**Conner Jordan**

**7-25-2023**

**CST 344 Lab 5**

**Write Up**

**What happens when you run the original program?**

thread runs the go() function, which prints a message containing the thread's identifier and the current value of i at the time of printf().

**How many threads are created and what values of i are passed?**

10 threads are created in the program. However, because the address of i is passed to each thread, not the value of i at the time the thread was created, the value that gets printed can be different from the value of i when the thread was created. This can lead to some threads printing the same value of i, or values of i that weren't present when the thread was created.

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

No. This is because the scheduling of threads can vary between runs, and can be influenced by other processes running on the system.

**What if you are also running some other demanding processes?**

If you are running other demanding processes, the scheduling of threads can be even more unpredictable. The thread scheduler may give more CPU time to other processes, affecting the output of the program.

**Are the** arg **variables in** go() **per-thread or shared state? Where are these variables stored?**

The arg parameter in the go() function is a per-thread variable, because each thread gets its own copy of the arg pointer. These variables are stored in each thread's stack memory.

**Is the** i **variable in** main() **per-thread or shared state? Where is this variable stored?**

The i variable in the main() function is a shared state among the threads, because its address is passed to the threads. It's stored in the main thread's stack memory.

**What is the bug in the program?**

The bug in the program is that multiple threads can print the same value of i, because the address of i is passed to the threads, not the value of i when the thread was created. This means that if the main thread increments i before a thread gets a chance to read it, the thread will print the new value of i, not the value it had when the thread was created.

**Why does this bug occur?**

This bug occurs due to the nature of thread scheduling and the fact that the threads are sharing the i variable. If the i variable is incremented by the main thread before a created thread gets a chance to read it, the created thread will read the new value, leading to unexpected output. The solution to this is to pass the value of i to each thread instead of passing the address of i.