C

**Batch: C1**

**Roll No.: 16010122221**

**Experiment No. 06**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

**TITLE: Implementation of Dining Philosophers problem using mutexes and semaphores.**

**AIM:** Implementation of Process synchronization algorithms using mutexes and semaphore – Dining Philosopher problem

# Expected Outcome of Experiment:

**CO 2.** To understand the concept of process, thread and resource management.

**CO 3.** To understand the concepts of process synchronization and deadlock.

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems”, McGraw Hill Third Edition.**
3. **Sumitabha Das “ UNIX Concepts & Applications”, McGraw Hill Second Edition.**

**Pre Lab/ Prior Concepts:**

Knowledge of Concurrency, Mutual Exclusion, Synchronization, Deadlock, Starvation, threads.

# Description of the chosen process synchronization algorithm:

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers.

To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available.

In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again. The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

Solution for the problem: 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.

The drawback of the above solution of the dining philosopher problem: The above solution makes sure that no two neighbouring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously.

Then none of them can eat and deadlock occurs. Some of the ways to avoid deadlock are as follows

• There should be at most four philosophers on the table

• An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.

• A philosopher should only be allowed to pick their chopstick if both are available at the same time.

**Implementation details:**

using mutexes

#include <iostream> #include <thread> #include <mutex> #include <vector> #include <chrono>

const int NUM\_PHILOSOPHERS = 5; std::mutex forks[NUM\_PHILOSOPHERS];

void eat(int philosopher\_id) {

std::cout << "Philosopher " << philosopher\_id << " is eating.\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000))

;

}

void think(int philosopher\_id) {

std::cout << "Philosopher " << philosopher\_id << " is thinking.\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000))

;

}

void philosopher(int id) { while (true) {

think(id);

std::unique\_lock<std::mutex> left\_fork(forks[id]); std::this\_thread::sleep\_for(std::chrono::milliseconds(10

));

std::unique\_lock<std::mutex> right\_fork(forks[(id + 1) % NUM\_PHILOSOPHERS]);

eat(id);

}

}

int main() {

std::vector<std::thread> philosophers;

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

philosophers.emplace\_back(philosopher, i);

}

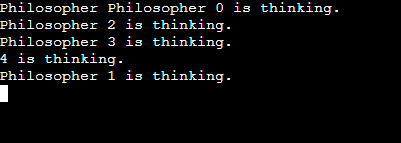
for (auto& philosopher\_thread : philosophers) { philosopher\_thread.join();

}

return 0;

}

Output



Using semaphore-

#include <iostream> #include <thread> #include <mutex>

#include <condition\_variable>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N #define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

std::mutex mutex; std::condition\_variable S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY && state[LEFT] != EATING

&& state[RIGHT] != EATING) {

state[phnum] = EATING; std::this\_thread::sleep\_for(std::chrono::milliseconds(20

00));

std::cout << "Philosopher " << phnum + 1 << " takes fork " << LEFT + 1 << " and " << phnum + 1 << std::endl;

std::cout << "Philosopher " << phnum + 1 << " is Eating"

<< std::endl;

S[phnum].notify\_all();

}

}

void take\_fork(int phnum)

{

std::unique\_lock<std::mutex> lock(mutex);

state[phnum] = HUNGRY;

std::cout << "Philosopher " << phnum + 1 << " is Hungry" << std::endl;

test(phnum);

S[phnum].wait(lock); std::this\_thread::sleep\_for(std::chrono::milliseconds(1000))

;

}

void put\_fork(int phnum)

{

std::unique\_lock<std::mutex> lock(mutex);

state[phnum] = THINKING;

std::cout << "Philosopher " << phnum + 1 << " putting fork "

<< LEFT + 1 << " and " << phnum + 1 << " down" << std::endl; std::cout << "Philosopher " << phnum + 1 << " is thinking"

<< std::endl;

test(LEFT); test(RIGHT);

}

void philosopher(int num)

{

while (true) { take\_fork(num); put\_fork(num);

}

}

int main()

{

std::thread threads[N];

for (int i = 0; i < N; i++) {

threads[i] = std::thread(philosopher, i);

std::cout << "Philosopher " << i + 1 << " is thinking"

<< std::endl;

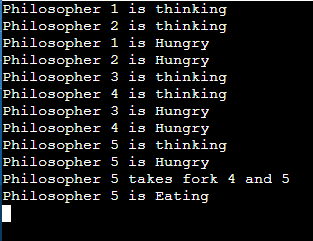
}

for (int i = 0; i < N; i++) threads[i].join();

return 0;

}

Output-



**Conclusion:**

In this experiment we successfully implemented a deadlock free solution for the dining philosophers problem using threads.

**Post Lab Descriptive Questions**

1. Differentiate between a monitor, semaphore and a binary semaphore?

Ans- **Monitor**:

* + A high-level synchronization construct that provides a convenient way to manage access to shared resources.
  + It encapsulates shared variables, the procedures that operate on them, and the synchronization mechanisms.
  + Automatically handles mutual exclusion; when one thread is executing in the monitor, no other thread can enter until the first thread exits.
  + Can use condition variables for signaling and waiting, making it easier to manage complex synchronization scenarios.

# Semaphore:

* + A lower-level synchronization primitive used for controlling access to a finite number of resources.
  + Can be counting (allowing multiple threads to access resources up to a limit) or binary (0 or 1).
  + Threads can wait (decrement) and signal (increment) the semaphore, but it does not inherently manage shared data or provide mutual exclusion.
  + Requires careful programming to avoid race conditions and ensure mutual exclusion.

# Binary Semaphore:

* + A specific type of semaphore that can take only two values: 0 (locked) or 1 (unlocked), similar in behavior to a mutex.
  + Used for mutual exclusion and signaling between threads.
* Unlike a mutex, a binary semaphore can be signaled by any thread, not just the one that locked it.

1. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?

ANS- Deadlock in the Dining Philosophers problem can occur due to several scenarios:

# Circular Wait:

Each philosopher picks up their left fork (mutex or semaphore) and then waits for the right fork. For example:

* + - Philosopher 1 picks up Fork 1 and waits for Fork 2.
    - Philosopher 2 picks up Fork 2 and waits for Fork 3.
    - This continues in a circular fashion until Philosopher 5 picks up Fork 5 and waits for Fork 1, creating a cycle of dependencies.

# Hold and Wait:

Philosophers hold one fork while waiting for another. For instance, if all philosophers pick up their left fork simultaneously, they will all wait for their right fork, leading to deadlock.

# No Preemption:

Once a philosopher picks up a fork, they will not release it until they have finished eating. This condition means that if every philosopher is holding one fork and waiting for another, none can proceed, causing deadlock.

1. Which of the following can be used to avoid deadlock in the Dining Philosophers Problem?
2. Using a semaphore initialized to the number of philosophers.
3. Using a semaphore initialized to one less than the number of philosophers.
4. Using a mutex for each philosopher.
5. Using a monitor for each fork

# ANS- b. Using a semaphore initialized to one less than the number of philosophers.

**And d.Using a monitor for each fork**

1. Which synchronization construct encapsulates shared variables, synchronization primitives, and operations on shared variables?
2. Semaphore
3. Binary Semaphore
4. Monitor
5. Mutex

**Ans- Monitor**

**Date: Signature of faculty in-charge**