****

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**DEPARTMENT OF COMPUTER ENGINEERING**

**COURSE CODE: CEF 427**

**COURSE TITLE: ADVANCED OPERATING SYSTEM**

**1ST SEMESTER 2023/2024**

**PROJECT TITLE**

**IMPLEMENT THE DINING PHILOSOPHER SYSTEM**

**PRESENTED BY:**

|  |  |  |
| --- | --- | --- |
| **Names** | **Matricule** | **Specialty** |
| KAMCHE YANN ARNAUD | FE21A208 | SOFTWARE |
| AMABO JOSHUA | FE21A135 | SOFTWARE |
| NKWI CYRIL AKINIMBOM | FE21A281 | SOFTWARE |
| METAGNE KAMGA MAIVA C. | FE21A237 | SOFTWARE |
| MUYANG ROSHELLA MBAMUZANG | FE21A243 | NETWORK |

**INSTRUCTOR: DR.SOP Lionel**

**1.** **Abstract**

This project addresses the Dining Philosophers problem, a classic challenge in concurrency and synchronization. The goal is to design a solution that enables philosophers to alternate between thinking and eating while avoiding deadlocks and ensuring fairness. The implemented solution utilizes semaphores, threads, and mutexes, ensuring each philosopher has access to both forks and can eat freely in a simulated dining scenario.

**2.** **Introduction:**

Concurrency issues arise when multiple processes or threads share resources. The Dining Philosophers problem is an illustrative example where philosophers seated around a table alternate between thinking and eating, utilizing shared chopsticks. The primary challenge is to prevent deadlocks and ensure fair access to resources.

**3.** **Understanding the Dining Philosophers Problem:**

The Dining Philosophers problem represents the challenges of resource allocation and synchronization in a multi-process system. Philosophers must contend for limited resources (chopsticks) to eat, and deadlocks can occur if each philosopher holds one chopstick and waits for the other. The project aims to address this classic problem by ensuring each philosopher holds both chopsticks before eating.

**4.** **Methodology:**

The Dining Philosophers problem can be tackled using synchronization techniques Using semaphores and mutexes, ensuring that philosophers could access shared resources—chopsticks—in a mutually exclusive manner. A critical aspect is empowering philosophers to make autonomous decisions regarding when to think and eat. This means allowing each philosopher to independently choose their course of action, promoting a sense of agency. The solution should be fairness by guaranteeing equal access to resources, ensuring that philosophers acquired both forks before embarking on their meal.

**5.** **Implementation in C Programming Language**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

sem\_t chopstick[NUM\_PHILOSOPHERS];

sem\_t mutex; // Mutex to protect access to the chopsticks

int cocoyams\_eaten[NUM\_PHILOSOPHERS]; // Counter for the number of cocoyams eaten

void \*philosopher(void \*arg);

void eat(int philosopher\_id);

void think(int philosopher\_id);

int main()

{

int i;

pthread\_t philosophers[NUM\_PHILOSOPHERS];

// Initialize semaphores

sem\_init(&mutex, 0, 1);

for (i = 0; i < NUM\_PHILOSOPHERS; i++)

{

sem\_init(&chopstick[i], 0, 1);

cocoyams\_eaten[i] = 0; // Initialize cocoyams eaten counter

}

// Create philosopher threads

for (i = 0; i < NUM\_PHILOSOPHERS; i++)

pthread\_create(&philosophers[i], NULL, philosopher, (void \*)(intptr\_t)i);

// Join philosopher threads

for (i = 0; i < NUM\_PHILOSOPHERS; i++)

pthread\_join(philosophers[i], NULL);

// Destroy semaphores

sem\_destroy(&mutex);

for (i = 0; i < NUM\_PHILOSOPHERS; i++)

sem\_destroy(&chopstick[i]);

return 0;

}

void \*philosopher(void \*arg)

{

int philosopher\_id = (int)(intptr\_t)arg;

while (1)

