***Dining Philosopher’s problem***

The Dining Philosophers Problem is an illustrative example of a common computing problem in concurrency. The dining philosopher’s problem describes a group of philosophers sitting at a table doing one of two things - eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The philosophers sit at a circular table each with a bowl of spaghetti. A chopstick is placed in between each philosopher; thus, each philosopher has one chopstick to his or her left and one chopstick to his or her right. As spaghetti is difficult to serve and eat with a single chopstick, it is assumed that a philosopher must eat with two chopsticks. The philosopher can only use the chopstick on his or her immediate left or right.

***Pseudocode***

process P[i]

while (true)

do

{

think(); //initially all philosophers are thinking

pickupChopsticks (CHOPSTICK [i], CHOPSTICK [i+1 mod 5]);

eat(); //start eating

putdownChopsticks (CHOPSTICK [i], CHOPSTICK [i+1 mod 5]) //after eating puts down both the forks

}

***Explain Pseudocode***

* A philosopher will use both chopsticks (right and left) to eat.
* The remaining one chopstick may be picked up by any one of its adjacent philosophers but not both.
* A philosopher may eat if both chopsticks are available.
* After eating, a philosopher will put down both chopsticks and starts thinking again.
* Those chopsticks can be picked by the other philosophers who will repeat the same process.
* No two neighbor philosophers (right and left) can eat together.
* Initially, all philosophers are thinking. After some time, gets hungry and want to eat. The philosopher looks for the chopsticks on either side. When the philosopher gets both the chopsticks, he starts eating. After eating, he puts down the chopsticks and starts thinking again. When the philosopher puts down the chopsticks, those chopsticks may be used by neighbor philosophers.
* In such a case, there is the possibility of deadlock, a condition in which two or more processes cannot continue execution. The problem is used to design a scheduling technique in such a way no philosopher will starve.

***The Solution to the Dining Philosophers Problem***

The solution to the dining philosophers' problem is to use Semaphore. It is a tool that is used for concurrent processes. There is a drawback of using Semaphore as a solution.

The correct solution is to let a philosopher acquire both the chopsticks or none. This can be done within a monitor wherein only one philosopher is allowed to check for availability of chopsticks at a time

(Within a test function).

***Deadlock***

* Is a condition in which two or more processes cannot continue execution.
* is a drawback of using Semaphore as a solution.

In this case the philosophers never speak to each other, which creates a dangerous possibility of deadlock.

**Examples of Deadlock**

* Suppose a scenario when all philosophers pick up the left chopstick and wait for the right chopstick. The situation leads to a deadlock.
* if every philosopher holds a left chopstick and waits perpetually for a right chopstick. Originally used as a means of illustrating the problem of deadlock, this system reaches deadlock when there is a ’cycle of unwarranted requests’. In this case philosopher P1 waits for the chopstick grabbed by philosopher P2 who is waiting for the chopstick of philosopher P3 and so forth, making a circular chain.

**To avoid the deadlock solution, there are some measures that should have been taken are**

* A philosopher should allow picking chopsticks when both chopsticks are available at the same time.
* The philosophers can alternatively eat and think. For example, if the first philosopher is eating then the adjacent neighbor philosophers should wait and think, and so on.
* Each philosopher can get the chance to eat in a certain finite time.

***Starvation***

* Starvation happens when a low priority program requests a system resource but cannot run because a higher priority program has been employing that resource for a long time.

**Examples of Starvation**

* when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.
* Though simplistic, this solution may still lead to starvation (literally) when every philosopher around the table is holding his/her left chopstick and waiting for the right one.

**To avoid the starvation solution, there are some measures that should have been taken are**

* A philosopher should allow picking chopsticks when both chopsticks are available at the same time.
* When few Philosophers are waiting then one gets a chance to eat in a while.
* 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.

***Real world application:***

The problem is that we have 5 customers, and each customer wants to reserve two **adjacent** rooms.

The data set has five rooms, and each customer has two states either reserving the room or waiting on the waiting list to book the room.

When the two adjacent rooms are free the customer reserve the room

When one of the two adjacent rooms is not free the customer waits on the waiting list.

***The Solution of the real word application***

The solution to the real word application is to use Semaphore. It is a tool that is used for concurrent processes. There is a drawback of using Semaphore as a solution.

The correct solution is to let a customer reserve both the room or none. This can be done within a monitor wherein only one customer is allowed to check for availability of rooms at a time

(Within a test function).

**To avoid the deadlock solution, there are some measures that should have been taken are**

* A customer should allow booking rooms when both rooms are available at the same time.
* The customers can alternatively reserve and wait. For example, if the first customer is reserving then the adjacent neighbor customer should wait, and so on.
* Each customer can get the chance to reserve in a certain finite time.

**To avoid the starvation solution, there are some measures that should have been taken are**

* A customer should allow booking rooms when both rooms are available at the same time
* When a few customers are waiting then one gets a chance to reserve in a while.
* No customer may starve. For example, suppose you maintain a queue of customers. When a customer wants to book a room, he/she gets put onto the tail of the queue. A customer may eat only if he/she is at the head of the queue, and if the rooms are free.