# Assignment No. 6

**Aim:** Implement Concurrent Dining Philosopher Problem.

**Theory:**

The problem of the dining philosophers, first proposed by Edsger Dijkstra and reformulated by Tony Hoare, is a famous problem for concurrent programming that illustrates problems with synchronizing access to data.

The description of the problem, is as the following:

- Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.
- Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The idea is to find a solution so that none of the philosophers would starve, i.e. never have the chance to acquire the forks necessary for him to eat.

**Code:**

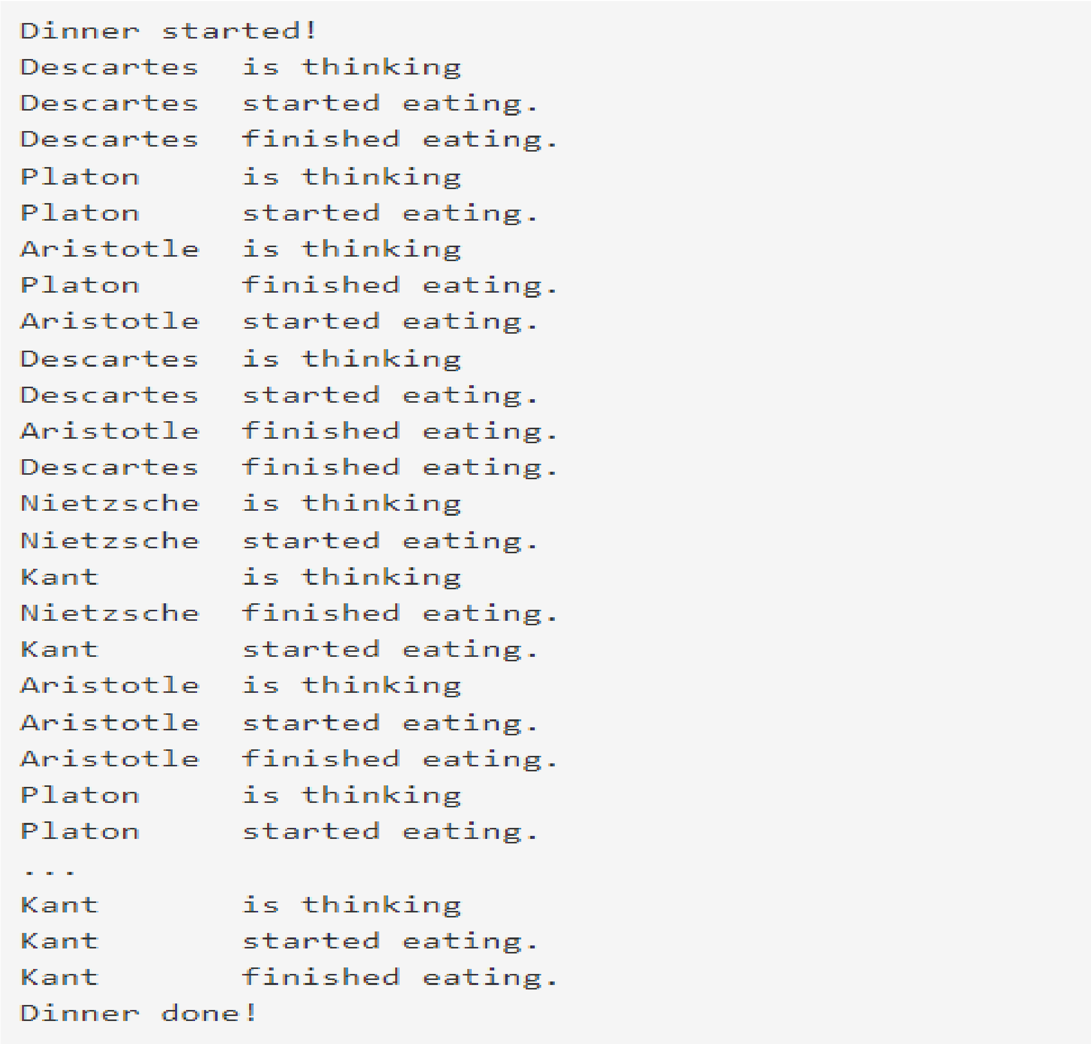
```
#include <array>
#include <mutex>
#include <thread>
#include <atomic>
#include <chrono>
#include <iostream>
#include <string>
#include <random>
#include <iomanip>
#include <string_view>
std::mutex g_lockprint;
```

```
constexpr  int no_of_philosophers = 5;
struct fork
{
  std::mutex mutex;
};
struct table
{
  std::atomic<bool>                ready{ false };
  std::array<fork, no_of_philosophers> forks;
};
struct philosopher
{
private:
  std::string const name;
  table const &    dinnertable;
  fork&           left_fork;
  fork&           right_fork;
  std::thread      lifethread;
  std::mt19937      rng{ std::random_device{}() };
public:
  philosopher(std::string_view n, table const & t, fork & l, fork & r) :
                         name(n),     dinnertable(t),     left_fork(l),     right_fork(r),
lifethread(&philosopher::dine, this)
  {
  }
  ~philosopher()
  {
    lifethread.join();
  }
  void dine()
  {
    while (!dinnertable.ready);
    do
    {
      think();
      eat();
    } while (dinnertable.ready);
  }
  void print(std::string_view text)
  {
    std::lock_guard<std::mutex> cout_lock(g_lockprint);
    std::cout
      << std::left << std::setw(10) << std::setfill(' ')
      << name << text << std::endl;
  }

  void eat()
  {
    std::lock(left_fork.mutex, right_fork.mutex);
    std::lock_guard<std::mutex> left_lock(left_fork.mutex,   std::adopt_lock);
    std::lock_guard<std::mutex> right_lock(right_fork.mutex, std::adopt_lock);
    print(" started eating.");
    static thread_local std::uniform_int_distribution<> dist(1, 6);
    std::this_thread::sleep_for(std::chrono::milliseconds(dist(rng) * 50));
    print(" finished eating.");
  }
  void think()
  {
    static thread_local std::uniform_int_distribution<> wait(1, 6);
    std::this_thread::sleep_for(std::chrono::milliseconds(wait(rng) * 150));
    print(" is thinking ");
  }
};
void dine()
{
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "Dinner started!" << std::endl;
  {
    table table;
    std::array<philosopher, no_of_philosophers> philosophers
    {
      {
        { "Aristotle", table, table.forks[0], table.forks[1] },
        { "Platon",    table, table.forks[1], table.forks[2] },
        { "Descartes", table, table.forks[2], table.forks[3] },
        { "Kant",      table, table.forks[3], table.forks[4] },
        { "Nietzsche", table, table.forks[4], table.forks[0] },
      }
    };
    table.ready = true;
    std::this_thread::sleep_for(std::chrono::seconds(5));
    table.ready = false;
  }
  std::cout << "Dinner done!" << std::endl;
}

int main()
{
  dine();
  return 0;
}
```

**Output:**



```
Dinner started!
Descartes   is thinking
Descartes   started eating.
Descartes   finished eating.
Platon      is thinking
Platon      started eating.
Aristotle   is thinking
Platon      finished eating.
Aristotle   started eating.
Descartes   is thinking
Descartes   started eating.
Aristotle   finished eating.
Descartes   finished eating.
Nietzsche   is thinking
Nietzsche   started eating.
Kant        is thinking
Nietzsche   finished eating.
Kant        started eating.
Aristotle   is thinking
Aristotle   started eating.
Aristotle   finished eating.
Platon      is thinking
Platon      started eating.
...
Kant        is thinking
Kant        started eating.
Kant        finished eating.
Dinner done!
```