Parallel and concurrent Programming, Network Programming

1. Create Simple Client Server Application using TCP Socket where server issue a command which will be executed at the client slide as a process of remote command execution

2. Write a Socket-based Python server program that responds to client messages as follows: When it receives a message from a client, it simply converts the message into all uppercase letters and sends back the same to the client. Write both client and server programs demonstrating this.

3. Write a ping-pong client and server application. When a client sends a ping message to the server, the server will respond with a pong message. Other messages sent by the client can be safely dropped by the server.

4. Write a Socket based program server-client yo simulate a simple chat application where the server is multithreaded which can serve for multiple clients at the same time.

5. Online Chess

Server Side

- 1) Start server
- 2) Client A connects
  - Client sends handle
  - Server sends color
- 3) Client B connects
  - Client sends handle
  - Server sends color
- 4) Game starts!
- 5) Server sends turn

Client Side

- 6) If clients turn
  - Client sends move
  - Server sends OK/Illegal
- 7) Else
  - Server sends opponent move
- 8) When checkmate
  - Server sends winner
- 9) Close
- 10) Server returns to wait for next game

6.Modify the provided ReverseHelloMultithreaded file so that it creates a thread (let's call it Thread 1). Thread 1 creates another thread (Thread 2); Thread 2 creates Thread 3; and so on, up to Thread 50. Each thread should print "Hello from Thread <num>!", but you should structure your program such that the threads print their greetings in reverse order. When complete, ReverseHelloTest should run successfully. It's critical to note, though, that passing the test isn't sufficient for succeeding at this problem. You could pass the test just by writing code with a loop to count down. You've got to do this correctly via threading

7.In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem
To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
Solution
The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

8.**Problem Statement –** We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

9.**The Dining Philosopher Problem** – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.
There are three states of philosopher : **THINKING, HUNGRY and EATING**. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

10. Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**
Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time

- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read