

Dining Philosophers Problem

Introduction

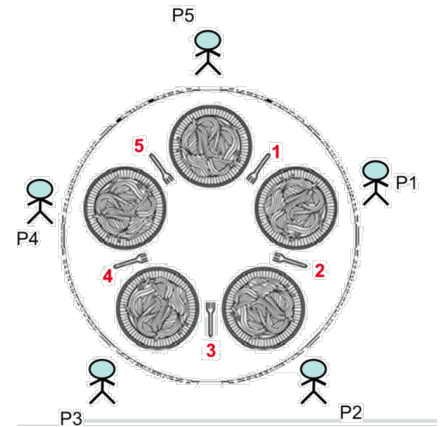
The **Dining Philosophers problem** is one of the classic problems used to **describe synchronization issues in a multi-threaded environment**

The Problem:

The problem was designed to illustrate the challenges of avoiding deadlock.

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;
- put the left fork down;
- put the right fork down;
- repeat from the beginning.



A Solution:

```
while(true) {  
    // Initially, thinking about life, universe, and everything  
    think();  
    // Take a break from thinking, hungry now  
    if (leftChopSticks.pickUp()) {  
        //able to acquire leftChopstick  
        if (rightChopSticks.pickUp()) {  
            //able to acquire rightChopstick  
            eat();  
            rightChopSticks.putDown();  
        }  
        leftChopSticks.putDown();  
    }  
    // Not hungry anymore. Back to thinking!  
}
```

pseudo code describes, each philosopher is initially thinking. After a certain amount of time, the philosopher gets hungry and wishes to eat.

At this point, he reaches for the forks on his either side and once he's got both of them, proceeds to eat. **Once the eating is done, the philosopher then puts the forks down, so that they're available for his neighbor.**

Output:

```
Philosopher{id=2} is thinking...
Philosopher{id=4} is thinking...
Philosopher{id=0} is thinking...
Philosopher{id=1} is thinking...
Philosopher{id=3} is thinking...
Philosopher{id=4} picked up LEFT Chopsticks{id=4}
Philosopher{id=3} picked up LEFT Chopsticks{id=3}
Philosopher{id=4} picked up RIGHT Chopsticks{id=0}
Philosopher{id=4} is eating..
Philosopher{id=3} put down LEFT Chopsticks{id=3}
Philosopher{id=3} is thinking...
Philosopher{id=0} is thinking...
Philosopher{id=0} is thinking...
Philosopher{id=4} put down RIGHT Chopsticks{id=0}
Philosopher{id=4} put down LEFT Chopsticks{id=4}
Philosopher{id=4} is thinking...
Philosopher{id=1} picked up LEFT Chopsticks{id=1}
Philosopher{id=1} picked up RIGHT Chopsticks{id=2}
Philosopher{id=1} is eating..
```

All the *Philosophers* initially start off thinking, and we see that *Philosopher 4* proceeds to pick up the left and right fork, then eats and proceeds to place both of them down.

the Deadlock problem:

What is Deadlock?

A **deadlock** is a situation in which more than one process is blocked because it is holding a resource and requires some resource that is acquired by some other process. Therefore, none of the processes gets executed. The four necessary conditions for a deadlock situation to occur are mutual exclusion, hold and wait, no preemption and circular set.

Necessary conditions for Deadlocks

1. Mutual Exclusion

A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

2. Hold and Wait

A process waits for some resources while holding another resource at the same time.

3. No preemption

The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

4. Circular Wait

All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

Solution for Deadlock:

There are three ways to handle deadlock:

1. **Deadlock prevention:** The possibility of deadlock is excluded before making requests, by eliminating one of the necessary conditions for deadlock.

Example: Only allowing traffic from one direction, will exclude the possibility of blocking the road.

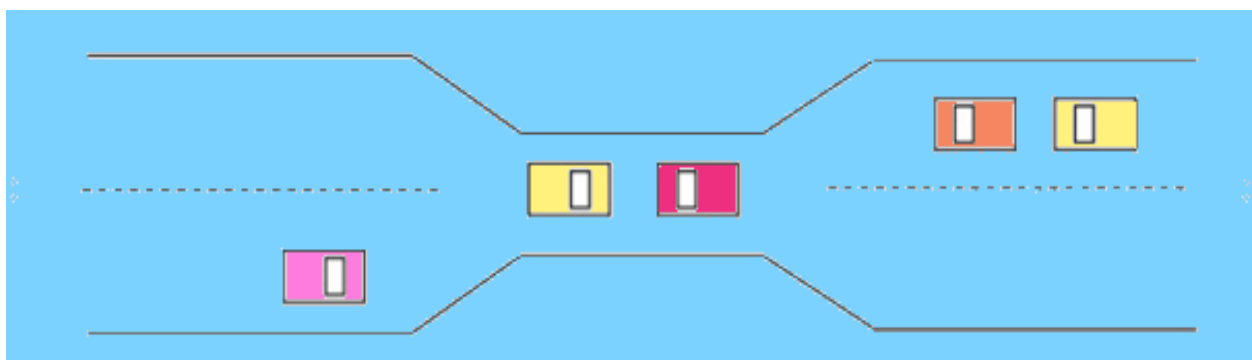
2. **Deadlock avoidance:** Operating system runs an algorithm on requests to check for a safe state. Any request that may result in a deadlock is not granted.

Example: Checking each car and not allowing any car that can block the road. If there is already traffic on road, then a car coming from the opposite direction can cause blockage.

3. **Deadlock detection & recovery:** OS detects deadlock by regularly checking the system state and recovers to a safe state using recovery techniques. **Example:** Unblocking the road by backing cars from one side. Deadlock prevention and deadlock avoidance are carried out before deadlock occurs.

Example of Deadlock

- A real-world example would be traffic, which is going only in one direction.
- Here, a bridge is considered a resource.
- So, when Deadlock happens, it can be easily resolved if one car backs up (Preempt resources and rollback).
- Several cars may have to be backed up if a deadlock situation occurs.
- So starvation is possible.



Second example for Deadlock

No, you hang up first” problem. So imagine a couple having a romantic phone conversation and when it’s time to hang up neither of them wants to hang up first and they keep saying “No, *you hang up first!*”.

There is nothing stopping them from hanging up but they both are waiting the other person ending the conversation.

Third example for Deadlock:

jack and Jill happen to want to make a sandwich at the same time. Both need a slice of bread, so they both goes to get the loaf of bread and a knife.

Jack gets the knife first, while Jill gets the loaf of bread first. Now Jack tries to find the loaf of bread and Jill tries to find the knife, but both find that what they need to finish the task is already in use. If they both decide to wait until what they need is no longer in use, they will wait for each other forever.

Difference Between Deadlock and starvation

| Sr. | Deadlock | Starvation |
|-----|--|---|
| 1 | Deadlock is a situation where no process got blocked and no process proceeds | Starvation is a situation where the low priority process got blocked and the high priority processes proceed. |

| | | |
|---|---|--|
| 2 | Deadlock is an infinite waiting. | Starvation is a long waiting but not infinite. |
| 3 | Every Deadlock is always a starvation. | Every starvation need not be deadlock. |
| 4 | The requested resource is blocked by the other process. | The requested resource is continuously be used by the higher priority processes. |
| 5 | Deadlock happens when Mutual exclusion, hold and wait, No preemption and circular wait occurs simultaneously. | It occurs due to the uncontrolled priority and resource |

Example of deadlock in our code:

The locking and unlocking is that after we have done using the shared resource, we may forget to unlock the lock object. This would leave the lock object locked forever. This is the source for bugs and deadlocks. But even if we remember to unlock the lock, we still have the problem of a method throwing an exception inside our critical section. Please see the below code.

```
public boolean pickUp(Philosopher philosopher, States state) throws InterruptedException {
```

```
    lock.lock
```

```
    System.out.println(philosopher + " picked up " + state.toString() + " " + this);
```

```
    return true;
```

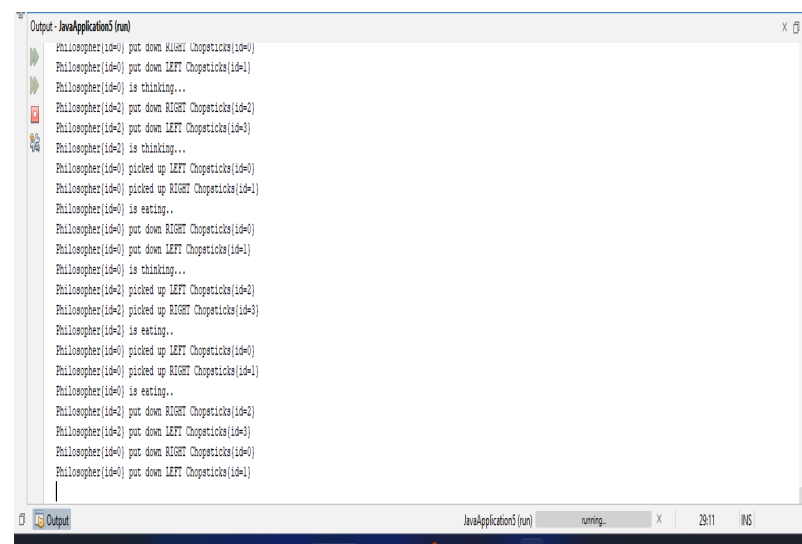
```
}
```

```
public void putDown(Philosopher philosopher, States state) {
```

```
    // lock.unlock(); // and we forget to put lock.unlock
```

```
    System.out.println(philosopher + " put down " + state.toString() + " " + this);
```

```
}
```



```
Output - JavaApplication5 (run)
Philosopher(id=0) put down RIGHT Chopsticks(id=0)
Philosopher(id=0) put down LEFT Chopsticks(id=1)
Philosopher(id=0) is thinking...
Philosopher(id=2) put down RIGHT Chopsticks(id=2)
Philosopher(id=2) put down LEFT Chopsticks(id=3)
Philosopher(id=2) is thinking...
Philosopher(id=0) picked up LEFT Chopsticks(id=0)
Philosopher(id=0) picked up RIGHT Chopsticks(id=1)
Philosopher(id=0) is eating..
Philosopher(id=0) put down RIGHT Chopsticks(id=0)
Philosopher(id=0) put down LEFT Chopsticks(id=1)
Philosopher(id=0) is thinking...
Philosopher(id=2) picked up LEFT Chopsticks(id=2)
Philosopher(id=2) picked up RIGHT Chopsticks(id=3)
Philosopher(id=2) is eating..
Philosopher(id=0) picked up LEFT Chopsticks(id=0)
Philosopher(id=0) picked up RIGHT Chopsticks(id=1)
Philosopher(id=0) is eating..
Philosopher(id=2) put down RIGHT Chopsticks(id=2)
Philosopher(id=2) put down LEFT Chopsticks(id=3)
Philosopher(id=0) put down RIGHT Chopsticks(id=0)
Philosopher(id=0) put down LEFT Chopsticks(id=1)
```

```
public boolean pickUp(Philosopher philosopher, States state) throws InterruptedException {
    // here we deal with deadlock

    if (lock.tryLock(10, TimeUnit.MILLISECONDS)) {
        lock.lock();
        System.out.println(philosopher + " picked up " + state.toString() + " " + this);
        return true;
    }

    return false;
}

public void putDown(Philosopher philosopher, States state) {
    //lock.unlock();
    System.out.println(philosopher + " put down " + state.toString() + " " + this);
}
```

lock.unlock

```
public boolean pickUp(Philosopher philosopher, States state) throws InterruptedException {
    if (lock.tryLock(10, TimeUnit.MILLISECONDS)) {
        System.out.println(philosopher + " picked up " + state.toString() + " " + this);
        return true;
    }

    return false;
}

public void putDown(Philosopher philosopher, States state) {
    lock.unlock();
    System.out.println(philosopher + " put down " + state.toString() + " " + this);
}
```

the Starvation problem:

Starvation problem:

Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. In starvation resources are continuously utilized by high priority processes. Problem of starvation can be resolved using Aging. In Aging priority of long waiting processes is gradually increased.

Example 1:

- imagine that we have 4 processes p1,p2,p3 and p4 having the priority 20 , 1 , 5 and 2 respectively (lesser number is highest priority) and duration time is 10 , 5 , 2 and 40 respectively.

| Process | Burst time | priority |
|---------|------------|----------|
| P1 | 10 | 20 |
| P2 | 5 | 1 |
| P3 | 2 | 5 |
| P4 | 40 | 2 |

- main scenario is processor has been allocated to this process p2 from second zero to 4 then process p4 is allocated from second 5 to 44 then process p3 from 45 to 46.

0 4 44 46

| | | | |
|----|----|----|--|
| P2 | P4 | P3 | |
|----|----|----|--|

then before p1 is allocated another process with highest priority is in queue p5,p6,p6... having priorities 3,4,7 respectively and another processes coming with priority higher than p1 (priorities from 0 to 19) , which is make p1 wouldn't get CPU then p1 will wait indefinitely although p1 is ready to run in ready queue but it is waiting because higher priority processes is coming and CPU is biased of this priority scheduling so CPU will allocated to processes having higher priority and this is known as **starvation**.

Example 2:

A real-world example is the “**Airport with a single check-in counter**” problem.

Imagine an airport with a single check-in counter and two queues, one for business class and one for economy class. As you may know the business class has the priority over the economy class.

However, there is a business convention period, and the business queue is always full. The economy queue doesn't move at all and the people in that queue will miss the plane as they cannot access the check-in counter.

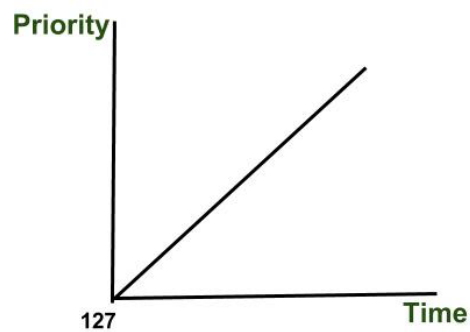
Example 3:

Let say that you work in a big company. Each day there is a rush hour in lunch time - everyone wants to get in the food line first. Your office is at the top floor and only way to get to the lobby is to use a lift. So, you call the lift and wait... and wait. Your waiting time could be infinite because everyone in bottom floors is loading the lift, so it never reaches the top! And when it finally does, you decided to go back to your office and eat some leftovers from yesterday.

Solution of starvation:

We could solve starvation problem with aging, it is gradually increase priority of processes priority of those processes which are waiting in the system for along amount of time, or we can say that aging is the method to ensure the processes with lower priority were eventually complete their execution but how??

Aging increasing priority by decreasing the number of processes which have lower priority, for example, if priority range from 127(low) to 0(high), we could increase the priority of a waiting process by 1 Every 15 minutes. Eventually, even a process with an initial priority of 127 would take no more than 32 hours for the priority 127 process to age to a priority-0 process.

