Documentation

deadlock is a situation in which more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process.

Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time.

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl • Need both to eat, then release both when done
- In the case of 5 philosophers
- Shared data
- Dish of rice (data set)
- Semaphore chopstick [5] initialized to 1

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

```
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 algorithm solve the deadlock:

```
public synchronized void occupy(philosopher philosopher) {
if (occupiedBy != null) {            if (philosopher !=
occupiedBy) {                this.reserve(philosopher);
          }
      } else {
          this.occupiedBy = philosopher;
      }
   } //who reserved the fork
```

Examples of starvation

```
monitor DiningPhilosophers { enum {
THINKING, HUNGRY, EATING)
state [5] ; //Default for all 5 is THINKING
condition self [5];  void pickup (int i) {
state[i] = HUNGRY;
test(i); //try to acquire the two forks  if
(state[i] != EATING)
self[i].wait; //block myself if the two forks isn't acquired
}
 void putdown (int i)
{
state[i] = THINKING; //philosopher has finished eating
test((i + 4) % 5); //see if left can now eat test((i + 1) %
5); //see if right can now eat
```

How did algorithm solve the starvation

```
Private synchronized void reserve(philosopher philosopher) {        this.reservedBy = philosopher;

      Try {

         Logger.printOut(philosopher.getName() + " has to wait fork. ");           this.wait();
```

```
} catch (InterruptedException ex) {            ex.printStackTrace();

}

This.reservedBy = null;        this.occupy(philosopher);
```

```
public synchronized void release(philosopher philosopher) {
this.occupiedBy = null;        if (this.reservedBy != null) {
this.notifyAll(); //wake up all thread
      }
```

Solution pseudocode

```
Y:=new philosopher
While (true){
Wait
//wait for a random interval of time
//check for left fork
If (occupiedby != null){
If(philosopher !=occupiedby){
Wait
Reservedby :=null
```

```
Occupy fork for philosopher
}
}
//check for right fork
If (occupiedby != null){
If(philosopher !=occupiedby){
Wait
Reservedby :=null
Occupy fork for philosopher
}
}
//philosopher is eating
Print philosopher is eating
//release left fork object
Occupiedby  :=  null
If(rservedby != null)
notifyAll
//release right fork object
Occupiedby  :=  null
If(rservedby != null)
notifyAll }
```

Boxing player Problem

Boxing player spend their lives alternating playing and waiting

Don’t interact with their neighbors, occasionally try to pick up 2 gloves (one at a time) to play

- Need both to play , then release both when done

In the case of 5 Boxing player Shared

data :

- Sandbag
- Semaphore gloves[5] initialized to 1