

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**DEPARTMENT OF COMPUTER ENGINEERING**

**COURSE CODE**: CEF 427

**COURSE TITLE:** ADVANCED OPERATING SYSTEM

**1ST SEMESTER 2023/2024**

**PROJECT TITLE**

THE DINNING PHILOSOPHERS PROBLEM

PRESENTED BY:

|  |  |
| --- | --- |
| NAMES | MATRICULE |
| NANGLEFACK Leodia FIETSOP | FE21A244 |
| NGUEDIA JEATSA Joyce Grace | FE21A263 |
| QUINUEL TABOT NDIP-AGBOR | FE21A300 |
| SAMUEL OSOH | FE21A303 |
|  |  |

INSTRUCTOR: DR.SOP Lionel

**TABLE OF CONTENT**

[ABSTRACT 3](#_Toc152204689)

[Introduction 4](#_Toc152204690)

[The Dining Philosopher problem 5](#_Toc152204691)

[SOLUTION 6](#_Toc152204692)

[Constraints and properties of the solution 6](#_Toc152204693)

[Semaphore solution to the dinning philosopher 7](#_Toc152204694)

[Code for n philosophers using c 7](#_Toc152204695)

[example 8](#_Toc152204696)

[Code for 5 philosophers using c 8](#_Toc152204697)

[OUTPUT 10](#_Toc152204698)

[CONCLUSION 10](#_Toc152204699)

[REFERENCES 11](#_Toc152204700)

# ABSTRACT

Semaphores are normal variables used to coordinate the activities of multiple processes in a computer system.

They are used to enforce mutual exclusion, avoid race conditions and implement synchronization between processes. Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

This is demonstrated in the dinning philosopher problem.

Since using semaphores activities are well coordinated in the CPU, there is less risk of *DEADLOCK* (general starvation).

# Introduction

The operating system is a program that links the user and the computer system. This operating system must be capable of controlling resource usage. In the process of designing the operating system, there is a common foundation called concurrency. Concurrent processes are when the processes work at the same time. This is called the multitasking operating system. Concurrent processes can be completely independent of the other but can also interact with each other [1]. Processes that require synchronization to interact properly controlled. However, the concurrent processes that interact, there are some problems to be solved such as deadlock and synchronization. One of the classic problems that can illustrate the problem is the Dining Philosophers Problem. Dining Philosophers Problem can be illustrated as follows; there are five philosophers who would eat. On the table was reserved five chopsticks. If philosophers really hungry, then it will take two chopsticks, which is in the right and left hands. However, sometimes only one course takes chopsticks. If there are philosophers who took two chopsticks, then there are philosophers who have to wait until the chopsticks are placed back. Inside this problem, there is the possibility of deadlock, a condition in which two or more processes can not continue execution.

# The Dining Philosopher problem

This problem was proposed by E. Dijkstra in concurrent programming and seen then it has been considered as a classical problem.

The problem state’s that:

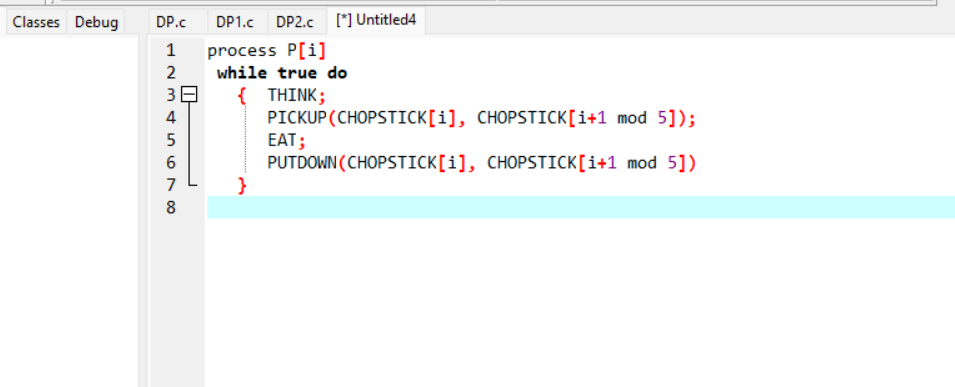
Supposing a number of philosophers are sitting around a circular table. A plat of food is served to them with one chopstick between each pair of philosophers. The life of a philosopher is thinking. But from time to time , it may happen that a philosopher gets hungry from too much thinking. He then wants to eat from the bowl of food. He has to pick the two chopsticks closest to him (his right and left). They are only as many chopsticks as they are as philosophers. A philosopher may only pick up one chopstick at a time and obviously he cannot pick up a chopstick which is already in the hand of a neighbor for example, if each time he tries to pick up a chopstick it happens to be in the hand of the appropriate neighbor, then the philosopher *STARVES*. If a hungry philosopher gets to hold both his chopsticks at the same time, he eats (without releasing his chopsticks), eventually satisfies his material needs and puts down his two chopsticks.

