NAME: Raymond Shum
DATE: April 6, 2021
TITLE: Lab 5 – Part 5 – Write-Up
DESCRIPTION: This document answers questions found in the Lab 5 instructions. Questions are bolded.

Question 1

Explain what happens when you run this threadHello.c program.

```
Hello from thread 1 with iteration 2
Hello from thread 2 with iteration 3
Hello from thread 3 with iteration 7
Hello from thread 4 with iteration 9
Thread 0 returned
Hello from thread 5 with iteration 0
Hello from thread 6 with iteration 0
Hello from thread 7 with iteration 0
Hello from thread 8 with iteration 0
Thread 1 returned
Hello from thread 9 with iteration 1
Hello from thread 10 with iteration 1
Thread 2 returned
Thread 3 returned
Thread 4 returned
Thread 5 returned
Thread 6 returned
Thread 7 returned
Thread 8 returned
Thread 9 returned
Main thread done.
```

This is sample output from the threadHello_step1.c program.

10 threads are created. As part of their start-up function, *go, each prints the statement “Hello from thread [threadID] with iteration [&i]” and returns. After returning, each thread confirms its return by printing to console. After all threads have completed, the main process confirms by printing “Main thread done.” and returns.

The issue with this version of the program is caused by the unsynchronized and concurrent access of the stack-allocated variable “I”. This lead to the repeated iteration numbers seen in the “Hello from thread…” output.

Can you list how many threads are created and what values of i are passed?

10 threads were created, with the address of “I” passed to each pthread_create() statement at each iteration of the loop.

Do you get the same result if you run it multiple times?

You do not get the same output in the same sequence if you run it multiple times. However, you will see the same issue with repeated iteration numbers.

What if you are also running some other demanding processes (e.g., compiling a big program, playing a Flash game on a website, or watching streaming video) when you run this program?

In this situation, you would still not see the same output in the same sequence when you run the program. However, due to the unsynchronized and concurrent access of the stack-allocated address “I”, you would run into the same issue with repeated iteration numbers. With more cores being utilized, the program may take longer to complete, due to the OS needing to schedule and context switch between more ready processes, with interactive processes (such as games) likely entering at a higher priority.

Question 2

The function go() has the parameter arg passed a local variable. Are these variables per-thread or shared state?

Local variables are per-thread while global variables are a shared state.

Where does the compiler store these variables' states?

Local variables are stored in the associated thread's thread-local stack. Global variables are stored in the program code.

Question 3

The main() has local variable i. Is this variable per-thread or shared state? Where does the compiler store this variable?

The local variable "I" is a per-thread variable that is stored in a thread's thread-local stack.

Question 4

Write down your observations.

The threads will access and print the value of "I" in a non-deterministic order. However, the last print statement "Main thread done." will always execute at the end, due to the use of the pthread_join() statement before it.

You probably have seen that there is a bug in the program, where threads may print same values of i. Why?

The threads will print the same values of "I" because they concurrently access the value stored at the address of the variable. There are no locks or controls that dictate when or in what order the threads access this variable.