ISTANBUL BILGI UNIVERSITY

OPERATING SYSTEM

DINING PHILOSOPHERS

Name Surname:
Ayberk SUNAL
113200034

ISTANBUL BILGI UNIVERSITY

Dining Philosophers Problem Report

May 25, 2017

1 WHAT IS THE DINING PHILOSOPHERS PROBLEM?

The dining philosophers problem is a sample of a concurruncey problems. The problem orihinates with Edsger Dijikstra.

The dining philosophers problem is: how do you make sure every philosopher gets to eat? In the problem, there are 5 different philosophers who are seated around a circular table. Every philosophers behaves is in this way: Either he wants to think, or he wants to eat. There is a chopstick places between each pair of philosophers which both can use.

A philosopher have to use two of them these chopsticks to eat spaghetti. A philosopher can only pick up one chopstick at one time. When a philosopher finishes eating, they replaced their chopsticks to the table and they return thinking again. The problem is chopsticks are not enough for all philosophers to eat at the same time but each of them have to eat for a time.

- When a philosopher thinks, he does not interact with his others.
- From time to time, a philosopher gets hungry and tries to pick up the two forks that are closest to him

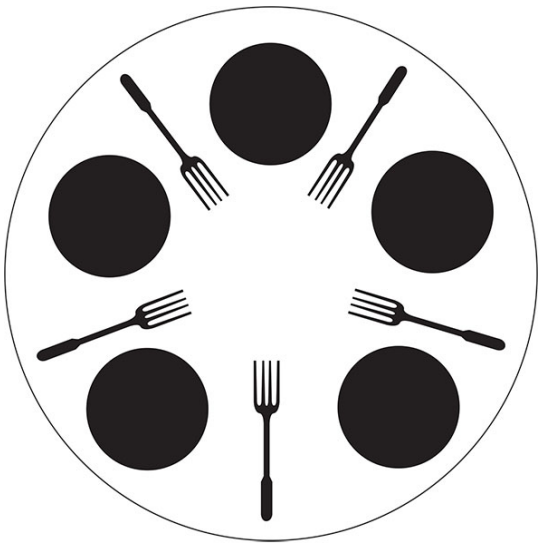

Figure 1.1: Dining Philosophers

- A philosopher may pick up only one chopstick at a time. Obviously, he cannot pick up a chopstick that is already in the hand of a neighbor.

- When he is finished eating, he puts down both of his chopsticks and starts thinking again.

2 WHICH ARE THE MAIN PROBLEMS OF PHILOSOPHERS?

The dining philosophers problem is a “classical” synchronization problem.So, We can explain the problem with these steps;

Deadlock:

If each philosopher has acquired her left chopstick the threads are mutually waiting for each other. Generally causes starvation. Sometimes, when a philosopher want to eat his meal, he may take one of the chopsticks and may wait the other chopstick to eat. But an other (nearby) philosopher can be in the same situation.And then they waited for someone to give up their chopstick so they waited forever and can not eat the meal. This situation is named as 'DEADLOCK'. Potential for deadlock exists independent of thinking and eating times.

Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)

1. Every philosopher get a chopstick.

2. Anyone does not have two of them and waiting the an other chopstick to eat.

3. This will never end and they will never give up!

Deadlock avoidance,
Deadlock in dining philosophers can be avoided;
if one philosopher picks up sticks in reverse order (right before left).
if there is always one philosopher who thinks.
if a blocked philosopher puts back the chopsticks and waits a random interval.

Livelock:

It is policy that makes them all do something endlessly without ever eating. Think that, We are setting a time limit. If the philosopher have been wairing a chopstick for 10 min. , he have to give up. Before dropping choptick, he has to try again. By then, any other philosopher philosopher will be done and the firts philosopher will be able to use this chopstick. In this case, there will NO deadlock. However, if all philosophers do these actions at the same time, we will have livelock.

1. You all picked up your fork at the exact same time.
2. After 10 minutes, you all put down your forks.
3. After 10 minutes , you just picked up your forks again!
4. This will go on forever!

3 WHICH ARE THE SOLUTIONS?

1. **Resource hierarchy solution**

   The resource hierarchy basically consists of ordering the forks by number. They will then always be requested by order and released in the reverse order.

   This way assigns a partial order to the chopsticks , and establishes the convention that all resources will be requested in this order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the chopsticks will be numbered 1 to 5 and each philosopher will always get the lower chopstick firstly, and then the higher chopstick, from among the two chopsticks they plan to use. The order in which each philosopher drop the chopsticks does not matter. In this case, if four of the five philosophers simultaneously get their lower chopstick, only the highest chopstick will remain on the table, so the fifth philosopher will not be able to get any chopstick. Moreover, only one philosopher will have access to that highest chopstick, so they will be able to eat using two chopsticks.

2. **Chandy/Misra solution**

   We have a solution wherein you describe the chopsticks as either being dirty or clean and based on this categorization the philosophers requests the chopsticks from one another. This request equals communication and so the restraint inherent to the problem has been violated.

