CS6550 Advanced Operating Systems
Homework Assignment 3

Question 1: The *dining philosophers' problem* is a legendary problem given by Dijkstra:
There are *n* philosophers sitting at a round table in a Chinese restaurant. The restaurant provides only *n* chopsticks, one chopstick between every pair of adjacent philosophers. A philosopher can only eat while holding both chopsticks to his/her left and right. Chopsticks must be picked up and put down one by one. The problem is to design a program to regulate philosophers' behavior such that everybody obeys the same rule and nobody starves to death.

Open any OS textbook and you may find discussion of this problem. Now think the problem in a distributed setting, where each philosopher is a process and can only communicate with his/her left and right neighbors (so that the system is a ring). Propose a solution to the dining philosophers' problem.

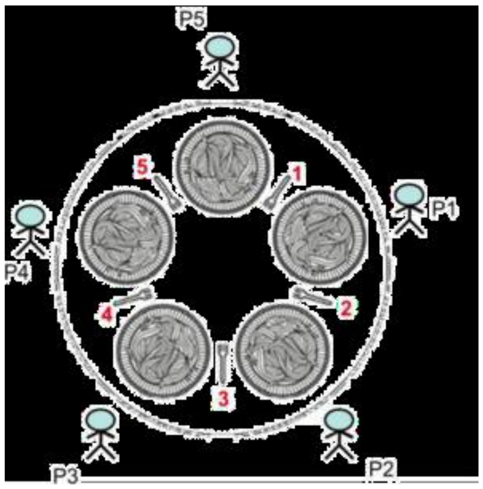
Solution:

The Dining Philosopher Problem states that n philosophers seated around a circular table with one chopstick/fork between each pair of philosophers. There is one chopstick/fork between each philosopher. A philosopher may eat if he could pick up the two chopsticks adjacent to him. If the Philosopher picks only one chopstick he couldn't eat.
There are three kinds of problems that may occur due to various scenarios and they are Deadlock, Live Lock and Starvation.
Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
In this dining philosopher's problem when all the philosophers take one chopstick it leads to a deadlock since a philosopher needs two chopsticks to eat.
In this every process is in waiting state.
Live Lock is same as deadlock, except that the states of the processes involved in the live-lock changes continuously with regard to one another.
It is a special case of resource starvation.
In this every process is in running state continuously.
Starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work.
It is problem which is related to both deadlock and livelock.
In this Dining Philosophers problem out of the n philosophers only two or three philosophers gets the chance to eat whereas the other philosophers are let to starve.
But in the Distributed systems, to solve this problem we have a solution that was given by Chandy-Misra.

- There is a chopstick/fork between each philosopher.
- A philosopher must pick it's two nearest chopstick/fork in order to eat.
- He can only pick one fork/chopstick first and next pick the second one.
- To run this case, we should write an algorithm for allocating the resources among the philosophers.
- We can get a solution, free from deadlock and free from starvation.

- we need to distinguish among three states in which we may find a philosopher.

    - **THINKING –** He doesn't want any access to either forks.
    - **HUNGRY –** He want to enter into critical section.
    - **EATING –** Got both the chopstick/forks and entered the section.



- To initial solution we should make each of philosophers to follow the below given protocol:

```
public void run()
{
        try{
                while(true)
                {
                        thinking();
                        if(test(id))
                        {
                                synchronized(left)
                                {
                                        pickUpLeftFork();
                                        synchronized(right)
                                        {
                                                pickUpRightFork();
                                                eat();
                                                putDownRightFork();
                                        }
                                        putDownLeftFork();
                                }
                        }
                }
        }
```

- **What method did you use to ensure that one version of your program was susceptible to starvation but not deadlock?**
  - Starvation occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.
  - In this program, we used synchronized mechanism with testing the neighbor philosophers state and sleeps the process for random period of time**.**

code:

```
        public void eat()throws InterruptedException
        {
                state.status[id]="eatting";
                System.out.println(getName()+" eatting ");
                sleep(((int) (Math.random() * 150)));
        }
        public void thinking()throws InterruptedException
        {
                state.status[id]="thinking";
                System.out.println(getName()+" thinking ");
                sleep(((int) (Math.random() * 100)));
                state.status[id]="hungry";
                System.out.println(getName()+" hungry ");
        }
        public boolean test(int i)
        {
                if(!state.status[(i + 1) % 5] .equalsIgnoreCase( "eatting")&&!
                state.status[(i + 4) % 5] .equalsIgnoreCase( "eatting") &&
                state.status[i].equalsIgnoreCase("hungry"))
        }
        return true;
}
```