{

think(philosopher\_id);

sem\_wait(&mutex);

sem\_wait(&chopstick[philosopher\_id]);

sem\_wait(&chopstick[(philosopher\_id + 1) % NUM\_PHILOSOPHERS]);

sem\_post(&mutex);

eat(philosopher\_id);

sem\_post(&chopstick[philosopher\_id]);

sem\_post(&chopstick[(philosopher\_id + 1) % NUM\_PHILOSOPHERS]);

}

}

void eat(int philosopher\_id)

{

printf("Philosopher %d is eating. Cocoyams eaten: %d\n", philosopher\_id, ++cocoyams\_eaten[philosopher\_id]);

sleep(rand() % 5 + 1); // Simulate eating time

printf("Philosopher %d has finished eating\n", philosopher\_id);

}

void think(int philosopher\_id)

{

printf("Philosopher %d is thinking\n", philosopher\_id);

sleep(rand() % 5 + 1); // Simulate thinking time

}

**6.** **Explanation of the Code:**

1. **Semaphores and Mutex:**

- sem\_t chopstick[NUM\_PHILOSOPHERS];: An array of semaphores representing the chopsticks. Each semaphore controls access to a corresponding chopstick.

- sem\_t mutex; : A semaphore used as a mutex to protect access to the critical section, ensuring that only one philosopher at a time can access the chopsticks.

1. **Thread Creation:**

- pthread\_t philosophers[NUM\_PHILOSOPHERS];: An array of thread identifiers for each philosopher.

- A loop creates threads for each philosopher using `pthread\_create`.

1. **Initialization**

- sem\_init(&mutex, 0, 1);: Initializes the mutex semaphore with an initial value of 1, indicating it is available.

- A loop initializes each chopstick semaphore with an initial value of 1.

- int cocoyams\_eaten[NUM\_PHILOSOPHERS]; : An array to store the number of cocoyams eaten by each philosopher. Initialized to zero.

1. **Philosopher Function:**

- void \*philosopher(void \*arg): The function representing the behavior of a philosopher. It contains an infinite loop where philosophers alternate between thinking and eating.

- `think(philosopher\_id)`: Calls the `think` function to simulate thinking.

- Semaphore operations (`sem\_wait` and `sem\_post`) ensure that a philosopher acquires both chopsticks before eating and releases them afterward.

1. **Eating and Thinking Functions:**

- void eat(int philosopher\_id): Simulates the eating behavior of a philosopher. Prints the number of cocoyams eaten, increments the counter, simulates eating time, and then prints a message indicating the end of eating.

- void think(int philosopher\_id): Simulates the thinking behavior of a philosopher. Prints a message and simulates thinking time.

1. **Main Function:**

- Creates threads for each philosopher and initializes semaphores.

- Waits for all philosopher threads to finish using `pthread\_join`.

- Destroys semaphores to release system resources.

1. **Randomization:**

- sleep(rand() % 5 + 1); : Introduces randomness in the duration of thinking and eating times. The `% 5 + 1` ensures a random sleep time between 1 and 5 seconds.

**7.** **Result:**

The result of running the code can vary due to the random sleep times introduced in the `think` and `eat` functions. The program simulates philosophers engaging in a perpetual cycle of thinking and eating. Since the sleep times are random, the order in which philosophers think and eat, as well as the number of cocoyams eaten, will differ with each run.

This example illustrates the cyclic nature of the Dining Philosophers problem, with philosophers alternating between thinking and eating. The number of cocoyams eaten is incremented with each eating cycle. Keep in mind that the specific output will differ each time the code is run due to the random sleep times.