   - For every pair of philosophers linked for a resource, create a chopstick and give it to the philosopher with the lower ID. Each chopstick can either be clean or dirty. Initially, all chopsticks are dirty.
   - If a philosopher wants to use a set of chopstick to eat, said philosopher must obtain the chopsticks from their contending neighbors. For all chopsticks the philosopher do'nt have, they send a request message.
   - They keep the chopstick if it is clean, when a philosopher with a chopstick receives a request message, but give it up when it is dirty. They clean the chopstick before doing so, If the philosopher sends the chopstick over.
   - All their chopsticks are gonna be dirty when a philosopher is done eating. He philosopher that has just finished eating will clean the chopstick and sends it, If another philosopher had previously requested one of the chopsticks.

This solution also allows for a large degree of concurrency, and will solve an arbitrarily large problem.

3. **Waiter solution (Arbitrator solution)**

   Actually philosophers do not have a chance to talk with each other. One solution is to have them talk to a waiter who keeps an overview of the table, he can then make decide who gets chopsticks and when. The waiter can be implemented as a mutex. The waiter gives permission to only one philosopher at a time until the philosopher has picked up both of their chopsticks.

4 DOES YOUR PROGRAM SOLVE THE PROBLEM EFFICIENTLY?

I have used the resource hierarchy solution way to solve the problem. So, my code solves the problem efficiently. It controls the chopsticks and determines a way for philosophers to eat.

Philosophers are numerated from 0 to 4 and they share a circular table. In this table philosophers who are even numbered get the right chopstick firstly and then get the left chopstick. If the philosopher number is odd numbered, he gets the left chopstick firstly and then get the right chopstick. To run this methodology, I get help from the pthears(mutex). I created 5 threads for chopsticks and philosophers and I controlled the turn with these threads with locking and opening.

IF philosopher number even
RIGHT => LEFT

ELSE
LEFT = >RIGHT|

1. Allow only 4 philosophers to sit simultaneously
2. Asymmetric solution
   a) Odd philosopher picks left fork followed by right
   b) Even philosopher does vice versa
3. Pass a token
4. Allow philosopher to pick fork only if both available

5 EXAMPLE OUTPUT

The 3.Philosopher is THINKING for 5 second.
The 0.Philosopher is THINKING for 3 second.

The 2.Philosopher is THINKING for 3 second.
The 4.Philosopher is THINKING for 4 second.
The 1.Philosopher is THINKING for 4 second.
The 0.Philosopher is WAITING to grab the chopstick 0
The 2.Philosopher is WAITING to grab the chopstick 2
The 0.Philosopher GRABBED chopstick 0
The 2.Philosopher GRABBED chopstick 2
The 0.Philosopher is WAITING to pick up chopstick 1
The 2.Philosopher is WAITING to pick up chopstick 3
The 0.Philosopher GRABBED chopstick 1
The 2.Philosopher GRABBED chopstick 3
The 0.Philosopher is EATING for 2 second.
The 2.Philosopher is EATING for 2 second.
The 4.Philosopher is WAITING to grab the chopstick 4
The 1.Philosopher is WAITING to grab the chopstick 2
The 4.Philosopher GRABBED chopstick 4
The 4.Philosopher is WAITING to pick up chopstick 0
The 3.Philosopher is WAITING to grab the chopstick 4
The 2.Philosopher GAVE BACK the chopstick
The 0.Philosopher GAVE BACK the chopstick
The 0.Philosopher is THINKING for 4 second.
The 2.Philosopher is THINKING for 5 second.
The 4.Philosopher GRABBED chopstick 0
The 1.Philosopher GRABBED the chopstick 2
The 4.Philosopher is EATING for 3 second.
The 1.Philosopher is WAITING to grab the chopstick 1
The 1.Philosopher GRABBED the chopstick 1
The 1.Philosopher is EATING for 2 second.
The 1.Philosopher GAVE BACK the chopstick
The 1.Philosopher is THINKING for 3 second.
The 4.Philosopher GAVE BACK the chopstick
The 4.Philosopher is THINKING for 2 second.
The 3.Philosopher GRABBED the chopstick 4
The 3.Philosopher is WAITING to grab the chopstick 3
The 3.Philosopher GRABBED the chopstick 3
The 3.Philosopher is EATING for 4 second.
The 0.Philosopher is WAITING to grab the chopstick 0
The 0.Philosopher GRABBED chopstick 0
The 0.Philosopher is WAITING to pick up chopstick 1
The 0.Philosopher GRABBED chopstick 1

6 C Based Solution Code

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>


pthread_mutex_t chopsticks[5];
pthread_t philosophers[5];

void *philosopher(void *);
void think(int);
void grabChopstik(int);
void eatTheDinner(int);
void dropChopstick(int);
void even(int);
void odd(int );



void *philosopher(void *philosopherNumber) {
    while (1) {
        think(philosopherNumber);
        grabChopstik(philosopherNumber);
        eatTheDinner(philosopherNumber);
        dropChopstick(philosopherNumber);
    }
}

void think(int numberOfPhilosopher){
    //can create a waiting time from 2 to 6
    int waitingTime =  2 + rand()%4;
    printf("The %d.Philosopher is THINKING for %d seccond.\n",numberOfPhilosopher , waitingTime );
    sleep(waitingTime);

