Project 1: Multithreaded Programming and Synchronization
Lab Report

Antonio Hughes
Sandra Chavez
CECS 326 - Sec 04
Prof. Hailu Xu
09 - 27 - 2020

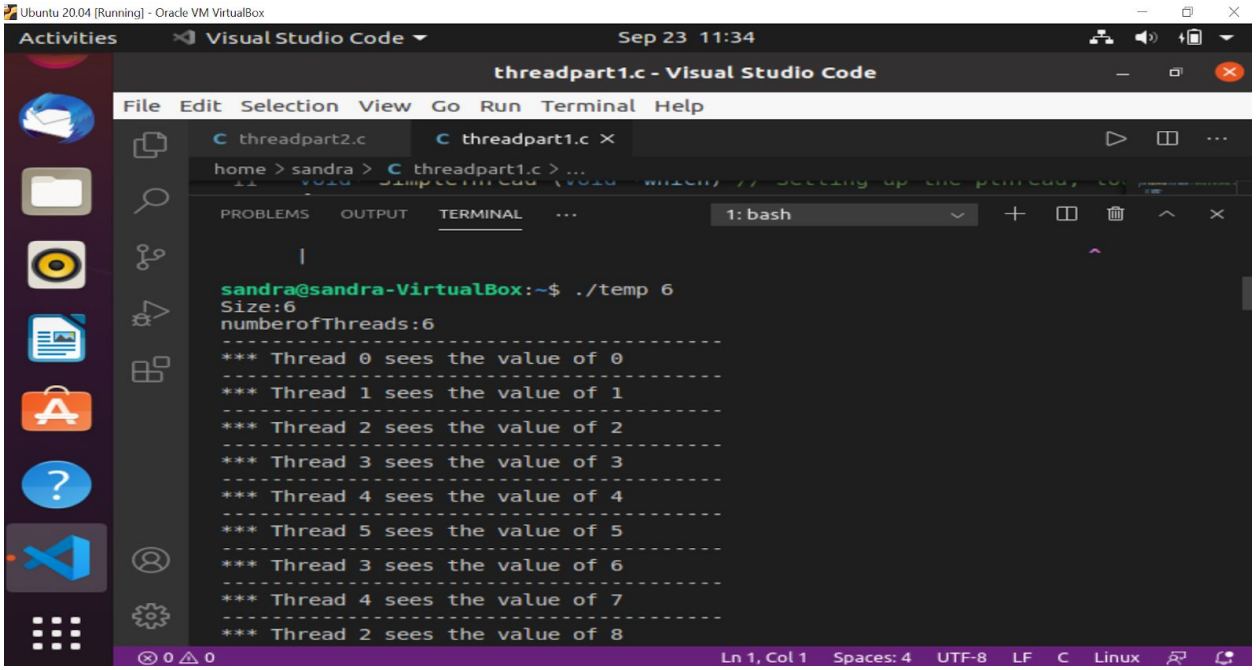
Abstract

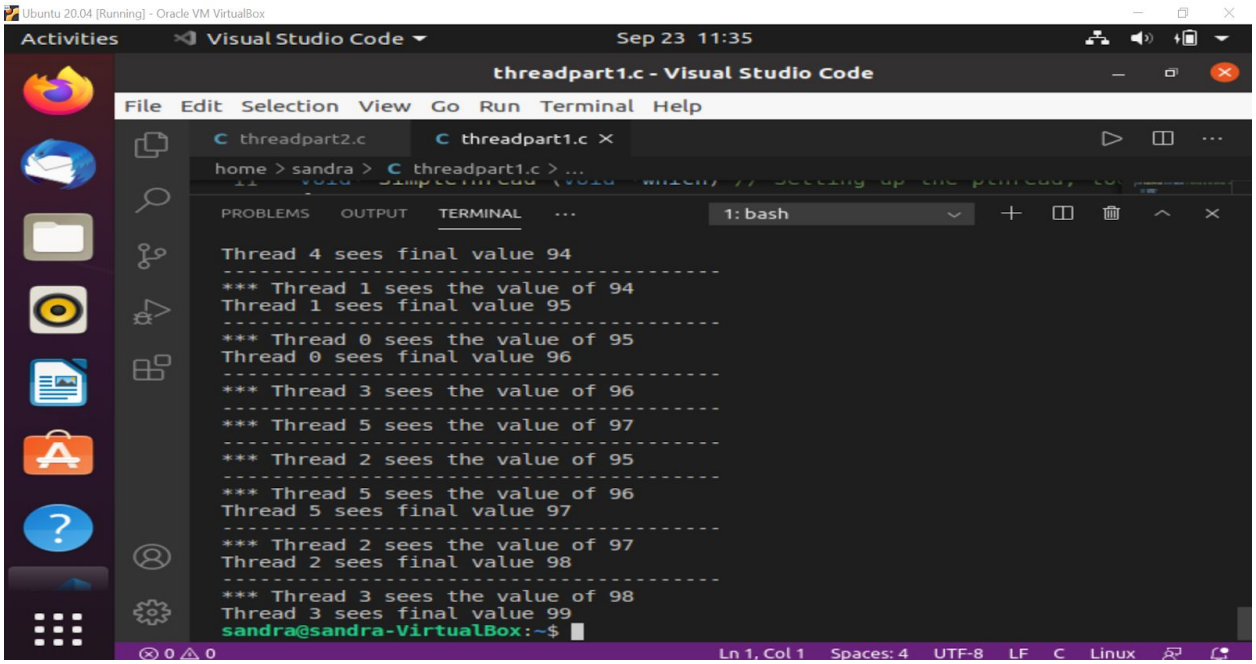
This first project focused on process management and implementing the library Pthreads in Linux. The exercise in this project includes writing a program that creates multiple threads. This project was split up into two parts: the first without synchronization and the second with synchronization introduced into the code. With no synchronization in the first part, the concurrent access to the data resulted in inconsistencies in the final values. The part was then configured to maintain data consistency by using synchronization tools in the form of Pthreads mutex.

