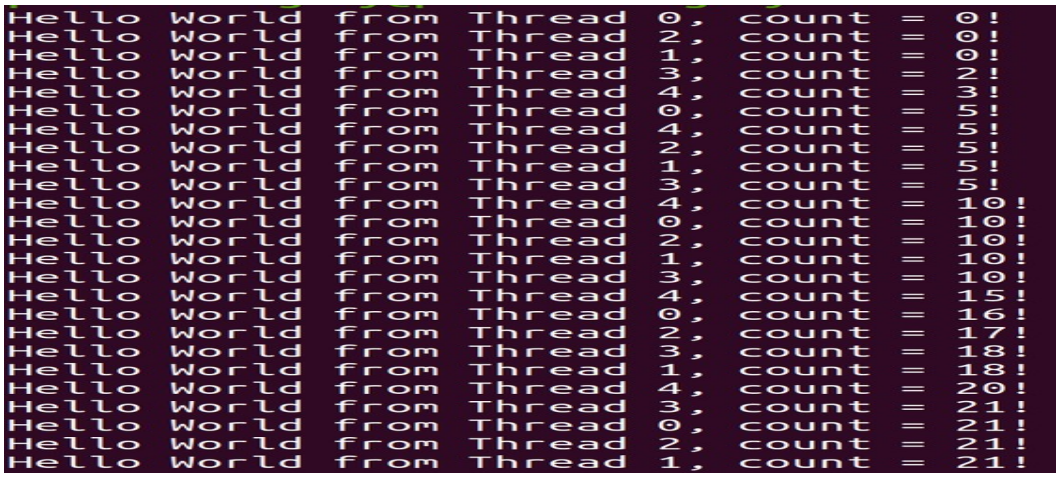
1.1

- **What is the expected output of the above program?** 5 threads in total, each increasing the value of the global variable counter by 1 in the for loop that loops 5 times inside the PrintMessage function, so counter starts at 0 and should be 24 after the code has executed.
- **Run the code and explain the output.** The output is not as expected because the value of count is not incremented by 1 for each new iteration of the loop inside PrintMessage. The order of the thread-ids are also crashing, where it doesn't finish with one thread before another starts using the PrintMessage function



- **What caused the discrepancy between the expected and real outputs?** The threads are colliding with each other. There is a lack of synchronization between the threads. Before the first thread is finished with the increment of the global variable counter, other threads also start to increment the value of counter. In the above screenshot we see that for example three threads are colliding and have all set the value of counter to 0.
  If we lock the part of the code where the loop is happening, we can assure that other threads can't interfere with the current thread until it has unlocked the part of the code that was locked to the thread.

1.2: I've fixed the code in my 1.2.c file by adding 5 lines of code; each of the 5 additions are commented in the code so you can see where.

2.1

- **Try to analyze above code without running and estimate what the expected output would be.** Two threads; thread 1 increasing the value of the global variable counter from it's original value of 0 by 1 in a loop that loops 10 times. After that thread two should take the value of counter and increase it another 5 times. So thread 1 would increase it with 1, from 0-9; then thread 2 will increase it by 1, from 9-14.
- **What is the actual output?** Thread2 is not waiting for thread1 to finish before starting it's loop to increase the value of counter.
- **Use what you have learnt from this lab to fix the program in order to have the expected output.** Fixed in the file called 2.1.c.
- **What would be the difference between the output of this exercise compared to that of**
- **Exercise 2?** Here each thread is operating in it's own function, so they don't have to wait for each other. If they were both using the same function, then it would be necessary for the first thread to unlock before the second thread can start locking, but here they run in separate functions and don't have another thread that locks the code to wait for. The way to solve it is by conditional variables where we don't let the waiting thread start until the condition is met. So as long as the condition isn't met, it has to wait. The thread sending the signal can change the value of the conditional variable to the value that will activate the waiting thread when it has finished doing what it must do before activating the waiting thread.