b0edcc9d259c164519ad  Thread 1 Crack digest: 4f8c0226b4b563061e64ba1b0d86d40f  Thread 1 Match: geste 4f8c0226b4b563061e64ba1b0d86d40f  Thread 1 Crack digest: 72d3196e21a7b6767c189dd76b56ab6b  Thread 1 Match: boily 72d3196e21a7b6767c189dd76b56ab6b |

Alpha Code: 240336

(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.

|  |
| --- |
| 59.957s |

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 | 59.451s |
| 2 | 32.126s |
| 4 | 18.031s |
| 8 | 12.550s |
| 16 | 11.529s |

(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 per socket \* 2 Threads per core = 12 Cores |

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

|  |
| --- |
| Since there were nine hashes in my file, each time I doubled the number of threads I saw a performance increase of almost half the time to crack all digests. The change from 8 to 16 was not super significant because all new threads weren’t used. |

(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 there was no significant increase above 8 threads and that is because of how small the data set is. With only 9 items in the data set there is no significant impact in having more threads, because one of the 8 will finish quickly and pick up the last item in the database. |