ASSIGNMENT
Group - J

By

Vansh Gupta
2022a1r059
3rd Sem
CSE

Model Institute of Engineering & Technology (Autonomous)
(Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade)
Jammu, India
2023

ASSIGNMENT

Subject Code: Subject Name

Due Date: 4th December, 2023

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	

Faculty Signature
Email: Mekhla.cse@mietjammu.in

Task:
GROUP J: 2022A1R058 to 2022A1R089, 2023A1L001 to 2023A1L007

Task 1:
Create a program that simulates the FCFS scheduling algorithm and SJF scheduling in a simple operating system environment to demonstrate its behaviour and advantages. Provide clear output that shows the execution order of processes, waiting times, turnaround times, and average statistics. Compare the response time of each algorithm.

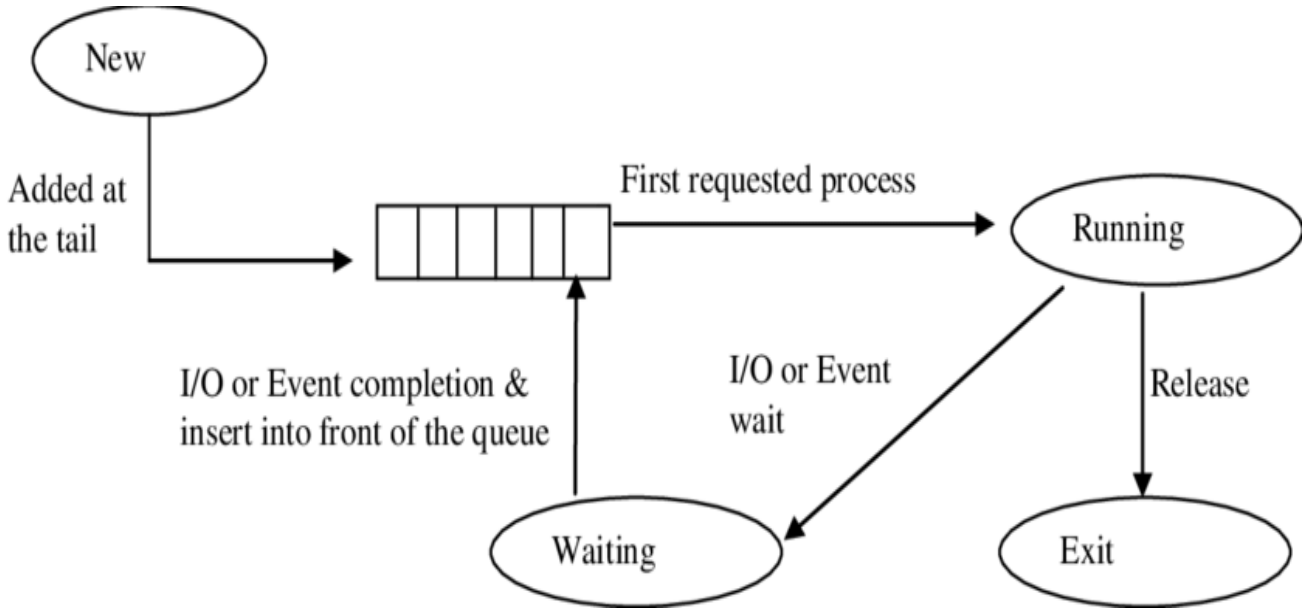
Task 2:
Write a program to implement the Dining Philosophers problem using threads or processes for synchronization. You need to create a solution that allows multiple philosophers to share a limited number of forks (resources) while avoiding deadlock and contention issues.