This means a philosopher goes indefinitely through this cycle

* Thinking
* Trying to get its chopsticks
* Eating

The problem is to describe a system of protocols for the philosophers, chopsticks and possibly some other entities, which will behave in the way indicated above (especially that at any time each chopstick is in at most one hand) and which will ensure that, with varying degrees of strength, philosophers will be able to eat.

Each philosopher is represented by the following pseudocode



# SOLUTION

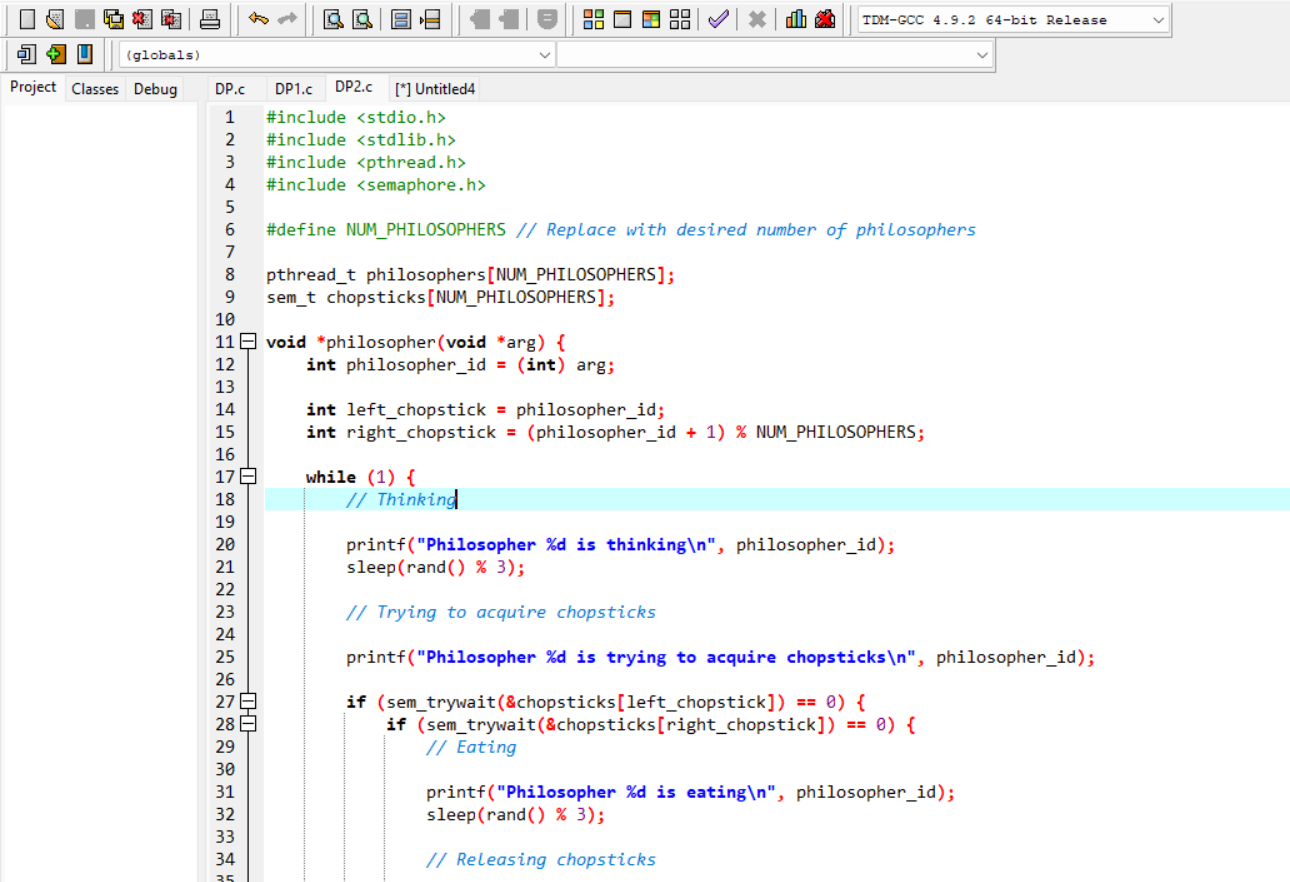
## Constraints and properties of the solution