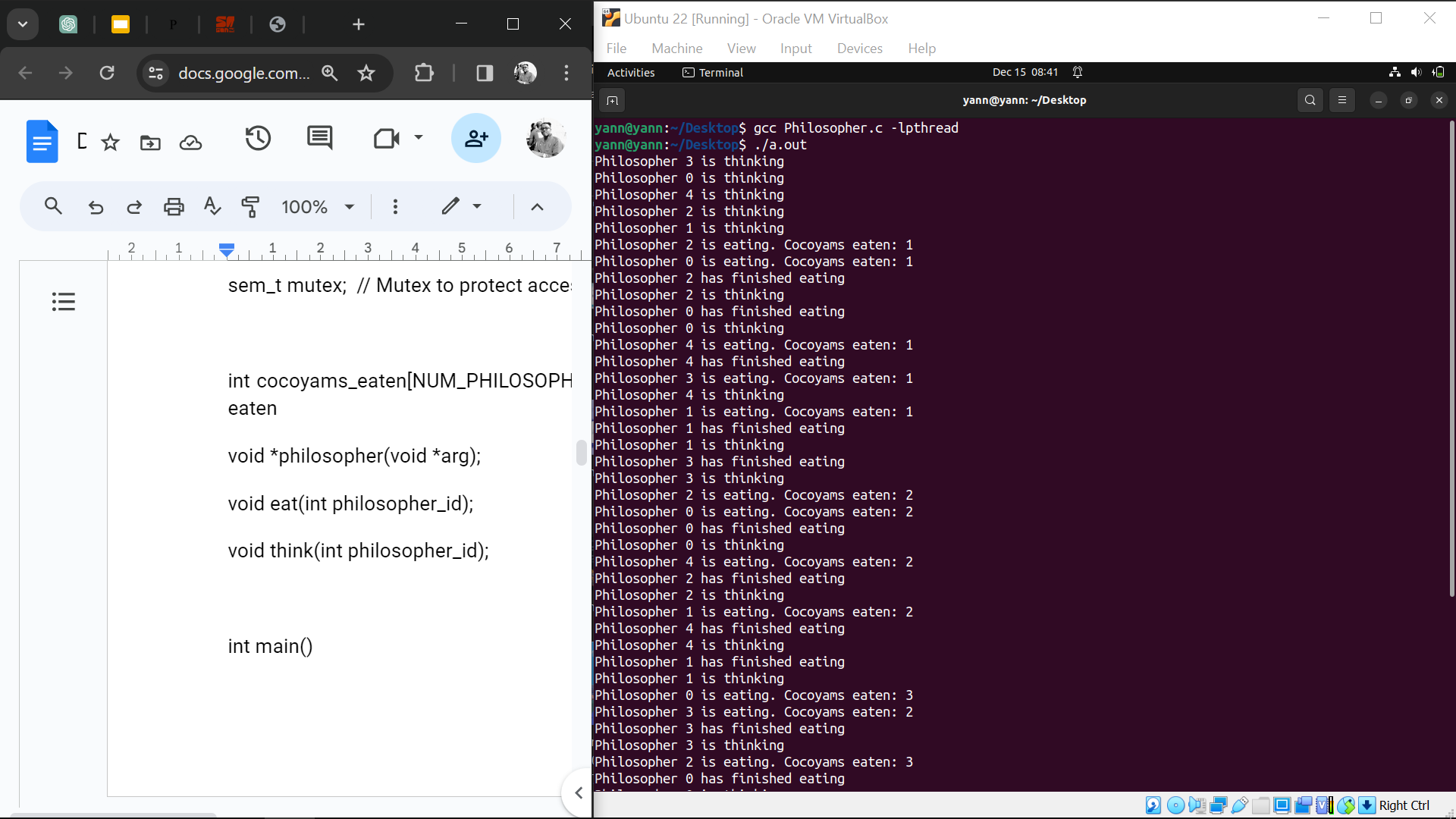
****

Fig1. Result after execution of the program (displays the cyclic nature of the problem)

**8.** **Conclusion:**

The presented solution effectively addresses the Dining Philosophers problem, considering classic synchronization challenges. Philosophers can autonomously decide when to think and eat, and equal access to resources is ensured. The code reflects principles of mutual exclusion, deadlock avoidance, and fair resource allocation.

**9.** **References:**

- Dijkstra, E. W. (1965). Solution of a Problem in Concurrent Programming Control. Communications of the ACM, 8(9), 569–570.

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). John Wiley & Sons.

- ChatGPT