**The Dining Philosophers Problem – Semaphore Solution**

**What Is The Dining Philosophers Problem**

This problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each philosopher and 5 chopsticks. A philosopher needs both a right and left chopstick to eat. If both not available, the philosopher puts down their forks and begins to think again.

This is a demonstration of a classic synchronization problem as it shows a large class of concurrency issues.

**What Is A Semaphore**

A semaphore is an integer variable S that is initialized with the number of resources present which is then used for process synchronization. There are two functions present to change the variable S which is **Wait()** & **Signal().** Both functions are used to change the value of the semaphore but only one function can change the semaphore at one time. There are two categories of semaphores **Counting Semaphores** and **Binary Semaphores**.

**Counting Semaphores**

The semaphore is initialized with the number of resources that are available. So, when a process needs a resource **the Wait()** function is called upon the value of the semaphore is decreased by one. The process the uses the resource gathered and after its use the **Signal()** function is called, and the value of the semaphore is increased by one. So, when the value of the semaphore variable reaches 0 there are no resources left so the next process that needs resources must wait which then achieves process synchronization. Process Synchronization is the mission of coordinating the execution of processes in a way that no two processes try to gain access to the same shared resource or data at the same time.

**Binary Semaphores**

The value of semaphore variable is either 0 or 1, the variable is first set to 1 and if a process wants to sue some resource, then the **Wait()** function is called, and the variable is then changed to 0. The process then uses the resource gathered and when the resource is released the **Signal()** function is called and the variable is set back to 1. If at any time the variable is at 0 and another process needs the resource, it must wait for the resource to be released by the previous process. This creates Process Synchronization.

**Implementation Of Semaphores Into The Dining – Philosophers Problem**

**Philosophers States**

**Thinking:** When a philosopher is **Thinking** he is not interacting with his colleagues(Resources)

**Eats:** When a philosopher gets **Hungary,** he turns into an **Eating** state by picking up the two forks that are closest to him(Left & Right), he must only pick up one fork at a time and not a fork from his neighbour beside him.

When a philosopher has two forks in his hand at the same time, he then eats the noodles without releasing the forks. When he is done eating h then puts down his forks and goes back into a **Thinking** state.

**Solution**

First, we must represent each of the forks with a **Semaphore,** A philosopher tries to grab a fork from the table by executing the **Wait()** operation on that semaphore. He then goes on to release the fork when done **Eating** by executing the **Signal()** operation on the appropriate semaphore.

**Semaphore forks[5];** - This is shared data of an array of forks that equal to 5. Also, all chopsticks are initialized to 1, when a fork is initialized to 1 it is free to use and when it is initialized to 0 it is not free to use. The method that is going to be used is a **Binary Semaphore**.

**The Code Solution**

The Structure Of The Philosopher

do{

    //The first wait operation is when the philosopher gets hungry and uses the wait()

    //function to grab the fork on his left

    wait (forks[i]);

    //This wait function is used to get the second fork from the philosopher adjacent to him

    //We have %5 due to i == 5 and + 1 would == 6 so we need % 5 to make it one to get the free fork

    wait(forks[(i + 1) % 5]);

    //Eat

    //The signal function make sure that when the philosopher is done eating he return the forks

    signal(forks[i]);

    signal(forks[(i + 1) % 5]);

    //Thinks

    //Returns to a thinking state when he is done eating

}while(true)

This solution now guarantees that no two neighbours(Philosophers) will be eating simultaneously, There can be a deadlock created now. A deadlock is when to processes are sharing the same resource are preventing each other from accessing the resource.

**How Is Deadlock Created Within This Problem**

Say that all the 5 philosophers go into a **Hungry** state all at the same time and they all go to grab their left fork, all the fork elements will now equal to 0 which means they can not be used. And as they are all sitting at a round table this means the fork on their right will now be equal to 0 which means this **resource** is in use which means they will never be able to go into an **Eating** state. Which now creates a deadlock.

**How To Avoid This Deadlock**

* Allow only four philosophers at a table at one time, what happens here is that if all of them get hungry when they all grab their forks due to there only being four at the table this leaves a fork left over for a philosopher to go into an Eating state. And when he finishes the next philosopher will get is chance.
* Only allow the philosophers to pick up both forks if they are available instead of just instantly picking up their left fork.
* They could also use a asymmetric solution which makes an ODD philosopher pick up first his left chopstick and then his right chopstick, and then an EVEN philosopher picks first his right chopstick and then his left chopstick.