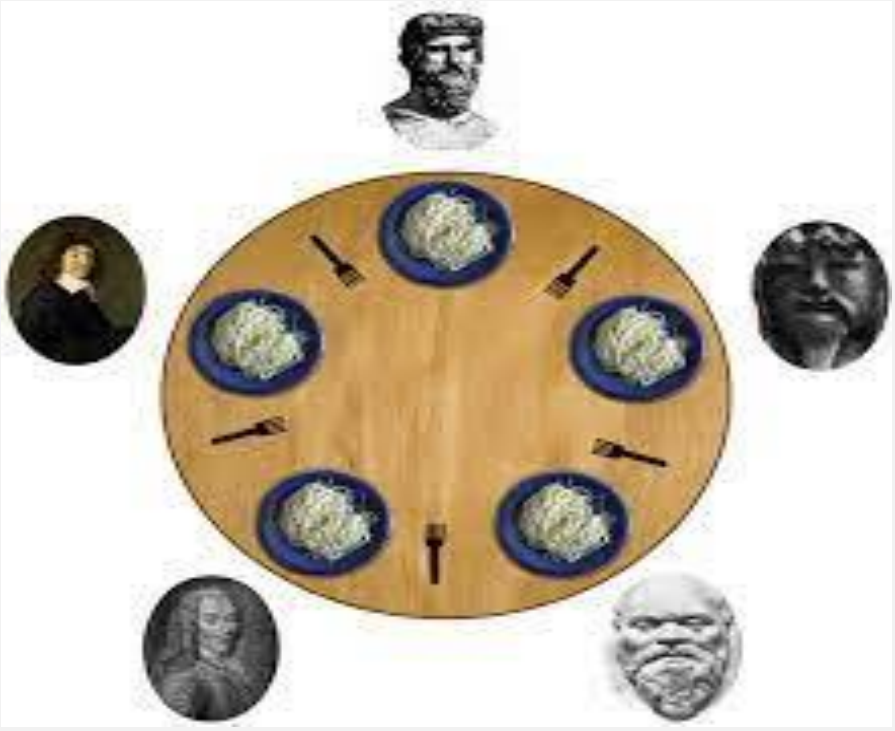
The dining philosophers problem

The definition of the problem:

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of rice is placed at the center of the table along with five chopsticks for each of the philosophers.

Shared data:

- 5 chopsticks.
- Bowl of rice.



Constraints and Condition for the problem :

1. Philosophers can eat or think not both of them.
2. Philosopher needs two chopsticks in order to eat.
3. we have to design pre and post protocols which ensures that a philosopher only eats if he had two chopsticks.

The algorithm to solve Dining Philosophers problem

```
do {
    wait (chopstick[i] ); // left chopstick
     wait (chopStick[ (i + 1) % 5] ); // right chopstick

                //  eat

     signal (chopstick[i] ); // left chopstick
     signal (chopstick[ (i + 1) % 5] ); // right chopstick

                 //  think

} while (TRUE);
```

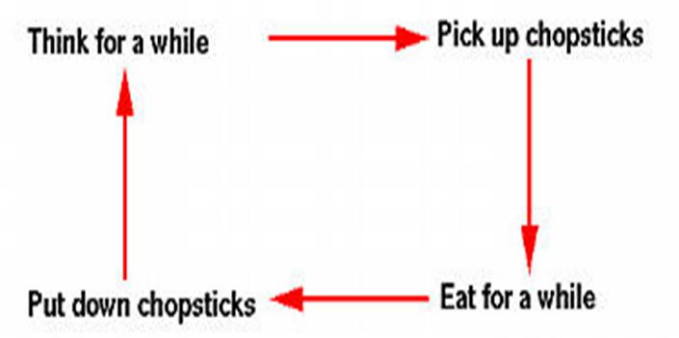
Although this solution guarantees that no two neighbors are eating at the same time, it could create a deadlock

Psuedocode:

1. Initialize that the philosophers[i]=thinking.
2. If philosopher[i] wants to eat.
3. pickup chopstick.
4. If he succeeds to pickup, he will eat

