Dining Philosopher’s Problem:

The Dining Philosophers Problem is an illustrative example of a common computing problem in concurrency.

The dining philosophers 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

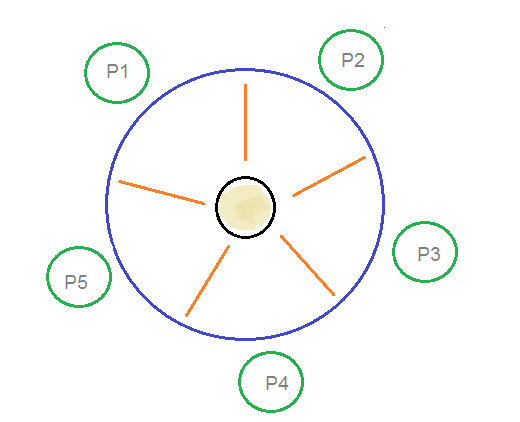
philosophers sit at a circular table each with a bowl of spaghetti. A chopstick is placed in between each

philosopher, thus each philosopher has one chopstick to his or her left and one chopstick to his or her right.

As spaghetti is diffcult 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.

1) Pseudocode

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.

Pseudocode:

while(TRUE)

{

wait(stick[i]);

/\*

mod is used because if i=5, next

chopstick is 1 (dining table is circular) \*/

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

/\* eat \*/

signal(stick[i]);

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

/\* think \*/

}

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

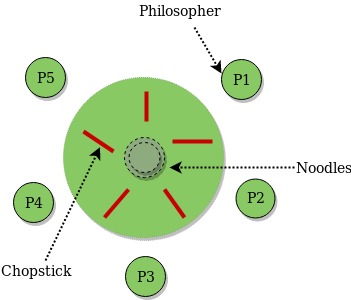
But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

* A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
* Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

**Deadlock :**

deadlock is a situation wherein two or more competing actions are waiting for the other to Finish, and thus none ever stop waiting. It is a logical error that can occur when programming with threads. Note that deadlock isn't starvation because in starvation progress is being made but in deadlock there is no progress.

**Example on Deadlock :**

****

Deadlock is the permanent blocking of two or more threads based on four necessary conditions. The first three are general properties of synchronization primitives that are typically unavoidable. The last is a system state that arises through a sequence of events.

* [**Mutual exclusion**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-mutual-exclusion): Once a resource has been acquired up to its allowable capacity, no other thread is granted access.
* [**No preemption**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-no-preemption): Once a thread has acquired a resource, the resource cannot be forcibly taken away. For instance, only the owner of a mutex can unlock it.
* [**Hold and wait**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-hold-and-wait): It is possible that a thread can acquire one resource and retain ownership of that resource while waiting on another.
* [**Circular wait**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-circular-wait): One thread needs a resource held by another, while this second thread needs a different resource held by the first.
* Deadlock solution :

• Make every even numbered philosopher pick up the right Chopstick first and every odd numbered philosopher pick up the left Chopstick first

• Don't let them all eat at once

— a philosopher has to enter a monitor to check if it is safe to eat

• can only get into the monitor if no one else in it

— each philosopher checks and sets some state indicating their condition

* Deadlock prevention

Preventing deadlocks is fairly easy. Remember that the list presented before enumerates conditions which

are all necessary for a deadlock to occur; therefore, it su\_ces that at least one of those conditions does not

hold.

1. Removing the mutual exclusion condition means that no thread may have exclusive access to a resource This isn't very practical if we want concurrency.

2. A "no preemption" condition may also be di\_cult or impossible to avoid as a thread has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent. This also isn't very practical.

Using Semaphore :

We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem. Wait and Signal operations will be used for the solution of the Dining Philosophers Problem, for picking a chopstick wait operation can be executed while for releasing a chopstick signal semaphore can be executed.

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

 wait( S )

{

**while**( S <= 0) ;

S--;

}

 signal( S )

{

S++;

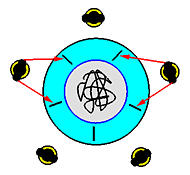
}

Starvation :

starvation happens when a particular philosopher is not getting a chance to eat at regular intervals, there could be a scenario that one of the philosophers might overeat eventually leading to starvation of another philosopher, even though this addresses the deadlock issue.

Example of Starvation :

**Starvation is also a 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!



How to Solve Starvation ?

### Preventing starvation

So, in order to prevent starvation you either need:

* A guarantee from the thread system that threads will be unblocked from monitors and condition variables in the sameorder that they are blocked. In such a case, solution #4 will work fine, since a blocked philosopher will then be guaranteed to get a chopstick once it is released. Of course, this doesn't solve the problem that philosopher 4 gets to eat more than the rest, but it does prevent starvation.

To do it yourself. In other words, 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.

implements this. When a philosopher calls **pickup()**, if the queue is empty, thechopsticks 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. Note how this checking must be performed in a monitor. When **putdown()** is called, the chopsticks are released, and then **test\_queue()** is called, which checks the head of the queue to see if the philosopher there can eat. If so, that philosopher is unblocked, and then he/she can eat.

Try the program out to see that it works. Moreover, note that there are times when a philosopher can call **pickup()** and the sticks can be available, but the philosopher blocks. This is because the queue isn't empty. Thus, the solution may not allow philosophers to eat as much as they would like, but it does prevent starvation. Think about ways that you could prevent starvation, but also allow less blocking time for philosophers.

* How to apply on real world :

 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. Any 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).