Purpose

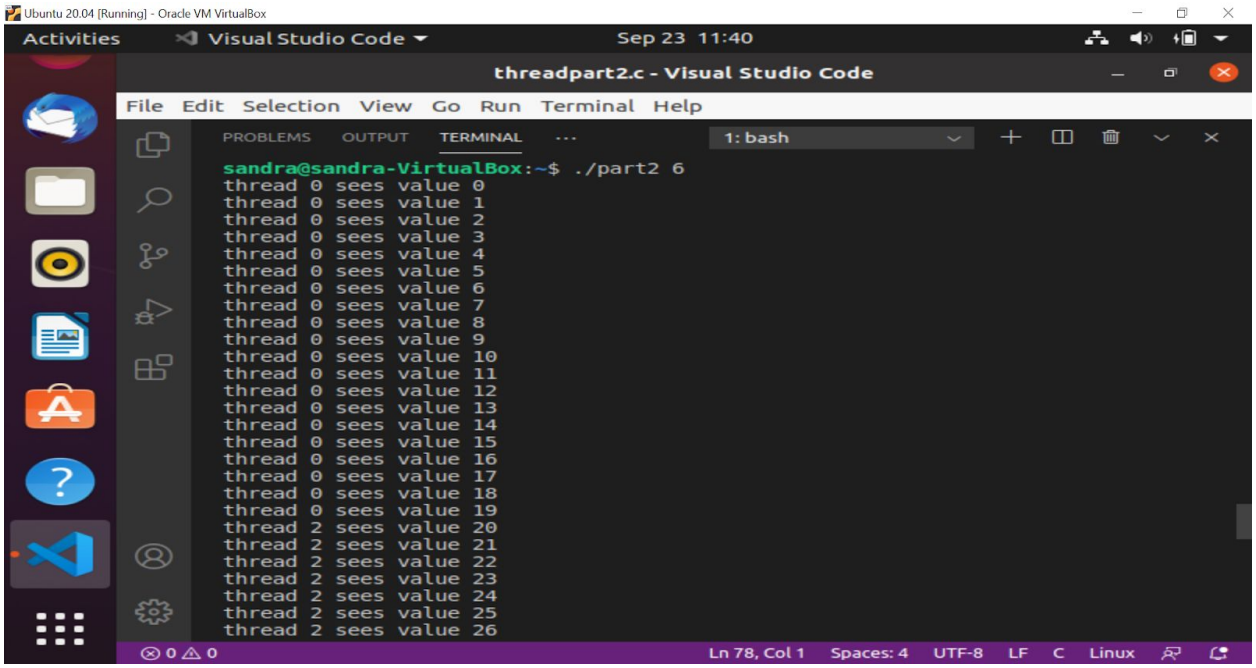
The purpose of this project is to demonstrate the effects from multiple threads performing unsynchronized access to shared data and later, correcting those effects by synchronizing access to the data.

