Operating system-2 project

pseudocode:

class philosophers.

Incement philosophers, forks,hreads,rscan,number.

Here I created an object called obj to think about eating.

DiningPhilosophers obj=new DiningPhilosophers().

obj<-init().

obj<-StartThinkingEating().

Here is Oman if as a condition, because the entered number is greater than one, because if the number is one, it will not work, because there will be no problem in the first place.

And I made a (try-catch) to protect the code from wrong entries, and an error appears if a number equal to one is entered.

```
   try
 print "Enter the number of philosophers.".
 number= next number.

        if(number<2)
 print"Number must begreater than 1."
 return.

catch (Exception e)
```

```
print "Youmustenteraninteger."

philosophers=new Philosopher[number].
forks=new Fork[number].
threads=new Thread[number].

  Here we reveal the status of the adjacent fork that he is trying to connect, available or not.

        for(input i=0.i<number.i++)
philosophers[i]=new Philosopher(i+1).
forks[i]=new Fork(i+1).

        for(inti=0.i<number.i++)
 input i.
threads[i]=new Thread(new Runnable())

        try
philosophers[index].start(forks[index],forks[(index+1)%(number)]).
catch(InterruptedException e)
e.printStackTrace().

threads[i].start().

        **************************************************
class forks

        input forkId.
```

choose ture and false status.

Fork(input forkId)

this<-forkId = forkId.

this<-status = true.

This class contains two variables, forkId and status,

these two express the state of the fork, whether it is in use or not, and the fork number.

When the case is true, the fork is in place, or if it is false, the fork is in use.

when status = true.

input counter = 0.

input waitUntil = new Random().nextInt(10)+5.

while(!status)

Thread.sleep(new Random().nextInt(100)+50).

counter++.

if (counter > waitUntil){

return false.}

status = false.

return true.

In this spiral, we detect the state of the fork, whether it is in use or not.

If the condition is equal to false, this means the fork is in use. In this case, the market enters the philosopher into a random waiting state.

If it is true, this means that the fork is available and starts eating directly, and the counting action begins after The philosopher attempts to eat while the fork is not available.

When the fork's state turns true, he starts eating, and this is a random period of time to avoid the starvigen problem.

****

class philosophera

private input philosopherId.

private Fork left, right.

There are two variables in this class, pholosipherId and fork direction

input philosopherId

this.philosopherId = philosopherId.

This is a construction to determine the figure of the philosopher, and he began to think and eat

this<-left = left.

this<-right = right.

And here the directions are attached to the forks.

while(true){

if(new Random().nextBoolean())

```
            eat().
        else:
            think().
```

This spiral allows for an endless exchange of eating and thinking

```
    print"The philosopher: " + philosopherId + " is now thinking." .
    Thread.sleep(new Random().nextInt(1000)+100).
    print "The philosopher: " + philosopherId + " has stopped thinking." .
```

This function makes the philosopher think for a random period

```
    chosse the true or false rightPick = false.
    chosse the true or false leftPick = false.
```

In this function, the philosopher defines the thorn

```
    print "The philosopher: " + philosopherId + " is now hungry and wants to eat.".
    print "The philosopher: " + philosopherId + " is now picking up the fork: " + left.forkId .
    leftPick = left.pick(philosopherId).
    if (!leftPick){
        return.
    }
    print"The philosopher: " + philosopherId + " is now picking up the fork: " + right.forkId.
    rightPick = right.pick(philosopherId).
    if (!rightPick){
```

```
            left<-free().
            return.
        }
        print"The philosopher: " + philosopherId + " is now eating.".
        Thread.sleep(new Random().next number(1000) + 100).

        left<-free().
        right<-free().
        print"The philosopher: " + philosopherId + " has stopped eating and freed the forks.".
```

Here it will clarify if the choice of the left fork is correct and that the fork is not used,

i.e. the status of the fork is false, it will move to detect the status of the right fork,

but if the status of the left fork is true, any user will stop for a random period.

Examples of Deadlock:

• Suppose that all five philosophers become hungry at the same time, and each take her left chopstick.

• All the elements of the chopstick will now be equal to 0.

• When each philosopher tries to take her right chopstick, she will be delayed forever.

```
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);
```

How did solve deadlock:

• by putting a condition that if a fork is held by a philosopher for a long enough time and he is not getting access to the other fork then the philosopher will effectively time out and put down the fork that he currently has

This will make the fork available to the other philosopher

```
public synchronized boolean pick(int philosopherID) throws InterruptedException{
    int counter = 0;
    int waitUntil = new Random().nextInt(10)+5;
    while(!status){
        Thread.sleep(new Random().nextInt(100)+50);
        counter++;
        if (counter > waitUntil){
            return false;
        }
    }
    status = false;
    return true;
}
```

Examples of starvation:

Starvation is basically when a philosophers dies of starvation duo to them waiting too long to be able to eat.

Example:

Starvation when both philosophers 1 and 3 pickup both chopsticks at the same time and start eating , after sometimes , they drop their chopsticks and both philosophers 2 and 5 get to eat

After they satisfied , they drop their chopsticks and both philosophers 1 and 3 get to eat again

Both philosophers 1 and 3 and both philosophers 2 and 5 are taking turn to eat repeatedly .

Hence, philosopher 4 starves to death.

How did solve starvation:

• By giving priority , suppose you maintain a queue of philosophers ,when a philosopher is hungry , he gets put onto the tail of the queue . A philosopher may eat only if he is at the head of the queue and if the chopsticks are free.

Explanation for real world application:

- Boxing game that need to share limited gloves.

All philosophers represent players, and the chopsticks represent gloves that has to be shared between players

we have limited gloves and they have to be shared between players in a synchronized manner without violating any rules

how did apply the problem:

there may be gloves that can be used by only one player at a time, so if other players try to access the same gloves at the same time, then we will be having problems.