**ASSIGNMENT NO.**

**Title**: Dining Philosopher’s Problem.

**Aim**: To implement Dining Philosopher’s Problem Algorithm.

**Objective**:

1. To implement Dining Philosopher’s Problem which uses shared memory between neighboring processes to consume data.
2. Use latest open source software modeling implement algorithm and perform testing.

**Theory**:

* The dining philosopher’s problem is a classic concurrency problem dealing with synchronization.
* In this problem Five philosophers are sitting at a table eating rice. Each philosopher has two forks: left fork and right fork
* Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks.
* Each fork can be held only one philosopher and so philosopher can use the fork only if it is not being used by another philosopher.
* After one finishes eating,he needs to put down both forks so they become available to others.
* A philosopher can start eating food only if he has two forks in his hand.

Rules:

The philosophers are very logical

* + They want to settle on a shared policy that all can apply concurrently
  + They are hungry: the policy should let everyone eat (eventually)
  + They are utterly dedicated to the proposition of equality: the policy should be totally fair

Problems in solution:

Primarily, we worry about:

* + Starvation: A policy that can leave some philosopher hungry in some situation. (even one where the others collaborate)
  + Deadlock: A policy that leaves all the philosophers “stuck”, so that nobody can do anything at all.
  + Livelock: A policy that makes them all do something endlessly without ever eating.

**Mathematical Modeling:**

M={I, O, Fn, Sc, Fc}

Input I = {I1}

Where I1=Initial state.

Output O = {O1}

Where O1=State of every philosopher.

Functions Fn = {Fn1, Fn2, Fn3}

Where Fn1=Create threads.

Fn2=Eat.

Fn3=Think.

Success cases Sc = {Sc1, Sc2}

Where Sc1=Threads run parallelly.

Sc2=Locking mechanism works correctly.

Failure cases Fc = {Fc1, Fc2}

Where Fc1=Exceptions in threads.

Fc2=Locking mechanism works incorrectly

**Activity diagram:**

Activity diagrams are graphical representations of [workflows](https://en.wikipedia.org/wiki/Workflow) of stepwise activities and actionswith support for choice, iteration and concurrency. In the [Unified Modeling Language](https://en.wikipedia.org/wiki/Unified_Modeling_Language), activity diagrams are intended to model both computational and organizational processes (i.e. workflows). Activity diagrams show the overall flow of control.Activity diagrams are constructed from a limited number of shapes, connected with arrows. The most important shape types:

* rounded rectangles represent actions;
* diamonds represent decisions;
* bars represent the start (split) or end (join) of concurrent activities;
* a black circle represents the start (initial node) of the workflow;
* an encircled black circle represents the end (final node).

Arrows run from the start towards the end and represent the order in which activities happen.

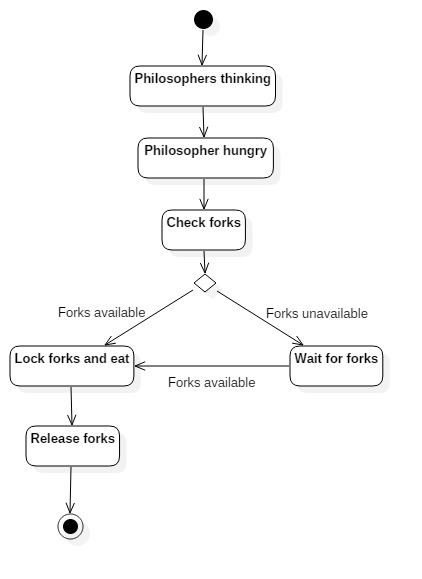


Fig: Activity Diagram

**State Chart Diagram:**

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a [component](https://www.tutorialspoint.com/) in a system. The states are specific to a [component](https://www.tutorialspoint.com/)/object of a system.

A [Statechart](https://www.tutorialspoint.com/) diagram describes a state machine. Now to clarify it state machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

[Activity](https://www.tutorialspoint.com/) diagram explained in next chapter, is a special kind of a [Statechart](https://www.tutorialspoint.com/) diagram. As [Statechart](https://www.tutorialspoint.com/) diagram defines states it is used to model lifetime of an object.

Following are the main purposes of using [State chart diagrams](https://www.tutorialspoint.com/):

* To model dynamic aspect of a system.
* To model life time of a reactive system.
* To describe different states of an object during its life time.
* Define a state machine to model states of an object.

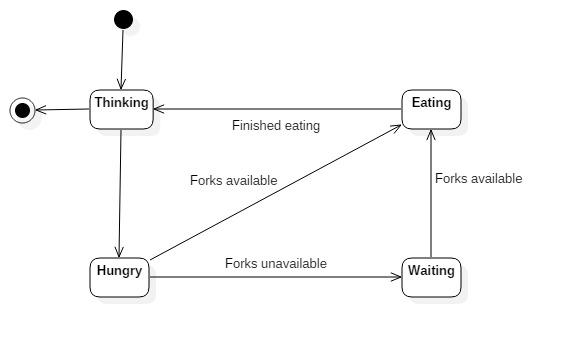


Fig: State Chart Diagram

**System testing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Id** | **Step** | **Input** | **Expected Output** | **Actual Output** | **Status** |
| T1 | No. of Activities | Run a number of philosophers simultaneously | Able to accept reliable Activities | Accepts all generated activities | Pass |
| T2 | Locking and Unlocking | Try to access forks by all the philosophers at the same time | Only single philosopher allowed access to the fork | At a time only one client is available to access the fork | Pass |
| T3 | Acquiring and releasing locks | Try to access locks on the already locked file. | It should get terminated due to lot of exceptions | It doesn’t crash | Fail |
| T4 | Fork Availability | Try to access the fork by the philosopher | The philosopher should not work properly | Client application crashes | Pass |

**Algorithm:**

1. Define the number of philosophers.
2. Declare one thread per philosopher.
3. Declare one semaphore (represent chopsticks) per philosopher.
4. if philosopher is hungry and both chopsticks are free

then {Acquire both chopsticks andEat}

Restore the chopsticks.

1. if chopsticks != free {then wait till they are available}

**Output:** State of every philosopher.

**Platform:** Ubuntu.

**Language:** c.

**Conclusion**: Implemented Dining Philosopher’s Problem using shared memory and also performed manual testing.