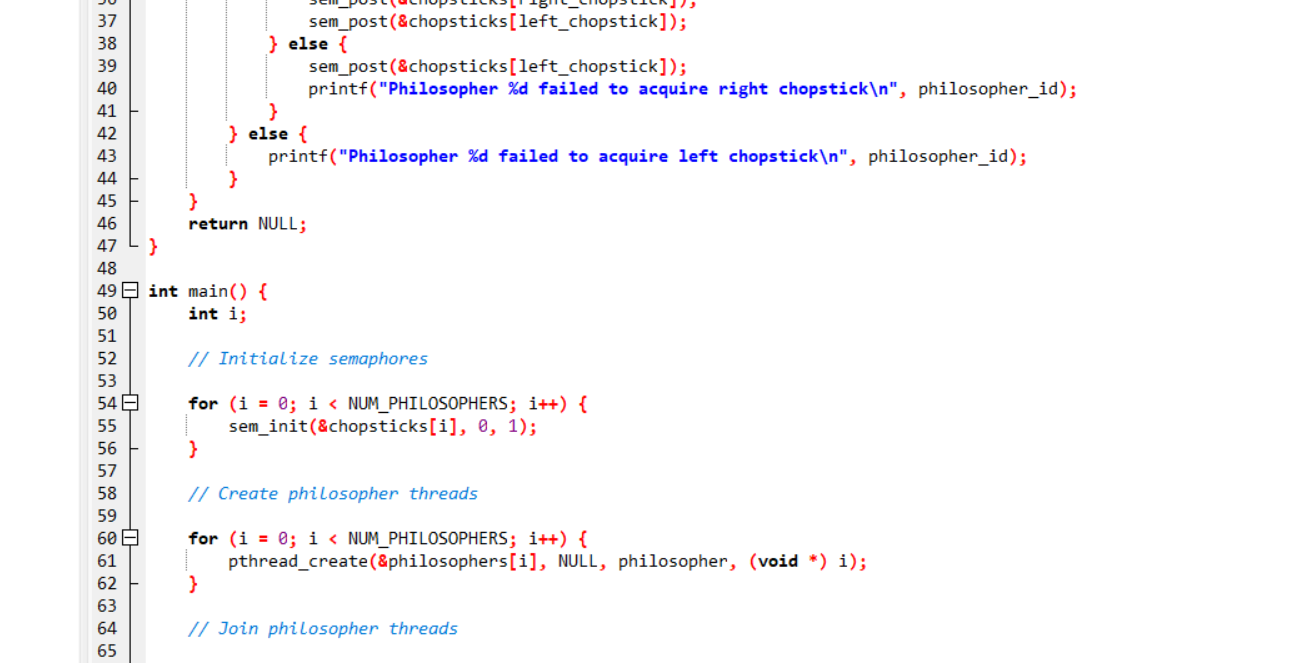
* We are interested only in truly distributed systems that is systems Where the only active agents are the philosophers who do not communicate directly with each other.
* All philosophers are identical that is we assume that at the beginning all philosophers are at the same state.

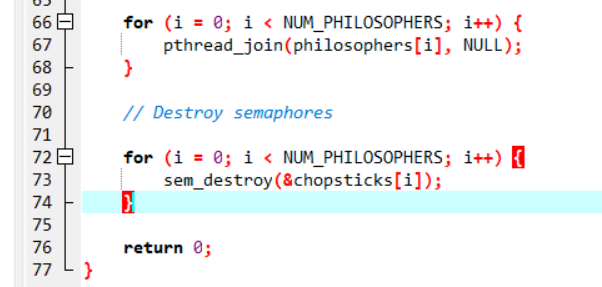
Therefore we are assuming that a philosopher may, in one move and without risk of being disturbed by or of disturbing a neighbor, check that a chopstick is down on the table and pick it up. As will be seen later the picking up and putting down of a chopstick will be expressed by a change in the value of a shared variable.