- **What method did you use to ensure that one version of your program was susceptible to deadlock but not starvation?**
  - A deadlock is a situation where the progress of a system is halted as each process is waiting to acquire a resource held by some other process.
    - In this situation, each of the Philosophers has acquired his left fork, but can't acquire his right fork, because his neighbor has already acquired it. This situation is commonly known as the circular wait and is one of the conditions that results in a deadlock and prevents the progress of the system.
  - We model each of the forks as generic Java objects and make as many of them as there are philosophers. We pass each Philosopher his left and right forks that he attempts to lock using the synchronized keyword.
    - Running this code results mostly All the Philosophers initially start off thinking, and we see that Philosopher 1 proceeds to pick up the left and waits because the sleep() method is invoked, during this time next philosopher picks the left fork which is right of the first philosopher . This circular wait continues to all the philosophers that causes dead lock.

code:

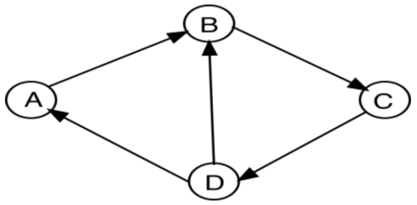
```
public void eat()throws InterruptedException
{
        System.out.println(getName()+" eatting ");
        sleep(100);
}
public void thinking()throws InterruptedException
{
        System.out.println(getName()+" thinking ");
        sleep(100);
}
```

- **What method did you use to ensure that one version of your program was not susceptible to starvation OR deadlock?**
  - This program is free from both starvation or deadlock in this solution imposes the restriction that a philosopher may pick up his or her chopsticks(FORK) only if both of them are available (ie)Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating
    - (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING)
  - The condition is as follows
    - (state[(i + 1) % 5] != eating && state[(i + 4) % 5] != eating && state[i] == hungry)
  - We also imposes Semapore and synchronization to lock the process when both the neighbors are not eating

Code:

```
public void eat()throws InterruptedException
{
        state.status[id]="eatting";
        System.out.println(getName()+" eatting ");
        sleep(((int) (Math.random() * 500)));
        System.out.println(getName()+" completed eatting ");
}
public void thinking()throws InterruptedException
{
        state.status[id]="thinking";
        System.out.println(getName()+" thinking ");
        sleep(((int) (Math.random() * 500)));
}
public void hungry()throws InterruptedException
{
        state.status[id]="hungry";
        System.out.println(getName()+" hungry ");
        sleep(((int) (Math.random() * 500)));
}
```

Question 2: Consider the following network where links are directional and FIFO.



To record the global state, node A sends an agent. The agent visits the processes in the order A-B-C-D-A. At each node, the agent collects local state. When the agent returns to A, it will have collected the global state of the system.

Will this approach work? Briefly justify your answer. If your answer is no, then how would you modify the algorithm so that the collected global state is correct?

Solution:

This algorithm will not work because, once when the agent reaches the starting node A then the other local states will be changed.
We can explain the above solution by using **Chandy-Lamport Algorithm.**

- Here the message 'I' which can be passed from any node in the given network, there is no-particular order that the agent should follows for the message 'I' to follow in the network.
- Here the agent follows a given order as particular mentioned like A-B-C-D-A.
- When the agent records the local state of the initiator node A, travels to node B, then to node C and then it reaches node D.
- Once if the agent reaches node D. Here in node D there is a possibility to reach node A or node B by the agent.
- Since it reaches after collecting all the local states and making a global state as we mentioned that it reaches A.
- There is an approach to this algorithm may fail because, what if there is a message in node D to change the state of node B. when the agent is at node C.
- Hence when the agent reaches the initiator then the local state of B would have changed making the global state to be wrong.
- We can modify this algorithm in such a way that each-and-every node records its own state before it changes its state by message 'I'.
- When the agent comes to record the state, it takes the local state that was recorded by each node and when it goes in the predefined way. It would have collected the global state of the system.

