REPORT ON DINING PHILOSOPHER PROBLEM

BY Rohit Negi

Btech CSE 2nd year

130101144

**INTRODUCTION**

Dining philosopher is classical synchronization problem. It is invented by E. W. Dijkstra. Consider five philosophers sitting in a dining room and spend their most of their time thinking and whenever they feel hungry they eat. A philosopher needs 2 chopsticks to eat and when he does not get 2 chopsticks then he has to wait.

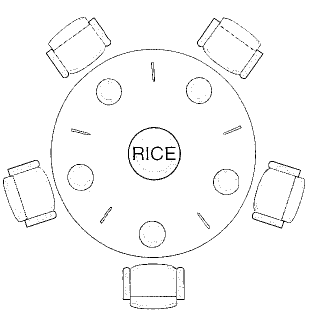


Fig 1: Dining philosopher problem.

This setup of simple dining problem represents a synchronization and recourse sharing problems which are faced while the interaction of different processes with the recourses or with another process. This solution of the problem demands such an algorithm which can ensure the fairness in giving chance to all philosophers to share chopsticks. Technically speaking, the algorithm developed for this dining philosopher problem can also be implemented to solve the real life recourse sharing problem in computer system.

**SOLUTION**

The solution to this problem can be given with the help of a Semaphore. Semaphore is Structure which is implemented to ensure the controlled sharing of the sharable resources.

typedef struct

{ int value;

Semaphore \*next;

}Semaphore;

Semaphore chopstick[i] where i belongs to [1,5]

Fig2: SemaPhore Structure

do {//think

wait(chostick[i]);

wait(chostick[(i+1)%5]);

//eat;

signal(chostick[i]);

signal(chostick[(i+1)%5];

}while(true);

Fig3: Eating process

Wait(Semaphore \*s)

{ s->value--;

If(s->value<0)

Block();}

Signal(Semaphore \*s)

{ s->value++;

If(s->value<=0)

{ invoke();// remove 1 process from the queue;

}}

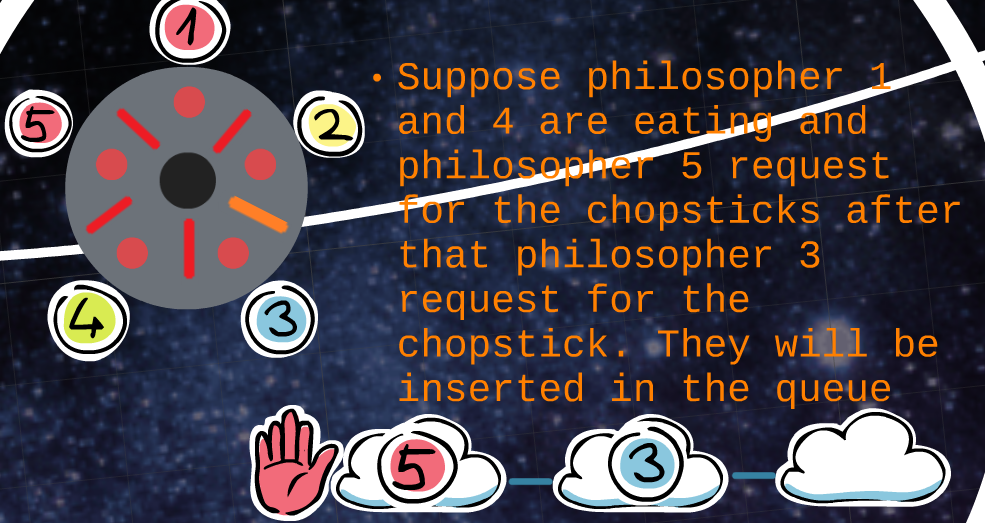
Fig5: wait and signal methods

**Explanation of the algorithm**

* First of all, initially all the chopsticks numbers are initialized to 1 that mean they are free to use.
* When every any chopstick is busy with a philosopher its number is set to 0.
* When a philosopher tries to access the chopstick, he has to go through the wait function in which the number of chopstick is decremented by 1. If the chopstick was free then the value of chopstick should be 1 before decrementing and after decrementing 0. When it is zero the process which invokes the wait function will be inserted into the queue for waiting.
* block() function is used to insert the process into a kind of waiting queue.
* Invoke() function is used to remove 1 process from the queue.

**PROBLEMS**

* **Queue Problem**



Here 5 is popped out of the queue but it is of no use because 5 still cannot eat with current condition of the dining table.

But still we have 3 in the which can eat in the current condition of the dining table

DEADLOCK

Consider a situation in which all the philosophers request at same time and all of them hold their right chopsticks simultaneously. In such situation all the philosopher will be waiting for each other to release the chopstick which will form a kind of deadlock. The occurrence of such kind of deadlock is very common on computer system where many processes share different resourse.



* STARVATION

Starvation is condition in which a philosopher is blocked by other philosophers and waiting for indefinite period of time. Such condition may arise in case of dining philosopher problem when a philosopher keep on requesting the chopstick due to which the philosopher who is waiting will keep on waiting.

SOLUTION OF ENCOUNTERED PROBLEM

* QUEUE PROBLEM

For the implementation portion we have used a special type of queue in which the element is inserted at the end but for removal, first philosopher which can eat in the current condition is selected (not necessarily first queue element).

* DEADLOCK

The condition of the deadlock is remove by enforcing a condition on the philosopher that a philosopher can acquire either 2 chopsticks or none.

* STARVATION

The condition of the starvation can be remove by ensuring that no philosopher should request the chopstick again if it is blocking the same philosopher as previous.