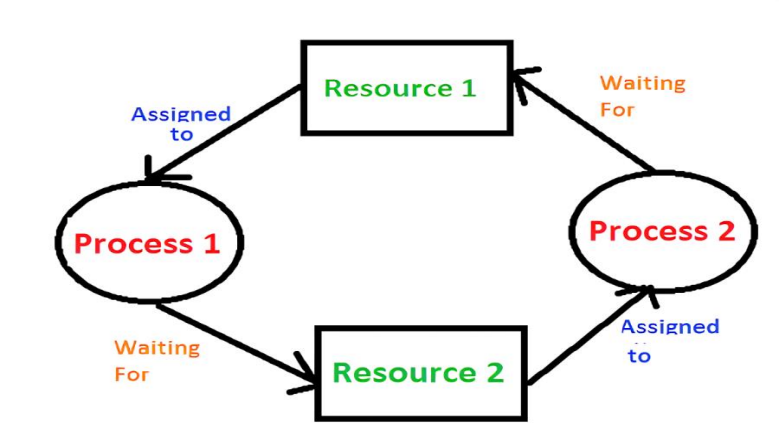
and when he complete he putdown chopstick .

5. than return to step 1.
6. make test[i+4]%5; , test[i+1]%5; .
7. If he not succeeds, he wait and signaled by others then return to step.



Deadlock:

Deadlock is a condition that is created in operation system when one process is holding some resources for making their execution but same resources are hold by another process, then this condition is known as “Deadlock“. Mostly, deadlock problem is arisen in multi processing system because in which multiple processes try to share particular mutually exclusive resource.



In philosopher problem:

If all five philosophers become hungry at the same time and each take his Left chopsticks. all the element chopstick will now be equal to 0 .When each philosopher tries to take his right chopstick, he will be delayed forever. Several possible solutions to the deadlock problem, one of them is to Allow a philosopher to pick up his chopsticks only if both chopsticks are available.

Examples of Deadlock:

In A real-world example:

would be traffic, which is going only in one direction. Here, a bridge is considered a resource.

In operating system:

Deadlock happens when two or more process need some resource to complete their execution that is held by the other process.

How did solve Deadlock:

There are different deadlock handling techniques like as Deadlock Prevention, Deadlock Avoidance and Deadlock Detection and Recovery.

1. **Deadlock Prevention :**
   Deadlock prevention means to block at least one of the four conditions required for deadlock to occur. If we are able to block any one of them then deadlock can be prevented.

The four conditions which need to be blocked are:

1. Mutual exclusion: only one process at a time can use a resource.
2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. No preemption: Resources cannot be preempted, a resource can be released only by the process holding it, after that process has completed its task.
4. Circular wait: there exists a set {P0 , P1 , …, Pn } of waiting processes such that P0 is waiting for a resource that is held by P1 , P1 is waiting for a resource that is held by P2 , …, Pn–1 is waiting for a resource that is held by Pn , and Pn is waiting for a resource that is held by P0.

2. Deadlock Avoidance :

In Deadlock avoidance we have to anticipate deadlock before it really occurs and ensure that the system does not go in unsafe state. It is possible to avoid deadlock if resources are allocated carefully. For deadlock avoidance we use Banker's and Safety algorithm for resource allocation purpose. In deadlock avoidance the maximum number of resources of each type that will be needed are stated at the beginning of the process.

3.Deadlock detection and recovery :

In this technique, CPU has ability to guess few criteria, if deadlock will be occurred in the entire system. Then, CPU will precede few recovery techniques to resolve the deadlock problem, and CPU frequently identify to all deadlock issues. CPU applies the Resource Allocation Graphs concept to detect the deadlock in the entire system.

In recovery scenario, CPU gets forcefully resources assigned to few processes, and those resources can also supply to another process but the process must be high level priority.

Explanation of solution for deadlock problem in our code:

- **The initial state of each philosopher is Think then he start hungery and to check if he can pickup left chopstick**

1. If yes he will also pickup the right one and eating then when he finish, he think and putdown the left and right chopstick.
2. If no he will put down it.

```
@Override
public void run(){
    String oldState = "state";
    think();
    while(true){

       if(!oldState.equals(state)){
          hungry();
          oldState = state;
       }

       Start();

        }
}
```

starvation:

Starvation is a condition where a process doesn't get the resources it needs for a long time because the resources are being allocated to other processes.