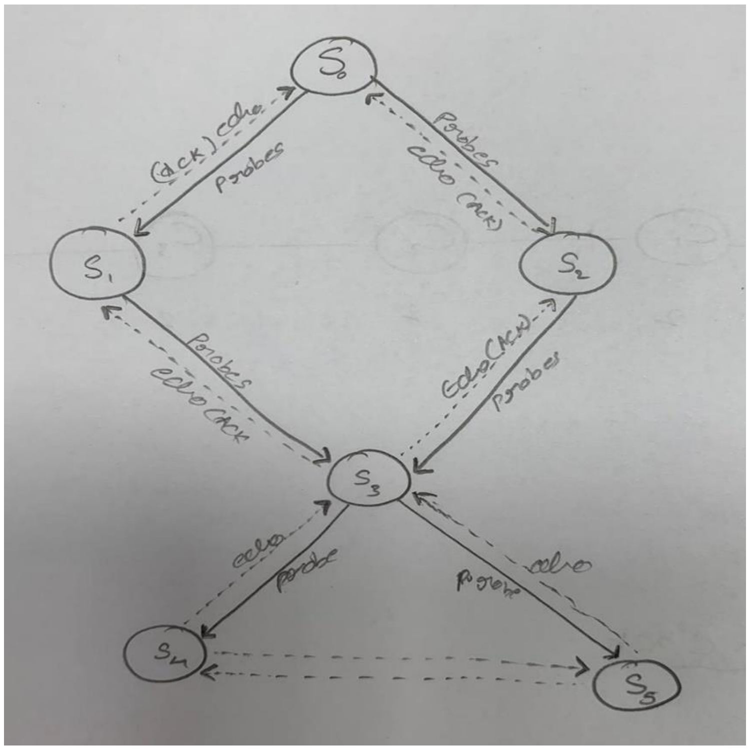
Hence by using this algorithm each and every node record its state before it changes its state due to the message 'I'. we can correct the time interval before making a global state.

Question 3: Consider a network whose topology is an undirected graph *G*= (*V*,*E*) . Each node *i* ∈*V* stores an integer *x*(*i*) . Design a probe-echo algorithm to compute the smallest value of *x*. The algorithm starts when an *initiator node* sends out a *probe* to each of its neighbors, and ends when the initiator receives an *echo* from each neighbor. When the algorithm terminates, the initiator of the algorithm should know the smallest *x*. Briefly argue why your solution will work.

Solution:

There are two kinds of nodes in this algorithm which are initiator acts as the root and non-initiator acts as an internal or a leaf node in the spanning tree. Initiator node is responsible for initializing and finalizing of spanning tree construction.

Here the graph is connected, and the termination of the algorithm would take place when an initiator receives echo (acknowledgement) from all the nodes in the network



The above figure shows that each-and-every node communicate each other. The nodes communicate by using 'Probes' and 'Echo'

The probes are sent to adjacent node form the initiator. Then the adjacent nodes sent back the acknowledgement.

When the node S0 send probe to message S1 and S1 send the echo reply message to S0.

- The node S0 wants to send a message 'I' to all other nodes (S1, S2, S3, S4, S5).
- Assume S0 knows a spanning tree, (S1, S2, S3, S4, S5) rooted at S. Send 'I' and (S1, S2, S3, S4, S5) to each child in (S1, S2, S3, S4, S5). Each node forwards 'I' only to children in tree (S1, S2, S3, S4, S5). 'I' received once and only once by each other process.
- This tree knows each-and-every node of its neighbors. S0 sends 'I' to each child (S1, S2, S3, S4, S5). When node receives message 'I' it forwards it to all its neighbors.
- Number of echoes expected at each node = number of probes sent.

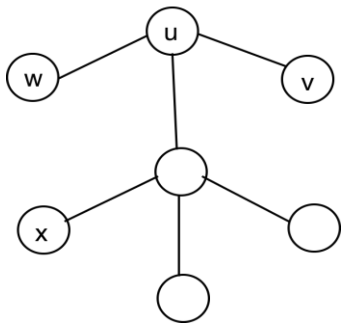
```
define N: integer {number of neighbors is Pi}
Ci, Di: integer
```

```
initially parent = i, Ci=0, Di=0
{for the initiator}
send probes message to each neighbor;
Di:=Ni {number of neighbors};
do Di ≠ 0 and echo --> Di:= Di–1 od
{for a non-initiator process i}
do
probe and parent = i and Ci=0 --> Ci:=1;
parent:= sender;
if i is a leaf --> send echo to parent;
i is not a leaf --> send probes message to non-parent neighbors;
Di:= no of non-parent neighbors;
fi
Echo --> Di:= Di-1;
Receive probe message from j;
If parent i ≠ j then send echo(ack) to j
Receive echo(ack);
D i := D i -1
Return when Ci =1 and Di = 0;
Send echo(ack) parent i;
 Ci :- 0;
probe and parent ≠ i --> send echo(ack) to sender;
Ci=1 and Di=0 --> send echo(ack) to parent; Ci:=0;
Od
```

The algorithm terminates when a msg has been sent and received through every edge exactly once and for all nodes, Ci=0 and Di=0
A node with Ci=1 can receive a signal from a node, but it promptly sends an acknowledgement.