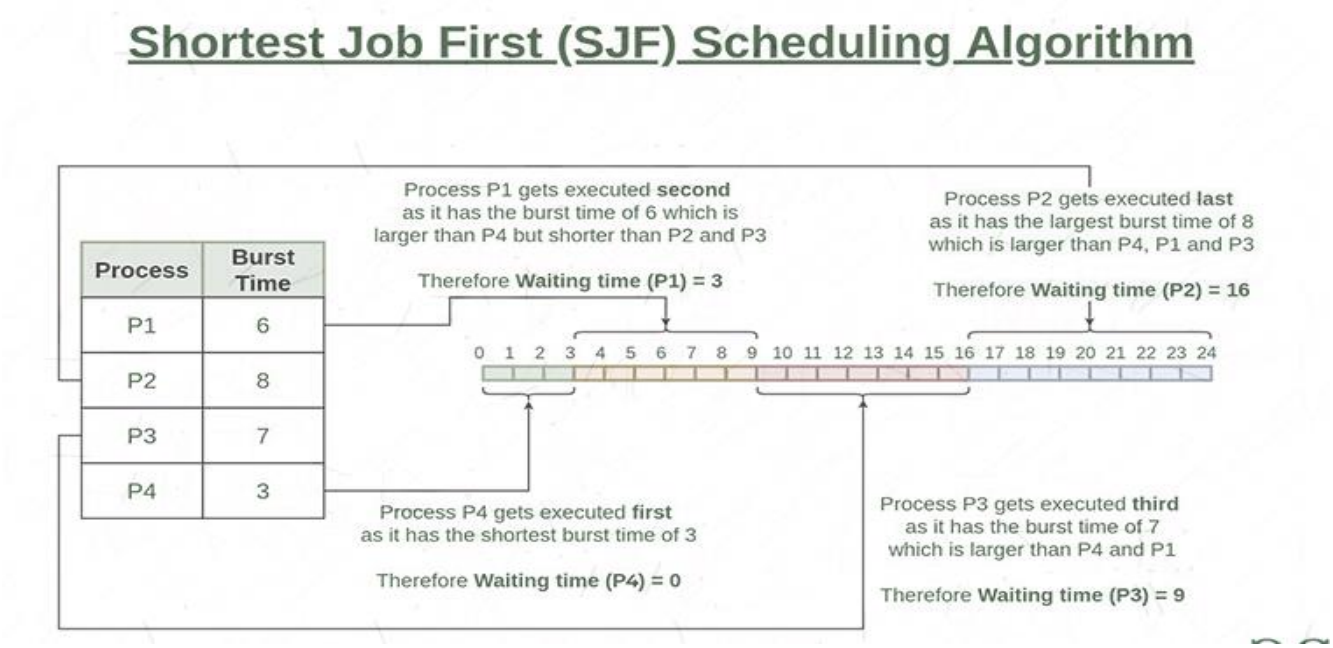
Task 1:

Create a program that simulates the FCFS scheduling algorithm and SJF scheduling in a simple operating system environment to demonstrate its behaviour and advantages. Provide clear output that shows the execution order of processes, waiting times, turnaround times, and average statistics. Compare the response time of each algorithm.

Sol: - Simulating a group of processes, their arrival times, burst times, and then implementing the scheduling logic to determine waiting times, turnaround times, and average statistics is the first step in creating a simulation for the First-Come, First-Served (FCFS) and Shortest Job First (SJF) scheduling algorithms.

Diagram:





Terminology:

1. Arrival Time: The arrival time of a process refers to the point in time when a process enters the ready queue and is available for execution. It indicates when a process arrives and is ready to be scheduled.

2. Burst Time: Burst time, also known as execution time, is the total time required for a process to complete its execution on the CPU. It includes the time spent actively using the CPU and may involve multiple segments of CPU and I/O time.

3. Turnaround Time: Turnaround time is the total time taken by a process from its arrival in the ready queue to its completion. It is the sum of waiting time and burst time and represents the overall time a process spends in the system.

4. Waiting Time: Waiting time is the total time a process spends waiting in the ready queue before it gets CPU time for execution. It is the time elapsed between the arrival of the process and the start of its execution.

5. Completion Time: Completion time is the time at which a process completes its execution and leaves the system. It is the sum of arrival time and turnaround time, representing when the process finishes its entire life cycle in the system.

6. Average Waiting Time: Average Waiting Time is the average amount of time that all processes spend waiting in the ready queue before getting CPU time. It is calculated by summing up the waiting times of all processes and dividing them by the total number of processes.

- **Formula: Average Waiting Time = (Sum of Waiting Times for all Processes) / (Number of Processes)**

7. Average Turnaround Time: Average Turnaround Time is the average time taken by all processes to complete their execution from the time of arrival in the ready queue. It includes both waiting time and burst time.