There are some common causes of starvation as follows:

- Starvation may occur if there aren't enough resources to provide to every process as needed.
- Starvation can occur if a process is never given the resources it needs for execution due to faulty resource allocation decisions.

Examples of starvation:

A common example of Starvation in operating systems is when a process called CPU seeks a certain resource but another process holds it until the first one runs out of its allocated time slice, thus leading to Starvation

For example :in a priority based system, the scheduler always picks the task with the highest priority therefore the execution of the low priority task may be delayed indefinitely ,if new higher priority tasks keep arriving at a rate that gives no chance to the lower priority task to execute.

How did solve starvation:

Some solutions that may be implemented in a system that helps to handle starvation are as follows:

The resource allocation priority scheme should contain concepts such as aging, in which the priority of a process increases the longer it waits. It prevents starvation.

An independent manager may be used for the allocation of resources. This resource manager distributes resources properly and tries to prevent starvation.

Random process selection for resource allocation or processor allocation should be avoided since it promotes starvation.

Explanation of solution for starvation problem in our code:

- **Each philosopher has a left chopstick so we have to check if the right one is exist:**
  -If it is exist the philosopher pickup it and start eating and when he finish, he start thinking and put down the right and left chopsticks.
  -if not he put down the left chopstick.

```
17    private void Start(){
18            state = "hungry";
19            if(left.pickUp()){
20                if(right.pickUp()){
21                    eat();
22
23                    think();
24                    right.putDown();
25                    left.putDown();
26                                }
27                else{
28                     left.putDown();
29                    }
30        }
31
32    }
```

Real world application-> ***Experiment of scientists problem:***

- Suppose we have 4 scientists (who share 3 activities of thinking, needing and working)with 2 microscopes and 2 PCs in the same room around the table and every scientist can’t work without 1 PC and 1 microscope.

Problem of this application

-The 4 scientists can't work together at the same time and we have to make solution guarantees that no two neighbours working at the same time.

Shared data:

- **Microscopes.**
- **PCs.**

Each scientist has specific data (microscope_id, PC_id and scientist_state)

```
9 usages
public class Scientist extends Thread {
    5 usages
    int scientist_id;
    2 usages
    int pc_id;
    2 usages
    int microscope_id;
    4 usages
    Tools pc;
    3 usages
    Tools microscope;
    6 usages
    String state;
4 usages
Scientist(Tools pc, Tools microscope, int scientist_id, int pc_id, int microscope_id) {
        this.scientist_id = scientist_id;
        this.pc = pc;
        this.microscope = microscope;
        this.microscope_id = microscope_id;
        this.pc_id = pc_id;
}
```

Each PC and microscope has specific data

Scientist.java | ExperimentOfScientistsProblem.java | Tools.java

```
1   package realworld;
2   import java.util.concurrent.locks.ReentrantLock;
3
    1 usages
4   public class Tools {
        2 usages
5       ReentrantLock fork = new ReentrantLock();
        1 usage
6       int num;
7
        2 usages
8       Tools(int num) { this.num = num; }
        2 usages
11      public boolean use() { return fork.tryLock(); }
14
        3 usages
15      public void end() { fork.unlock(); }
18  }
```

There are 3 states to each scientist:

1. Work.
2. Think.

```
1 usage
void work() {
    System.out.println("Scientist " + scientist_id + " is working");
    try {
        Thread.sleep((long) Math.round(Math.random() * 800));
    } catch (InterruptedException ex) {
        Logger.getLogger(Scientist.class.getName()).log(Level.SEVERE, s: null, ex);
    }
}

2 usages
void think() {
    System.out.println("scientist " + scientist_id + " is Thinking");
    state = "Thinking";
}
```

3. Need(PC – microscope)

```
void needPcAndMicro(){
    try {
        Thread.sleep((long) Math.round(Math.random() * 1000));
    } catch (InterruptedException ex) {
        Logger.getLogger(Scientist.class.getName()).log(Level.SEVERE, msg: null, ex);
    }
    System.out.println("scientist " + scientist_id + " needs PC " + pc_id + " and Microscope " + microscope_id );
    state = "needPC&Microscope";


}

}
```