Question 4:

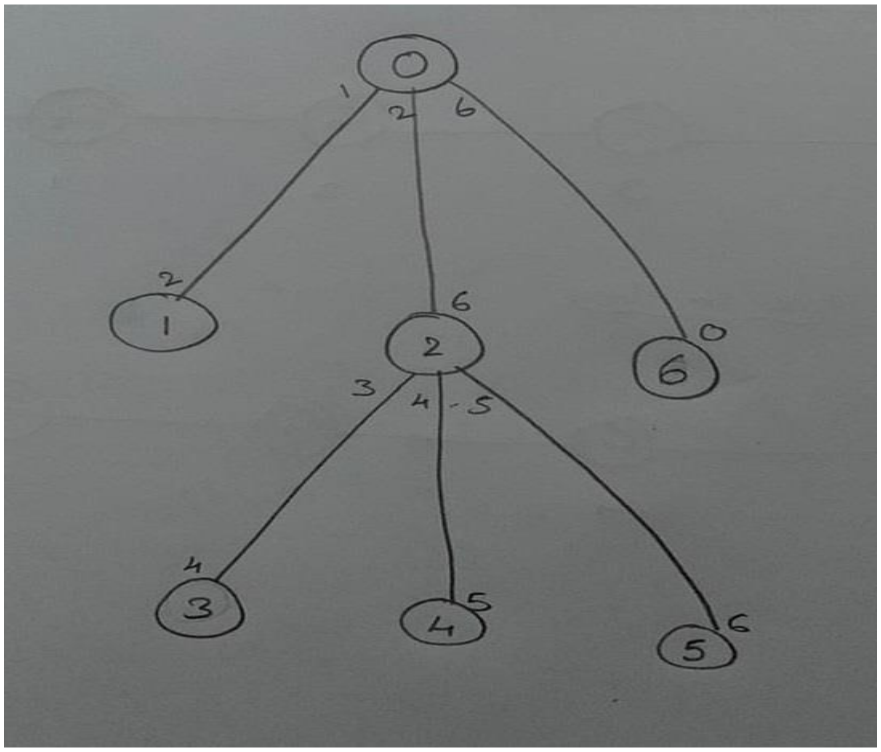
Produce a labeling of the nodes and the ports of following graph for the purpose of interval routing. Begin by labeling node u as 0. Check the correctness of these labels by verifying the routes between different pairs of nodes, for example, x sends message to v.



Solution:

If we want to send a message from node 3 to node 5. Then we can send message from node 3 through the port 4 to node 5.

If we want to send a message from node 3 to node 6. The we can go to node 2 port 6 then we will go to node 0 port 6 then we go to node 6 it is the destination node for the message.



References:

- https://gist.github.com/Alexey-N-Chernyshov/16198f75b284191bf2406d27eaad6b23
- Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach.* 2nd Ed. Chapman/CRC 2014.
- Professor's slides from CS-6550.
- https://www.academia.edu/3141815/The_Chang_Roberts_Distributed_Spanning_Tree_Algorithm_Simulation_on_TinyOS_Simulator_in_Wireless_Sensor_Networks
- http://homepage.cs.uiowa.edu/~ghosh/graph.pdf
- https://www.cs.princeton.edu/courses/archive/fall16/cos418/docs/P8-chandy-lamport.pdf

- https://www.inf.ed.ac.uk/teaching/courses/ds/handouts-2012-2013/part-3.pdf
- https://www.eecis.udel.edu/~cshen/361/Notes/chandy.pdf
- https://pdfs.semanticscholar.org/b714/4a146c807093b8d30ecc3ab09d7dc49b9d98.pdf
- https://stackoverflow.com/questions/19315732/dining-philosophers-chandy-misra-approach-how-does-it-avoid-a-deadlock
- Discussed with fellow mates.
- https://en.wikipedia.org/wiki/Dining_philosophers_problem#Chandy/Misra_solution
- https://www.engr.mun.ca/~theo/Courses/cp/pub/cp9-paradigms.pdf
- https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/
- https://github.com/clenk/School-Projects/blob/master/COS%20326/src/DiningPhilosophers.java
- https://www.baeldung.com/java-dining-philoshophers
- https://www.math.uni-hamburg.de/doc/java/tutorial/essential/threads/deadlock.html
- https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/