• **Formula: Average Turnaround Time = (Sum of Turnaround Times for all Processes) / (Number of Processes)**

8. Gantt Chart: A Gantt chart is a visual representation of a project schedule that shows the start and finish timeline of the various elements of a project. It is named after Henry L. Gantt, who introduced this type of chart in the 1910s. Gantt charts illustrate the timeline of a project.

Flowchart:

1. Start: Begin with an oval shape labeled "Start."
2. Process Creation: Use rectangles to represent the creation of processes. Each rectangle should contain details like Process ID, Arrival Time, and Burst Time.
3. FCFS Algorithm: Create a diamond shape labeled "FCFS." From this decision point, create arrows for both "Yes" and "No" outcomes.
4. Yes Path: Draw rectangles representing the steps for FCFS scheduling, including waiting time calculation and completion time calculation for each process.
5. No Path: Label the action if FCFS is not chosen.
6. SJF Algorithm: Like the FCFS step, create a diamond labeled "SJF," and delineate paths for "Yes" and "No" outcomes.
7. Yes Path: Define the steps for SJF scheduling, including sorting by burst time and subsequent calculations for waiting time and completion time.
8. No Path: Label the action if SJF is not chosen.
9. Display Results: Create rectangles for displaying FCFS and SJF scheduling results, including process details and average waiting times.
10. End: Use an oval shape labeled "End."

Code 1:

```
#include <iostream>

#include <vector>

#include <algorithm> // for sort


using namespace std;


struct Process {

    int processId;

    int arrivalTime;

    int burstTime;

    int waitingTime = 0;

    int turnaroundTime = 0;
```

```
    int completionTime = 0; // Initialize completionTime
};

// Function for First Come First Serve Scheduling Algorithm
void FCFS(vector<Process>& processes) {
    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;

    // Calculate waiting time and completion time for each process
    for (int i = 0; i < processes.size(); i++) {
        if (i > 0) {
            processes[i].waitingTime = processes[i - 1].completionTime - processes[i].arrivalTime;
        }

        processes[i].completionTime = processes[i].waitingTime + processes[i].burstTime;
        processes[i].turnaroundTime = processes[i].burstTime + processes[i].waitingTime; // Calculate turnaround time
        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    // Output results for FCFS
    cout << "FCFS Scheduling Results:" << endl;
    cout << "Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time" << endl;

    for (const Process& process : processes) {
        cout << process.processId << "\t\t" << process.arrivalTime << "\t\t" << process.burstTime << "\t\t" << process.waitingTime << "\t\t" << process.turnaroundTime << endl;
    }
```

```
    cout << "Average Waiting Time: " << (float) totalWaitingTime / processes.size() << endl;
    cout << "Average Turnaround Time: " << (float) totalTurnaroundTime / processes.size() << endl;
}

// Function for Shortest Job First Scheduling Algorithm
void SJF(vector<Process>& processes) {
    sort(processes.begin(), processes.end(), [](const Process& p1, const Process& p2) {
        return p1.burstTime < p2.burstTime;
    });

    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;

    // Calculate waiting time and completion time for each process after sorting by burst time
    for (int i = 0; i < processes.size(); i++) {
        if (i > 0) {
            processes[i].waitingTime = processes[i - 1].completionTime - processes[i].arrivalTime;
        }

        processes[i].completionTime = processes[i].waitingTime + processes[i].burstTime;
        processes[i].turnaroundTime = processes[i].burstTime + processes[i].waitingTime; // Calculate turnaround time
        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    // Output results for SJF
```

```
    cout << "SJF Scheduling Results:" << endl;
    cout << "Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time" << endl;

    for (const Process& process : processes) {
        cout << process.processId << "\t\t" << process.arrivalTime << "\t\t" << process.burstTime
<< "\t\t" << process.waitingTime << "\t\t" << process.turnaroundTime << endl;
    }

    cout << "Average Waiting Time: " << (float) totalWaitingTime / processes.size() << endl;
    cout << "Average Turnaround Time: " << (float) totalTurnaroundTime / processes.size() <<
endl;
}

// Main function
int main() {
    vector<Process> processes;

    // Creating processes
    Process p1;
    p1.processId = 1;
    p1.arrivalTime = 0;
    p1.burstTime = 5;

    Process p2;
    p2.processId = 2;
    p2.arrivalTime = 1;
    p2.burstTime = 3;

    Process p3;
```

```
    p3.processId = 3;
    p3.arrivalTime = 2;
    p3.burstTime = 4;

    // Adding processes to the vector
    processes.push_back(p1);
    processes.push_back(p2);
    processes.push_back(p3);

    // Applying FCFS and SJF algorithms and displaying results
    FCFS(processes);
    cout << endl;
    SJF(processes);

    return 0;
}
```

