THE DINING PHILOSOPHERS PROBLEM

Here is where your presentation begins

TABLE OF CONTENTS

01

OVERVIEW

Problem Statement

INTRODUCTION

The **Dining Philosophers Problem** is a classic resource-sharing synchronization problem. It is particularly used for situations, where multiple resources need to be allocated.
There are five philosophers sitting around a circular dining table. The table has a bowl of spaghetti and five chopsticks.
At any given time, a philosopher will either think or eat. For eating, he uses two chopsticks- one from his left and another from his right. When a philosopher thinks, he keeps down both the chopsticks at their place.

Pseudo code for Philosopher i

Unfortunately, the previous “solution” can result in deadlock – each philosopher grabs its right chopstick first

• causes each semaphore’s value to decrement to 0 – each philosopher then tries to grab its left chopstick

• each semaphore’s value is already 0, so each process will block on the left chopstick’s semaphore – These processes will never be able to resume by themselves - we have deadlock!

- while(1) {
- // obtain the two chopsticks to my immediate right
- and left P(chopstick[i]);
- P(chopstick[(i+1)%N];
- // eat //
- release both chopsticks V(chopstick[(i+1)%N];
- V(chopstick[i]);
- }

“how the above code is giving a solution to the dining philosopher problem”

Let value of i = 0(initial value), Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0, after that it execute **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C1 chopstick**(since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0
Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it will try to hold **C1 chopstick** but will not be able to do that, since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C3 chopstick**(since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

Pseudo code for Philosopher

```
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 eatingtes
t((i + 4) % 5);
//see if left can now eattes
t((i + 1) % 5);
//see if right can now eat}
```

“how the above code is giving a solution to the dining philosopher problem”

- We now illustrate monitor concepts by presenting a deadlock-free solution to the dining- philosophers problem.
- This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum { thinking, hungry, eating } state [5];
```

- Philosopher i can set the variable **state [i] = eating** only if his two neighbors are not eating: **(state [(i+4) % 5] != eating)** and **(state [(i+1)% 5] != eating)**.
- We also need to declare `condition self [5];` where philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.

02

Deadlock

PROBLEM

From the above solution of the dining philosopher problem, we have proved that no two neighboring philosophers can eat at the same point in time. The drawback of the above solution is that this solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.

SOLUTION

A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.

Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks

03

starvation

PROBLEM

Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a ***starvation***. Note that it is not a deadlock because there is no circular waiting, and every one has a chance to eat!

SOLUTION

you must guarantee that no philosopher may starve. For example, suppose you maintain a queue of philosophers. When a philosopher is hungry, he/she gets put onto the tail of the queue. A philosopher may eat only if he/she is at the head of the queue, and if the chopsticks are free.

When a philosopher calls **pickup()**, if the queue is empty, the chopsticks are checked, and if they are in use, the philosopher is put on the queue. If they are not in use, the philosopher is allowed to eat, and **pickup()** returns . The heart of the routine is a function that tests the head of the queue to determine if it can run (based on its neighbors' status) and signals the head if it can.

04

Real world application

Account Transaction

I find the problem of executing a transaction between two accounts very similar to the dining philosophers problem. To execute the transaction the thread must lock both accounts to ensure the correct value is debited from one account (first assuring there are available funds) and crediting to another.

The topology is not exactly the round table, but is very close. Imagine 5 accounts at the table. In this analogy, the accounts are the forks. two accounts can participate in a transaction. Transactions == philosophers. So in this example the transactions (philosopher) can not only sit at the edge of the table between two accounts (forks), but also on a line cutting across the table, connecting any two accounts (forks).

THANKS!