**Assignment on Socket Programming**

1. Implement producer consumer example. The problem describes two processes, the producer and the consumer, who 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. The problem is 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. The solution for 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. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Write a server client program, where producer and consumer are client processes and the server program has to monitor these.

1. Illustration of the dining philosopher’s problem.

Five silent philosophers sit at a round table with bowls of [spaghetti](https://en.wikipedia.org/wiki/Spaghetti). Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks. Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed. The problem is how to design a discipline of behavior (a [concurrent](https://en.wikipedia.org/wiki/Concurrency_%28computer_science%29) [algorithm](https://en.wikipedia.org/wiki/Algorithm)) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Problems

The problem was designed to illustrate the challenges of avoiding [deadlock](https://en.wikipedia.org/wiki/Deadlock), a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

* think until the left fork is available; when it is, pick it up;
* think until the right fork is available; when it is, pick it up;
* when both forks are held, eat for a fixed amount of time;
* then, put the right fork down;
* then, put the left fork down;
* repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available, vice versa. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

[Resource starvation](https://en.wikipedia.org/wiki/Resource_starvation) might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of [livelock](https://en.wikipedia.org/wiki/Deadlock#Livelock). If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

[Mutual exclusion](https://en.wikipedia.org/wiki/Mutual_exclusion) is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of [concurrent programming](https://en.wikipedia.org/wiki/Concurrent_Programming).

Write a solution using server-client where each philosopher is a client and write methods at the server program to give access to the forks.