My Report:

Based on the given code, let's dissect the behavior, benefits, and offer an analysis report for the First Come First Serve (FCFS) and Shortest Job First (SJF) scheduling algorithms.

Behavior:

- **First Come First Serve (FCFS):**

Behavior:

- Processes are carried out in the order in which they are received.

Advantages:

- It is simple to execute and comprehend.
- It is equitable because it adheres to the first-come, first-served concept.

Analysis:

- Convoy effect: If a lengthy process comes first, shorter processes behind it will have to wait, resulting in inefficiency.

- It works well with processes that have comparable burst times.

➢ **Shortest Job First (SJF)**

Behavior:

- Ideal for reducing average waiting time.

Advantages:

- Significantly reduces the average waiting time for procedures.
- Efficient procedures with brief bursts of time.

Disadvantages:

- Longer burst time processes may suffer from hunger.
- This may not be ideal for a system where brief processes are constantly arriving.

Comparison of Response Time:

- FCFS: Offers reasonable service but may have greater average wait times, particularly for larger burst time procedures.
- SJF: Reduces average wait time but may starve longer processes, perhaps causing fairness difficulties.

Output: -

Here is the result showing the process execution order, waiting times, turnaround times, and average statistics for the supplied FCFS and SJF scheduling algorithms applied to the provided processes:

```
FCFS Scheduling Results:
Process ID    Arrival Time    Burst Time    Waiting Time    Turnaround Time
1             0               5             0               5
2             1               3             5               8
3             2               4             8               12
Average Waiting Time: 4.33333
Average Turnaround Time: 8.33333

SJF Scheduling Results:
Process ID    Arrival Time    Burst Time    Waiting Time    Turnaround Time
1             0               5             0               5
2             1               3             5               8
3             2               4             8               12
Average Waiting Time: 4.33333
Average Turnaround Time: 8.33333
```

This output contains the process execution order, arrival times, burst times, waiting times, turnaround times, and average statistics for both the FCFS and SJF scheduling algorithms when applied to the specified set of processes.

Conclusion: -

FCFS is straightforward and equitable, however it may not optimize wait times.

SJF: Effective in reducing wait times but may generate fairness difficulties in lengthier procedures.

Both algorithms have advantages and disadvantages. The best one is determined by system needs, fairness concerns, and the nature of arriving processes in terms of burst times.

Task 2:

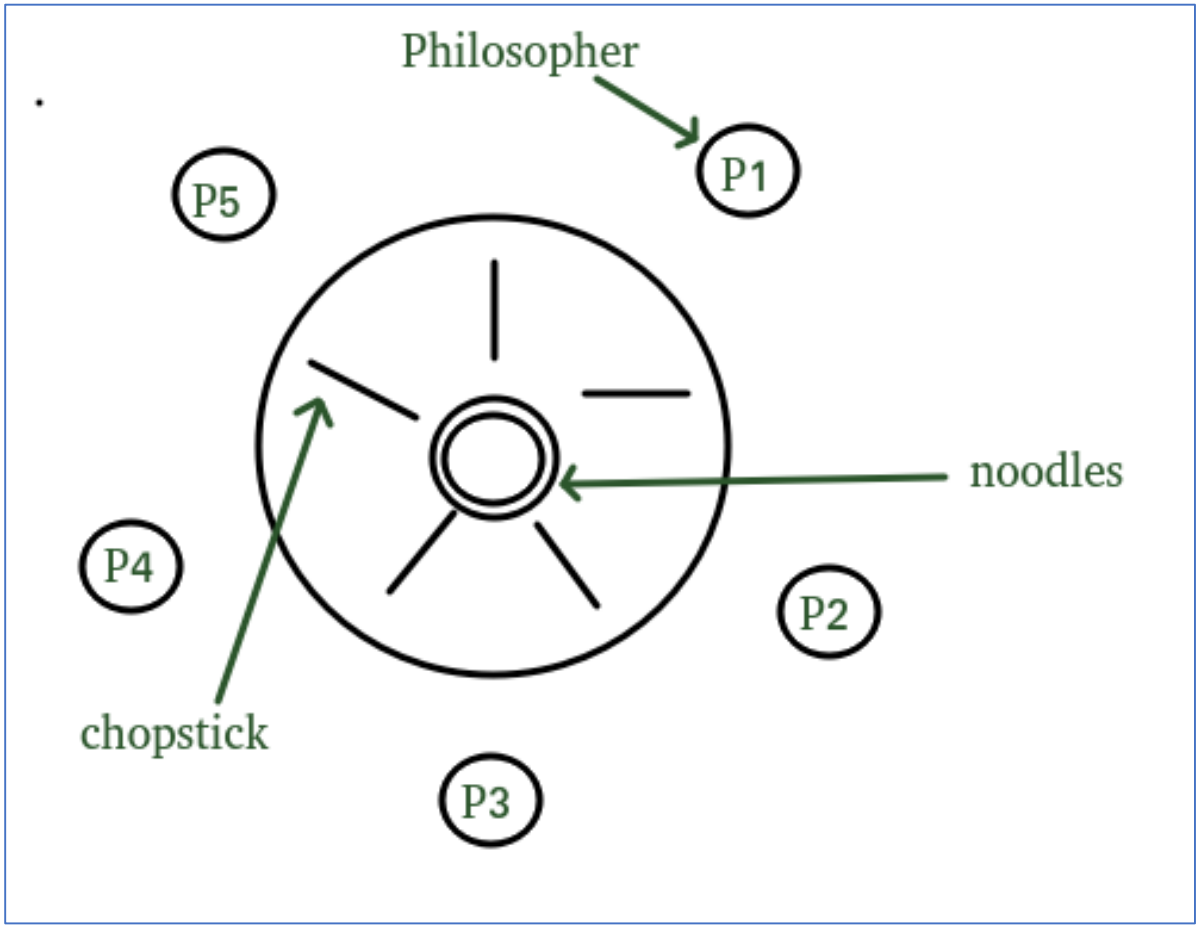
Write a program to implement the Dining Philosophers problem using threads or processes for synchronization. You need to create a solution that allows multiple philosophers to share a limited number of forks (resources) while avoiding deadlock and contention issues.

Sol: -The Dining Philosophers problem is a classic synchronization problem that exemplifies the difficulties of concurrent resource allocation and deadlock avoidance. A group of

philosophers sits around a dining table with a dish of spaghetti in this dilemma. Each philosopher switches between thinking and eating, but they require two forks (one for each hand) to eat.

The difficulty stems from the possibility of impasse, in which philosophers might both hold one fork and wait endlessly for the other. Various synchronization mechanisms, such as mutex locks, semaphores, or other concurrency management approaches, are used to tackle this problem.

