**Dinning philosopher problem**

**Solution pseudocode**

program idea:

this program is designed to help illustrate synchronization issues and techniques for resolving them by checking the philosopher's state and prioritizing which philosopher holds 2 chopsticks to eat without causing deadlock or starvation.

//initialize an array that contains 5 Philosophers

const num\_of\_Philosophers=5

Philosophers=[num\_of\_Philosophers]

//create the table that the philosophers will sit on

by assigning each philosopher to thinking state

and implementing hold\_chopsticks () {

acquire reentrantlock lock on the chopsticks to synchronize threads and prevent starvation by checking the oldest thread on the server and give it the lock

that when called sets the current philosopher state to wants to eat

and then checks if (the next and previous philosophers are not eating)

then set the philosophers current state to eating

Else

the philosophers wait till the next an previous philosophers are not eating

release lock;

}

and implementing the function release chopsticks () {

acquire reentrantlock lock on the chopsticks to synchronize threads and prevent starvation by checking the oldest thread on the server and give it the lock

that when called sets the current philosopher state to thinking to indicate that the philosopher is done eating

and then letting neighbours know that forks are on the table to use and check IF the neighbours (previous and next philosopher) wants to eat

release lock;

}

while(true)

{

create thread for each philosopher

assign each philosopher to the Table

and then each philosospher's thread starts

philosophers Alternate between Thinking and Eating States :

Philosophers[i] state is "thinking":

philosopher takes his time to think on what to eat

where each philosopher takes different time than the other

Philosophers[i] state is "wants to eat":

the philosopher calls hold\_chopsticks () function to check if he can eat

Philosophers[i] state is "starts eating":

philosopher takes his time to eat

where each philosopher takes different time than the other

Philosophers[i] state is "finished":

the philosopher calls release\_chopsticks() function to return chopsticks and check if neighbours wants to eat

}

**Examples of Deadlock**

Deadlock occurs if these conditions are satisfied:

* Mutual exclusion -- at least one resource must be held by a process.
* Hold and wait -- at least one process holds a resource while it is waiting for another resource.
* No preemption -- one process can't take another process's resources in order to make progress (nor can the OS)
* Circular wait -- there exists a circular chain of processes, each of which is waiting for a resource held by the next process in the chain

Deadlock could occur if every philosopher holds a left chopstick and waits perpetually for a right chopstick (or vice versa). Originally used as a means of illustrating the problem of deadlock, this system reaches deadlock when there is a 'cycle of unwarranted requests

**How did solve deadlock**

We can solve this problem as follows philosopher may pick up her chopsticks only if both are available. To code this solution, we need to distinguish among three states in which we may find a philosopher:

THINKING – When philosopher doesn’t want to gain access to either fork.

HUNGRY – When philosopher wants to enter the critical section.

EATING – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating

(State[(i+4) % 5]! = EATING) and (state[(i+1) % 5]! = EATING).

We also need to declare function wait ().

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

**Examples of Starvation**

Starvation occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress, or resource starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work. Starvation may be caused by mutual exclusion algorithm, the problem is how to design a discipline of behavior such that no philosopher will starve.

In dinner philosophers problem starvation happens when the philosopher is hungry it tests if the other philosopher on his right side is either eating or not if he is eating the philosopher comes out waits a while and tries to check again if it finds the right philosopher not eating and the left philosopher is eating then he will come out again and waits, if this situation will repeat endlessly this cause starvation

**How did solve starvation**

The solution for this problem is simply by putting priority on the ones that have waited the longest. By giving priority, we can prevent starvation from happening, we did that by using Reentrantlock.

**Explanation for real world application and how did apply Dining**

**Philosopher’s problem**

* Producer-consumer

– Audio-Video player: network and display threads; shared buffer

– Web servers: master thread and slave thread

* Reader-writer

– Banking system: read account balances versus update

* Dining Philosophers

– Cooperating processes that need to share limited resources

Set of processes that need to lock multiple resources

– Disk and tape (backup),

Travel reservation: hotel, airline, car rental databases

**Banker's Algorithm:**

* This algorithm handles multiple instances of the same resource.
* Force threads to provide advance information about what resources they may need for the duration of the execution.
* The resources requested may not exceed the total available in the system.
* The algorithm allocates resources to a requesting thread if the allocation leaves the system in a safe state.
* Otherwise, the thread must wait.

**Summary:**

* Deadlock: situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set.
* Detection and recovery: recognize deadlock after it has occurred and break it.
* Avoidance: don't allocate a resource if it would introduce a cycle.
* Prevention: design resource allocation strategies that guarantee that one of the necessary conditions never holds
* Code concurrent programs very carefully. This only helps prevent deadlock over resources managed by the program, not OS resources.
* Ignore the possibility! (Most OSes use this option!!)