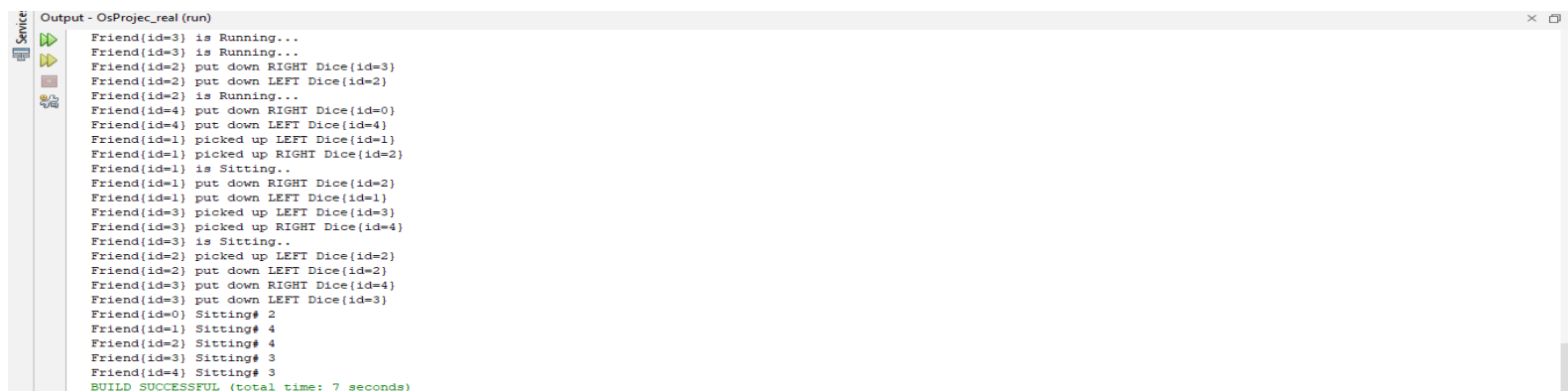


In previous example process p1 with priority 20 will become 19 after three unit of time and another three unit of time will become 18 and so on.

Example of Starvation in our code:

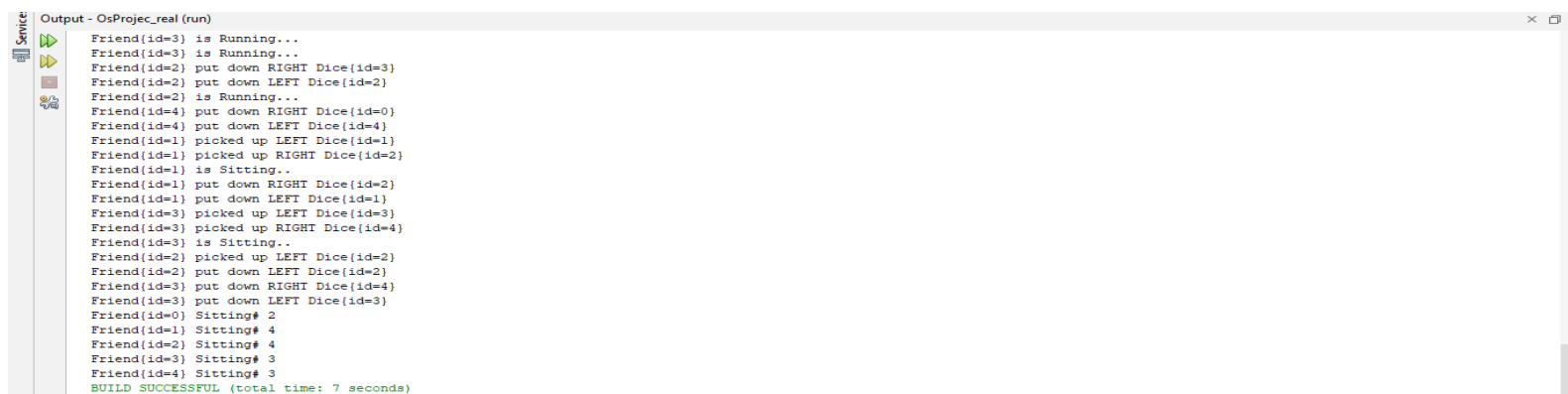
We use `reentrantlock` so if the Boolean inside the `reentrantlock` is false there may be a starvation

ReentrantLock(true)



```
Output - OsProjec_real (run)
Friend(id=3) is Running...
Friend(id=3) is Running...
Friend(id=2) put down RIGHT Dice(id=3)
Friend(id=2) put down LEFT Dice(id=2)
Friend(id=2) is Running...
Friend(id=4) put down RIGHT Dice(id=0)
Friend(id=4) put down LEFT Dice(id=4)
Friend(id=1) picked up LEFT Dice(id=1)
Friend(id=1) picked up RIGHT Dice(id=2)
Friend(id=1) is Sitting..
Friend(id=1) put down RIGHT Dice(id=2)
Friend(id=1) put down LEFT Dice(id=1)
Friend(id=3) picked up LEFT Dice(id=3)
Friend(id=3) picked up RIGHT Dice(id=4)
Friend(id=3) is Sitting..
Friend(id=2) picked up LEFT Dice(id=2)
Friend(id=2) put down LEFT Dice(id=2)
Friend(id=3) put down RIGHT Dice(id=4)
Friend(id=3) put down LEFT Dice(id=3)
Friend(id=0) Sitting# 2
Friend(id=1) Sitting# 4
Friend(id=2) Sitting# 4
Friend(id=3) Sitting# 3
Friend(id=4) Sitting# 3
BUILD SUCCESSFUL (total time: 7 seconds)
```

ReentrantLock(false)



```
Output - OsProjec_real (run)
Friend(id=3) is Running...
Friend(id=3) is Running...
Friend(id=2) put down RIGHT Dice(id=3)
Friend(id=2) put down LEFT Dice(id=2)
Friend(id=2) is Running...
Friend(id=4) put down RIGHT Dice(id=0)
Friend(id=4) put down LEFT Dice(id=4)
Friend(id=1) picked up LEFT Dice(id=1)
Friend(id=1) picked up RIGHT Dice(id=2)
Friend(id=1) is Sitting..
Friend(id=1) put down RIGHT Dice(id=2)
Friend(id=1) put down LEFT Dice(id=1)
Friend(id=3) picked up LEFT Dice(id=3)
Friend(id=3) picked up RIGHT Dice(id=4)
Friend(id=3) is Sitting..
Friend(id=2) picked up LEFT Dice(id=2)
Friend(id=2) put down LEFT Dice(id=2)
Friend(id=3) put down RIGHT Dice(id=4)
Friend(id=3) put down LEFT Dice(id=3)
Friend(id=0) Sitting# 2
Friend(id=1) Sitting# 4
Friend(id=2) Sitting# 4
Friend(id=3) Sitting# 3
Friend(id=4) Sitting# 3
BUILD SUCCESSFUL (total time: 7 seconds)
```

Real world Example (funny game!):

The Funny friends states that there are 5 Friends sharing a circular table and they sit and run alternatively. There is 5 dice. A philosopher needs both their right and left Dice to sit. A Friend may only sit if there are both adjacent Dice available .Otherwise a friend puts down their dice and begin running again.

Solution of funny game Problem:

A solution of the funny game Problem is to use a ReentrantLock to represent a chopstick. A chopstick can be picked up by executing a lock operation on the ReentrantLock and released by executing a unlock.

```
ReentrantLock chopstick [5];
```

Initially the elements of the dice are initialized to 1 as the chopsticks are on the table and not picked up by a Friend.

first lock operation is performed on dice [i] and dice [(i+1) % 5]. This means that the Friend i has picked up the dice on his sides. Then the sitting function is performed.

After that, unlock operation is performed on dice [i] and dice [(i+1) % 5]. This means that the Friend i has sitting and put down the dice on his sides. Then the Friend goes back to running.

Thank you Professor Ahmed Swar