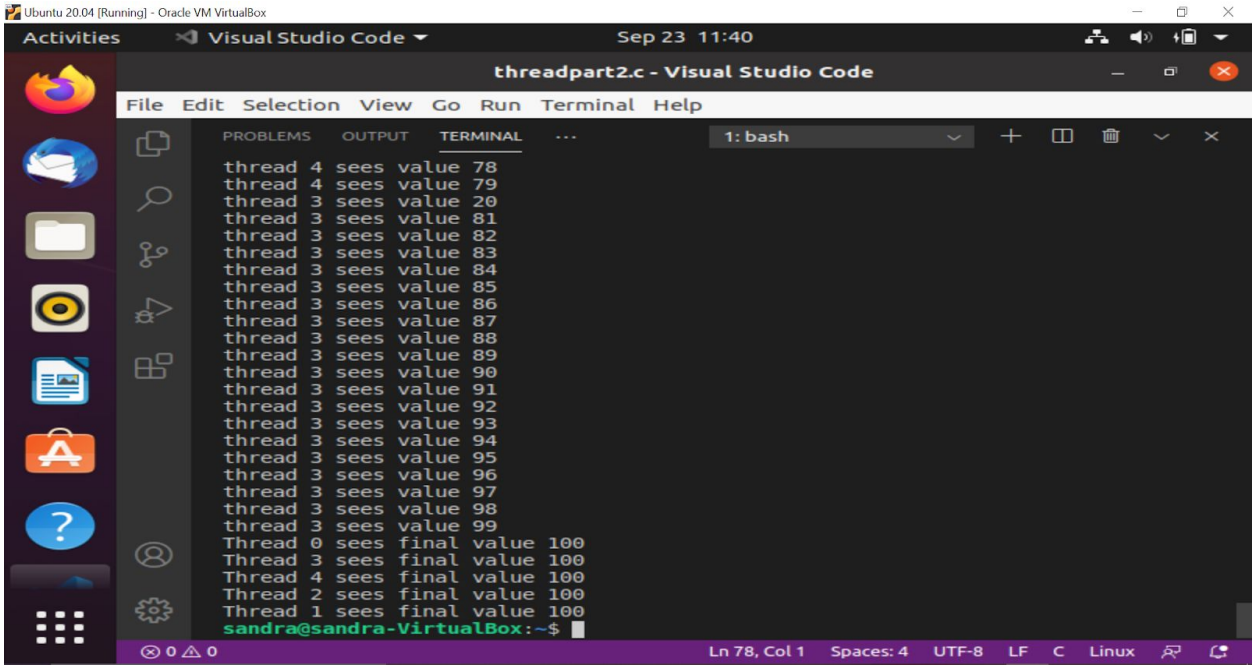
Results





The program in the screenshot is running without synchronization. Due to no synchronization, the thread number displayed is inconsistent, as if they are running independent of one another.





The program in this screenshot is running with synchronization. Due to synchronization, the threads have to wait before each thread can process the final value.

Contributions by group members:

- *Antonio Hughes*
    - *Wrote program that synchronized the access of the shared data (part 2)*
    - *Made the MakeFile*
    - *Made the ReadMe File*
    - *Debugging the second part*
    - *Worked on lab report*
- *Sandra Chavez*
    - *Wrote program without synchronization (part 1)*
    - *Debugging the first part*
    - *Worked on lab report*

Discussion/Analysis

As shown in the screenshots above, there are differences in the consistency depending on whether the flag "-PTHREAD_SYNC" is included in the command to compile the C code. We can observe that in the first part of our lab report, running the code without synchronization gives us multiple threads that arrive at different final values in their respective time frames. This is based on what we entered into the terminal for how many threads we were creating. As we enter a higher number of threads, it is evident what is happening in our code for the first part. The difference we observed in the second part of the lab, was the use of synchronization using a mutex barrier, mutex lock, and flags to call for our threads as it's running. The mutex barrier will allow the threads we create to all execute in an orderly fashion, while the mutex lock ensures two or more threads do not execute simultaneously in its more important phase in processing. The flag is used as an identifier when we want our threads to be processed in sync. The #ifdef and #endif are for defining a conditional statement. If the statement does not exist in the first instance, then it is defined then. That is the flag that we are calling when we compile the program. Remember we are calling the flag " -D FLAG" in our MakeFile as the identifier for this. When that is called then the program runs with synchronization and we are able to see that the threads "slowly" reach the same final value at the end of its process. We believe that processing threads with sync is preferred so that we can have an organized execution running on our most advanced systems. Without this core concept, a lot of different thread executions would be scrambled all the time which could result in data loss or being misused, which is not good for the user and the system.