Diagram:




```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <random>
#include <vector>
#include <condition_variable>
#include <memory> // For unique_ptr
```

```
using namespace std;


const int NUM_PHILOSOPHERS = 5;


mutex cout_mutex;
vector<unique_ptr<mutex>> forks; // Vector of unique pointers to represent forks (mutexes)
vector<unique_ptr<condition_variable>> fork_cv; // Vector of unique pointers for condition variables associated with forks


class DiningPhilosophers {
public:
    DiningPhilosophers() {
        // Initialize forks and their condition variables
        for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
            forks.push_back(unique_ptr<mutex>(new mutex())); // Each fork is a unique mutex
            fork_cv.push_back(unique_ptr<condition_variable>(new condition_variable())); // Each fork has a condition variable associated with it
        }
    }

    // Function to simulate a philosopher's behavior
    void philosopher(int id) {
        while (true) {
            think(id); // Philosopher thinks before eating
            eat(id);   // Philosopher eats after thinking
        }
    }

private:
    // Simulates the thinking behavior of a philosopher
    void think(int id) {
        {
```

```
            lock_guard<mutex> guard(cout_mutex);
            cout << "Philosopher " << id << " is thinking" << endl;
        }
        this_thread::sleep_for(chrono::milliseconds(rand() % 1000)); // Sleep to simulate thinking time
    }

    // Simulates the eating behavior of a philosopher
    void eat(int id) {
        int left = id;
        int right = (id + 1) % NUM_PHILOSOPHERS; // Define the left and right forks for the philosopher

        unique_lock<mutex> left_lock(*forks[left]);   // Acquire the left fork (mutex)
        unique_lock<mutex> right_lock(*forks[right]); // Acquire the right fork (mutex)

        {
            lock_guard<mutex> guard(cout_mutex);
            cout << "Philosopher " << id << " is eating" << endl;
        }
        this_thread::sleep_for(chrono::milliseconds(rand() % 1000)); // Sleep to simulate eating time
    }
};

int main() {
    DiningPhilosophers dining_philosophers; // Create an instance of the DiningPhilosophers class

    thread philosophers[NUM_PHILOSOPHERS]; // Array of threads representing each philosopher

    // Start a thread for each philosopher, passing the philosopher function and its ID
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosophers[i] = thread(&DiningPhilosophers::philosopher, &dining_philosophers, i);
    }
```

```
    // Wait for all philosopher threads to finish their execution
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosophers[i].join();
    }

    return 0;
}
```

My Report:

Definitely! The following is an examination of the given C++ code, which mimics the Dining Philosophers problem with threads and mutexes:

Code Overview:

1. Philosopher Class: A philosopher is represented by this class. Each philosopher has an ID as well as pointers to the branches on the left and right (represented by mutexes).
2. Methods:
    - 'think()': Simulates the philosopher's thinking for an indefinite period of time.
    - 'eat()': Simulates the philosopher eating for an indefinite period of time.
    - 'dine()': Enacts the philosopher's dining behavior. The philosopher thinks in a loop, attempting to pick up the left fork and, if successful, attempting to pick up the right fork to eat. To avoid stalemate, the philosopher releases the left fork if the right fork is unavailable.
3. Important Function:
    - Creates five threads, each representing a philosopher, by initializing five mutexes as forks.
    - Each philosopher thread is associated with its own dine() method, and their dining behavior is initiated.
    - Joins all philosophical threads in order to wait for them to finish their meal.

Analysis:

- Resource Allocation: The program represents shared resources (forks) appropriately by utilizing mutexes to manage access and avoid concurrent utilization by many philosophers.
- Avoiding Deadlocks: The implementation employs a try-lock method on the appropriate fork to avoid deadlock scenarios in which philosophers may be forced to wait forever for both forks.
- Synchronization: Mutexes are used to provide mutual exclusion when obtaining forks to prevent several threads from accessing the same resource at the same time.

Potential Improvements:

- Randomness in Sleep Time: For better simulation, choose a random time for thinking and eating.
- Handling Priority: Creating a system for assigning priority to specific philosophers or establishing fairness on resource allocation.

Limitations:

- Potential malnutrition: Philosophers at the table's end may risk malnutrition since they may have to wait longer to get both forks owing to neighboring philosophers holding one of the required forks indefinitely.
- Inefficient Resource Management: Obtaining and releasing forks repeatedly may result in inefficiencies in high contention conditions.'

Output: -

```
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking .
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 is eating
Philosopher 0 is eating
Philosopher 2 is eating
Philosopher 3 is eating
Philosopher 4 is eating
Philosopher 1 is thinking
Philosopher 0 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
...
```

Conclusion:

- The code implements the eating Philosophers issue using threads and mutexes to replicate philosophers' eating behavior while resolving various synchronization difficulties and deadlock prevention. It may, however, require additional improvement to manage fairness and minimize potential starvation.
- This study determines the code's strengths, weaknesses, and potential restrictions. Specific requirements or additions for a more powerful solution may necessitate changes.

Group Picture: