**CS5541 - Computer Systems**

**Assignment Name: Dining Philosophers**

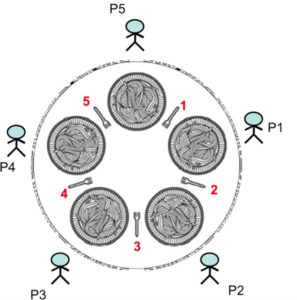
**Spring 2019**

Sampath Talluri

April 28, 2019

**Implementation:**

* I created this program using threads.
* There is a fork/spoon between each philosopher.
* A philosopher must pick it’s two nearest fork/spoon in order to eat.
* He can only pick one fork/spoon 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 in to critical section.
  + **EATING –** Got both the forks/spoons and entered the section.



**A solution:**

* 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();

}

}

}

* + The above code tell us, initially each philosopher is in thinking state.
  + After a certain time, each philosopher gets hungry and think to eat.
  + When philosopher gets his both fork/spoon, proceeds to eat.
  + Once the eating is completed, the philosopher puts the fork/spoon down.
  + So that other philosopher can eat.

1. **What method did you use to ensure that one version of your program was susceptible to starvation but not deadlock?**

**Ans:**

* + 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"))

{

//Thread.currentThread().notify();

return true;

}

1. **What method did you use to ensure that one version of your program was susceptible to deadlock but not starvation?**

**Ans:**

* + 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);

}

1. **What method did you use to ensure that one version of your program was not susceptible to starvation OR deadlock?**

**Ans:**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

{ //philosopher eating method

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

{ //after eating philosopher thinking

state.status[id]="thinking";

System.out.println(getName()+" thinking ");

sleep(((int) (Math.random() \* 500)));

}

public void hungry()throws InterruptedException

{ //philosopher hungry method

state.status[id]="hungry";

System.out.println(getName()+" hungry ");

sleep(((int) (Math.random() \* 500)));

}