## Semaphore solution to the dinning philosopher

### Code for n philosophers using c

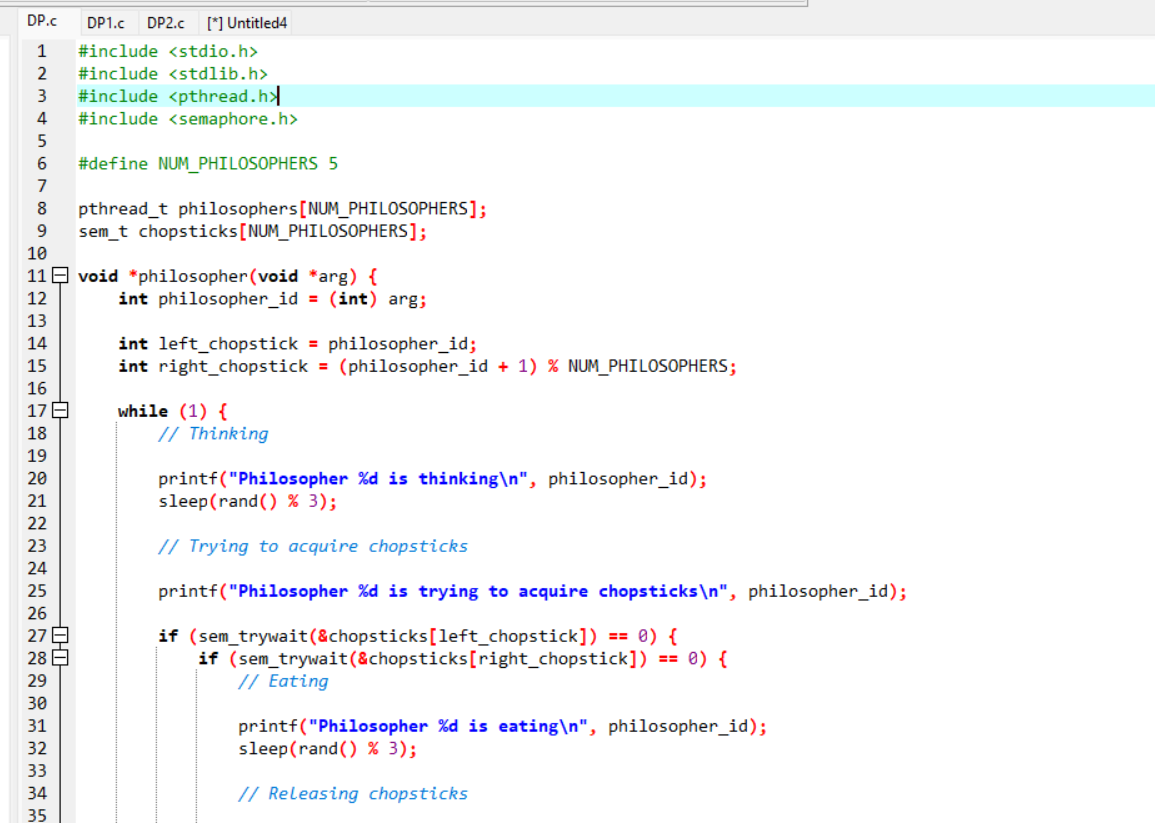
**`**

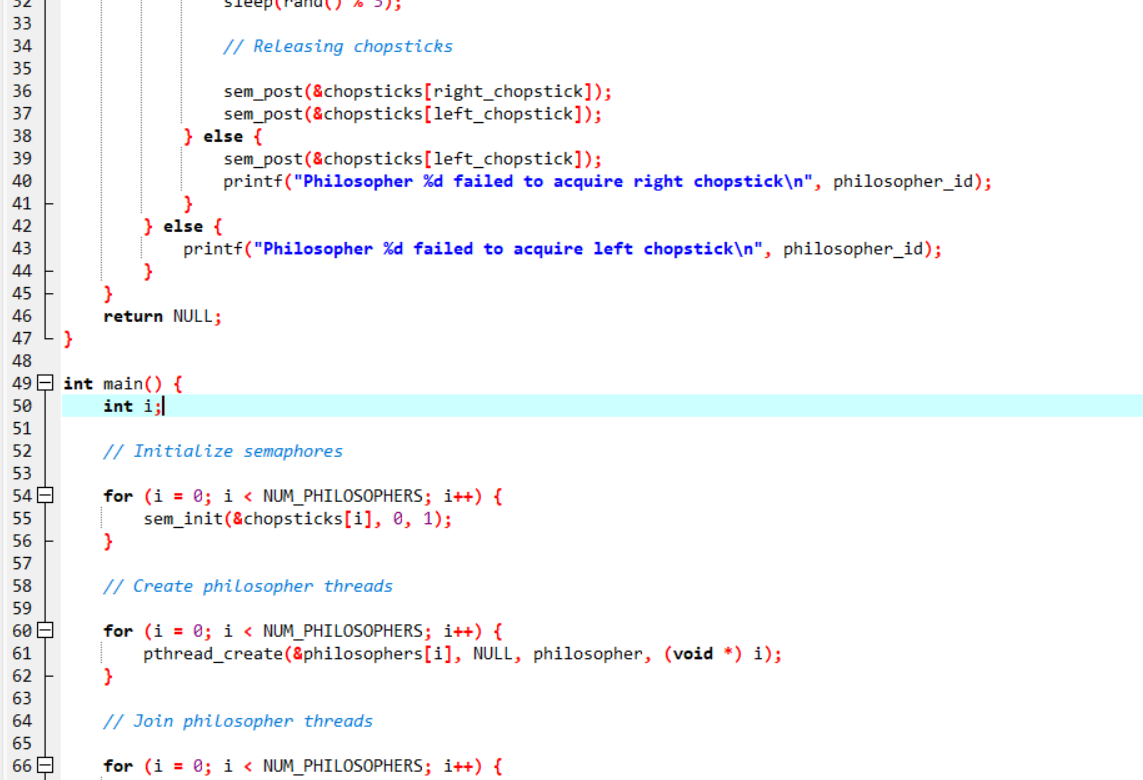
****

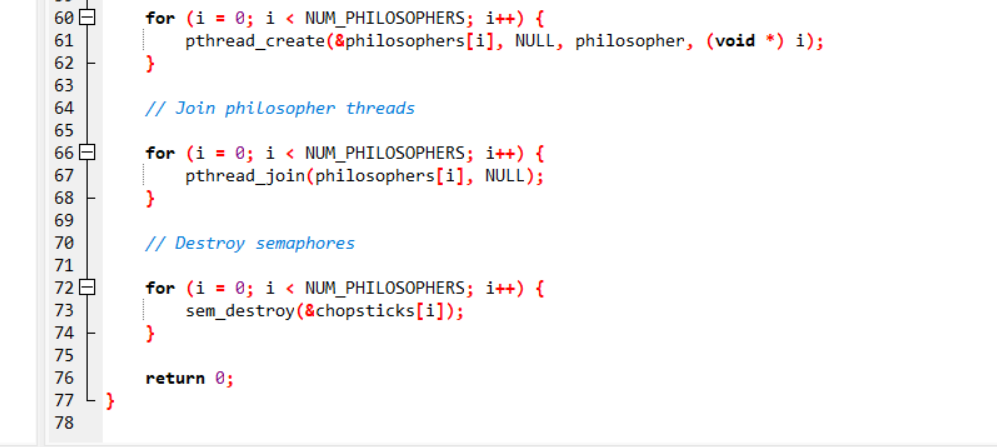
****

## example

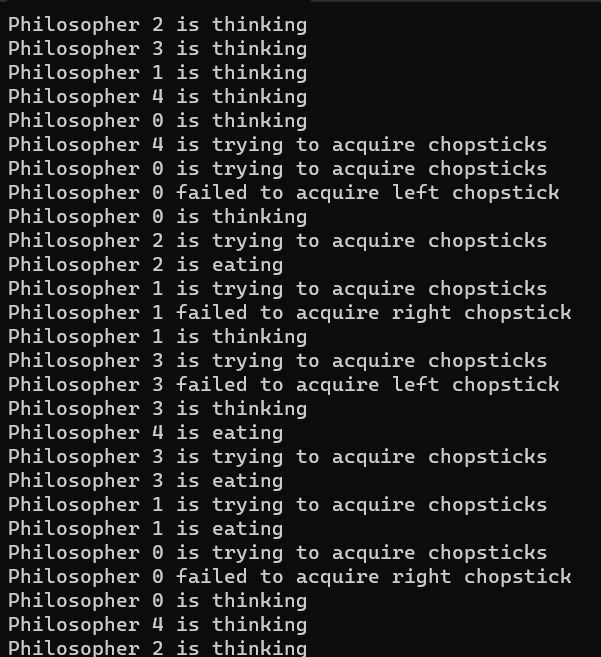
### Code for 5 philosophers using c

****

****

****

# OUTPUT

****

# CONCLUSION

The solution of the dinning philosopher problem presented here suggest an approach to the general question of programming methodology.

This explicitly shows the importance of semaphores in operating systems.

# REFERENCES

1. A SYMMETRIC AND FULLY DISTRIBUTED SOLUTION TO THE DINING PHILOSOPHERS PROBLEM by Daniel Lehmann and Michael 0. Rabin
2. DINING PHILOSOPHERS THEORY AND CONCEPT IN OPERATING SYSTEM SCHEDULING by Zuhri Ramadhan , and Andysah Putera Utama Siahaan
3. Google sources
4. Bard ai