Ryan Hoang

CECS 326 - Sec 1

Dr. Hailu Xu

February 21, 2021

Project 1: Multithreaded Programming and Synchronization

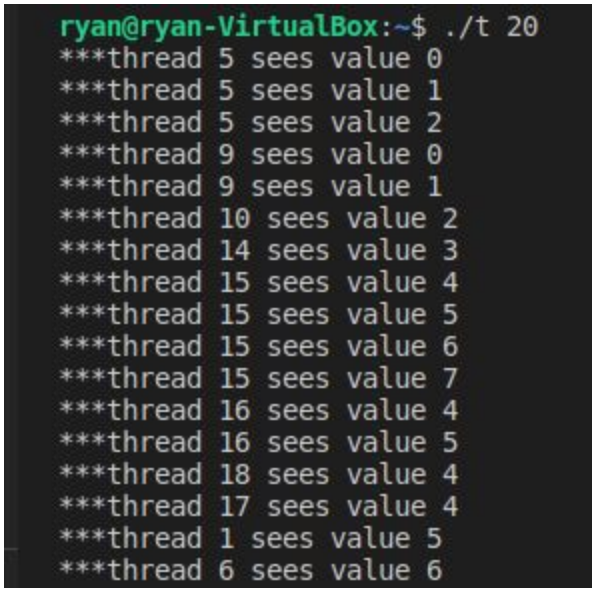
Introduction:

The purpose of this lab is for us to learn the concept of pthread through writing our code in C and running it on Linux through a terminal in a virtual machine. Our main focus is to understand what happens when a pthread is run with synchronization and without synchronization. This project is broken up into two parts as instructed in the lab directions, the first part is to write the code and run it without synchronization and the second part of the lab is to modify the first part so it can run the code with synchronization. We are running the pthread library in Linux which allows us to also learn how the Linux operating system works.

Result and Discussion with Analysis:

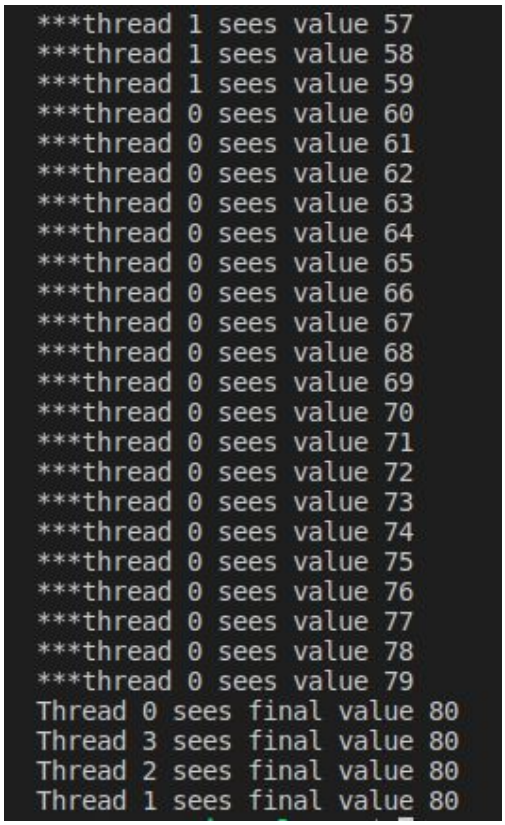
Part 1 of the project asked us to conduct simple multithreaded programming without synchronization. First, we had to acquire the input directly from the command line. We made sure to add an error message if the user forgot to add their input in the command line. Next, we had to make sure the input would be a number and not a word. To accomplish this we would have atoi() a function in C where it would take the input and convert it to a number. We also added an if statement to check if it would be a valid number. If a word was inputted, then it would give a negative number, which our if statement took into account and would display an error message. Once a valid input is taken, we can now create our pthread array with the

correct number of pthreads the user wanted. We would then use the function that was given to use in the project with some tweaks. We had to convert the pointer into an integer to print out which thread was seeing which value. To test this, we would create an integer variable to see if it functions properly. In our final code, we would add (int) on the which value to remove unnecessary coding. From our outputs, we would see almost consistent results from values 1-3. The threads would loop a total of 20 times each, so with basic math, we should determine the final value being the number input times 20. Although this was not always the case. When using values 4 and above we would soon see inconsistency in the data. At times we would reach our final value but most of the time the output would be less than expected. Also since we did not have synchronization during this time, each thread would independently print out the last value they see at random times. Another thing to note was with no synchronization, the values would repeat from the same values. This first step lets us see how threads operate with global variables and within themselves. Our sample output is:



According to our result from the output with no synchronization, the thread is inconsistent due to using the usleep() function, and the value each thread saw is repeated. It's like first come, first serve to whichever pthread reaches the variable called sharedVaraible first.

Part 2: In the second part of the project it would ask us to edit our code from step 1 to add synchronization to our threads. First, we need to examine the code, to understand how the threads were working in the system. With the resources linked from the projects, we noticed that we had to familiarize ourselves with mutexes and barriers. Also, we learned how to add an #ifdef SYNC method to have both the synchronization and non-synchronization code in one file. The method would be used when compiling the code using a preprocessor command(-D SYNC). Although it does have synchronization, the pthreads may look random due to the usleep(500). Before implementing #ifdef SYNC, our pthread output was more consistent and can be viewed below.



Based on the result that was provided from the output the threads have to wait because the mutex would be used to let a single thread enter the critical section while making sure to stop any other thread to pass until the primary thread passed the unlock section for the mutex. This

was used to increment the shared variable in an orderly fashion. Next, we would use the barrier to hold the finished threads until all the threads have gone through the critical section successfully. Once all the threads finished, the barrier would unlock and let the threads see the final value which would be the same for all the threads. After executing the synchronization part and the non-synchronization, the difference that we saw is that we used a mutex lock and unlock, barrier, and a flag variable to call the synchronization part. We implemented it this way because it was the most efficient choice for us to get the desired output. In order to make the code easier to run and compile, we made a MakeFile to compile the non-synchronization as well as the synchronization, saving the user from creating it themselves. All that's left would be the user to run the code with their respective names and proper number input.