}

void grabChopstik(int numberOfPhilosopher ){
    //number of the left stick
    int leftChopstick = (numberOfPhilosopher + 1) % 5;
    //number of the right stick
    int rightChopstick = numberOfPhilosopher;

    //to control wheter even or odd
    if(numberOfPhilosopher % 2 == 0){
        //if even philoshoper firt right stick and then left
        even(numberOfPhilosopher);

    }
    else if(numberOfPhilosopher % 2 == 1){
        //if odd philoshoper firt left stick and then right
        odd(numberOfPhilosopher);
```

Line 108, Column 13

```c
}

void even(int numberOfPhilosopher) {

    int leftChopstick = (numberOfPhilosopher + 1) % 5;
    //number of the right stick
    int rightChopstick = numberOfPhilosopher;

    printf("The %d.Philosopher is WAITING to grab the chopstick %d\n", numberOfPhilosopher, rightChopstick);
    pthread_mutex_lock(&chopsticks[rightChopstick]);
    printf("The %d.Philosopher GRABBED chopstick %d\n", numberOfPhilosopher, rightChopstick);
    printf("The %d.Philosopher is WAITING to pick up chopstick %d\n", numberOfPhilosopher, leftChopstick);
    pthread_mutex_lock(&chopsticks[leftChopstick]);
    printf("The %d.Philosopher GRABBED chopstick %d\n", numberOfPhilosopher, leftChopstick);


}

void odd(int numberOfPhilosopher){
    int leftChopstick = (numberOfPhilosopher + 1) % 5;
    //number of the right stick
    int rightChopstick = numberOfPhilosopher;

    printf("The %d.Philosopher is WAITING to grab the chopstick %d\n", numberOfPhilosopher, leftChopstick);
    pthread_mutex_lock(&chopsticks[leftChopstick]);
    printf("The %d.Philosopher GRABBED the chopstick %d\n", numberOfPhilosopher, leftChopstick);
    printf("The %d.Philosopher is WAITING to grab the chopstick %d\n", numberOfPhilosopher, rightChopstick);
    pthread_mutex_lock(&chopsticks[rightChopstick]);
    printf("The %d.Philosopher GRABBED the chopstick %d\n", numberOfPhilosopher, rightChopstick);

}
void eatTheDinner(int numberOfPhilosopher){
    //can create a waiting time from 2 to 6
    int waitingTime = 2 + rand()%4;
    printf("The %d.Philosopher is EATING for %d second.\n",numberOfPhilosopher , waitingTime );
    //to wait a few second
    sleep(waitingTime);
}

void dropChopstick(int numberOfPhilosopher) {
    //leaving the fork after eating the fork
    printf("The %d.Philosopher GAVE BACK the chopstick\n", numberOfPhilosopher);
    //number of the left stick
    int leftChopstick = (numberOfPhilosopher + 1) % 5;
    //number of the right stick
    int  rightChopstick = numberOfPhilosopher;

    //to go out from the critical section
    pthread_mutex_unlock(&chopsticks[leftChopstick]);
    pthread_mutex_unlock(&chopsticks[rightChopstick]);
}
```

```
106
107
108 int main() {
109     int j;
110     for (j = 0; j < 5; ++j) {
111         /*This part initialize the mutex for each chopstick*/
112         /*birinci parametresi mutex nesnesinin adresi, ikinci parametresi mutex
113         ozelliklerinin degerlerine iliskin yapinin adresidir*/
114         pthread_mutex_init(&chopsticks[j], NULL);
115     }
116
117
118     for (j = 0; j < 5; ++j) {
119         /*   create  fonksiyonu  ile  yeni  bir  thread  olusturup  bu  thread  calistirilabilir hale getirilir.
120          Ilk parametre: pthread_t tip adresidir.
121          Ikinci parametre: yeni thread in icermesini istedigimiz belli nitelikleri icerebilir. oncelik vb.
122          ucuncu parametre: bir fonksiyon isaretcisidir. Her thread bir fonksiyon baslatir. Bu parametre (fonksiyon adresi) ile kernel, thread in hangi fonksiyo
123          dorduncu parametre: void tipinde bir isaretcidir. Bu pointer ile fonksiyon birden fazla parametre kabul edebilir.*/
124         pthread_create(&philosophers[j], NULL, philosopher, (void *)(j));
125     }
126
127     /*Threadler  arasi  senkronizasyon  için pthread_join(..)  fonksiyonu  kullanilir
128     parametre  olarak  ilgili  threadin  ID'si  verilir*/
129     for (j = 0; j < 5; ++j) {
130         pthread_join(philosophers[j], NULL);
131     }
132     return 0;
133 }
134
135
136
```