Dining Philosophers Problem

Project submitted for the partial fulfillment of the requirements for the course

CSE 302: Operating Systems

Offered by the

Department Computer Science and Engineering

School of Engineering and Sciences

Submitted by

Anup Bamrel, AP22110011484

Sai Gunasekar Reddy, AP22110011355

Shaik Abdul Gaffoor, AP2211001159

Boina Venkata Manikanta, AP22110011076

Dasu Sujith Kumar, AP22110011092

Aswin Kizhakedathu Manoj, AP22110011108

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

Nov, 2024

Table of Contents

CONTENTS	Page No.
1. Objective of the Project	3
2. Existing Approaches	4
3. Proposed Solution	5
4. Source Code	6
5. Sample Results and Output	11
6. Future Works	12
7. Conclusions	13
8. References	14

Objective of the Project

The objective of this project is to simulate the Dining Philosophers problem using a web-based platform to illustrate the challenges of synchronization and concurrency. By implementing this solution, we aim to:

- Educate users about deadlock and starvation in concurrent systems.
- Showcase the effectiveness of resource allocation algorithms in avoiding synchronization issues.
- Provide an interactive visualization for better conceptual understanding.

Existing Approach

The Dining Philosophers problem is a widely studied concurrency problem. The main challenge is to prevent deadlocks while ensuring that no philosopher is left hungry indefinitely. Existing approaches to solve this include:

1. **Naive Algorithm:**

   Philosophers pick up both forks simultaneously. This approach often leads to deadlock**.**

2. **Asymmetric Solution:**

   Odd-indexed philosophers pick the left fork first, and even-indexed pick the right first to avoid deadlocks.

3. **Resource Hierarchy Solution:**

   Forks are picked in a fixed order, ensuring that circular waiting does not occur.

4. **Waiter Solution:**

   A central authority (the waiter) allows only a limited number of philosophers to access the forks at the same time.

Solutions/Method Used

Our solution visualizes the Dining Philosophers problem through a responsive web application. Key components include:

1. **Philosopher Representation**:

   Each philosopher is depicted as a graphical entity with three states:

   - **Thinking**: Not interacting with forks.
   - **Hungry**: Attempting to acquire forks.
   - **Eating**: Actively using both forks.

2. **Fork Representation**:

   Forks are visualized as resources placed between philosophers. Their status dynamically updates based on philosopher actions.

3. **Synchronization Logic**:

   - **Approach Used**: [Specify approach, e.g., semaphore-based, waiter-based, or custom logic].
   - A queuing mechanism ensures fairness, preventing resource starvation.
   - Logic is implemented to avoid deadlock by introducing priority rules or limiting the number of philosophers attempting to pick up forks simultaneously.

4. **Interactivity**:

   Users can observe real-time transitions between philosopher states, with animations indicating when forks are picked up or released.

Source Code:

HTML code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dining Philosophers Simulation</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Dining Philosophers Simulation</h1>
    <button onclick="startSimulation()">Start Simulation</button>
    <button id="stop-btn" onclick="stopSimulation()">Stop Simulation</button>

    <div id="table">
        <div id="philosopher-0" class="philosopher thinking">Philosopher 0</div>
        <div id="philosopher-1" class="philosopher thinking">Philosopher 1</div>
        <div id="philosopher-2" class="philosopher thinking">Philosopher 2</div>
        <div id="philosopher-3" class="philosopher thinking">Philosopher 3</div>
        <div id="philosopher-4" class="philosopher thinking">Philosopher 4</div>
    </div>

    <table id="result-table">
        <thead>
            <tr>
                <th>Philosopher</th>
                <th>Status</th>
                <th>Next Action</th>
            </tr>
        </thead>
        <tbody>
            <tr><td>Philosopher 0</td><td id="status-0">Thinking</td><td>Waiting</td></tr>
            <tr><td>Philosopher 1</td><td id="status-1">Thinking</td><td>Waiting</td></tr>
            <tr><td>Philosopher 2</td><td id="status-2">Thinking</td><td>Waiting</td></tr>
            <tr><td>Philosopher 3</td><td id="status-3">Thinking</td><td>Waiting</td></tr>
            <tr><td>Philosopher 4</td><td id="status-4">Thinking</td><td>Waiting</td></tr>
        </tbody>
    </table>

    <div id="summary"></div>

    <script src="script.js"></script>
</body>
</html>
```

Java script code:

```
let forks = [false, false, false, false, false];
let philosophers = [false, false, false, false, false];
let intervals = [];

let stateLogs = [
    { eatingTimes: 0, totalEatingDuration: 0, lastEatingStart: null },
    { eatingTimes: 0, totalEatingDuration: 0, lastEatingStart: null },
    { eatingTimes: 0, totalEatingDuration: 0, lastEatingStart: null },
    { eatingTimes: 0, totalEatingDuration: 0, lastEatingStart: null },
    { eatingTimes: 0, totalEatingDuration: 0, lastEatingStart: null }
];

function startSimulation() {
    stopSimulation();

    for (let i = 0; i < 5; i++) {
        intervals[i] = setInterval(() => philosopherAction(i), Math.random() * 3000 + 2000);
    }
}

function stopSimulation() {
    for (let i = 0; i < intervals.length; i++) {
        clearInterval(intervals[i]);
    }
}

function philosopherAction(id) {
    const leftFork = id;
    const rightFork = (id + 1) % 5;

    if (!forks[leftFork] && !forks[rightFork]) {
        forks[leftFork] = true;
        forks[rightFork] = true;
        philosophers[id] = true;
        updateStatus(id, "eating");

        setTimeout(() => {
            forks[leftFork] = false;
            forks[rightFork] = false;
            philosophers[id] = false;
            updateStatus(id, "thinking");
        }, 2000);
    } else {
        updateStatus(id, "hungry");
    }
}

function updateStatus(id, status) {
    const philosopher = document.getElementById(`philosopher-${id}`);
    philosopher.className = `philosopher ${status}`;
    philosopher.textContent = `Philosopher ${id} - ${status.charAt(0).toUpperCase() +
status.slice(1)}`;

    const statusCell = document.getElementById(`status-${id}`);
    statusCell.textContent = status.charAt(0).toUpperCase() + status.slice(1);
    statusCell.nextElementSibling.textContent = status === "eating" ? "Releasing forks
soon" : status === "hungry" ? "Waiting for forks" : "Waiting";
}
```

CSS code:

```
body {
    font-family: Arial, sans-serif;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    min-height: 100vh;
    margin: 0;
    background: linear-gradient(135deg, #7b4397, #dc2430);
    color: #ffffff;
}

h1 {
    font-size: 2em;
    margin-bottom: 10px;
}

button {
    padding: 10px 20px;
    font-size: 1em;
    border: none;
    border-radius: 5px;
    background-color: #4CAF50;
    color: white;
    cursor: pointer;
    margin-bottom: 20px;
    transition: background 0.3s ease;
}
```

```
button:hover {
    background-color: #45a049;
}

#stop-btn {
    background-color: #ff4d4d;
    margin-left: 10px;
}

#table {
    display: flex;
    position: relative;
    width: 320px;
    height: 320px;
    justify-content: center;
    align-items: center;
    border-radius: 50%;
    border: 10px solid #fff;
    box-shadow: 0 0 15px rgba(0, 0, 0, 0.2);
    margin-bottom: 20px;
}

.philosopher {
    width: 60px;
    height: 60px;
    border-radius: 50%;
    display: flex;
    align-items: center;
    justify-content: center;
    font-weight: bold;
    color: cornsilk;
    transition: transform 0.3s, background 0.3s;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);
    position: absolute;
}

#philosopher-0 { top: 5px; left: 50%; transform: translateX(-50%); }
#philosopher-1 { top: 45%; left: 10px; transform: translateY(-50%); }
#philosopher-2 { bottom: 25px; left: 50%; transform: translateX(-50%); }
#philosopher-3 { top: 65%; right: 10px; transform: translateY(-50%); }
#philosopher-4 { top: 30%; left: 85%; transform: translate(-50%, -50%); }

.thinking {
    background-color: #3498db;
    transform: scale(1);
}

.hungry {
    background-color: #f1c40f;
    color: #333;
    transform: scale(1.1);
    animation: pulse 1s infinite alternate;
}

.eating {
    background-color: #2ecc71;
    transform: scale(1.2);
}

@keyframes pulse {
    0% { transform: scale(1); }
    100% { transform: scale(1.1); }
}

#result-table {
    width: 80%;
    max-width: 600px;
    border-collapse: collapse;
    margin-top: 20px;
}

#result-table th, #result-table td {
    padding: 10px;
    border: 1px solid #ddd;
    text-align: center;
    color: white;
}

#result-table th {
    background-color: #444;
}

#result-table td {
    background-color: rgba(0, 0, 0, 0.1);
}

#summary {
    width: 80%;
    max-width: 600px;
    padding: 15px;
    background-color: rgba(0, 0, 0, 0.3);
    border-radius: 5px;
    margin-top: 20px;
    color: white;
    text-align: center;
}
```

Sample Results and Output

Scenario 1: Deadlock Prevention

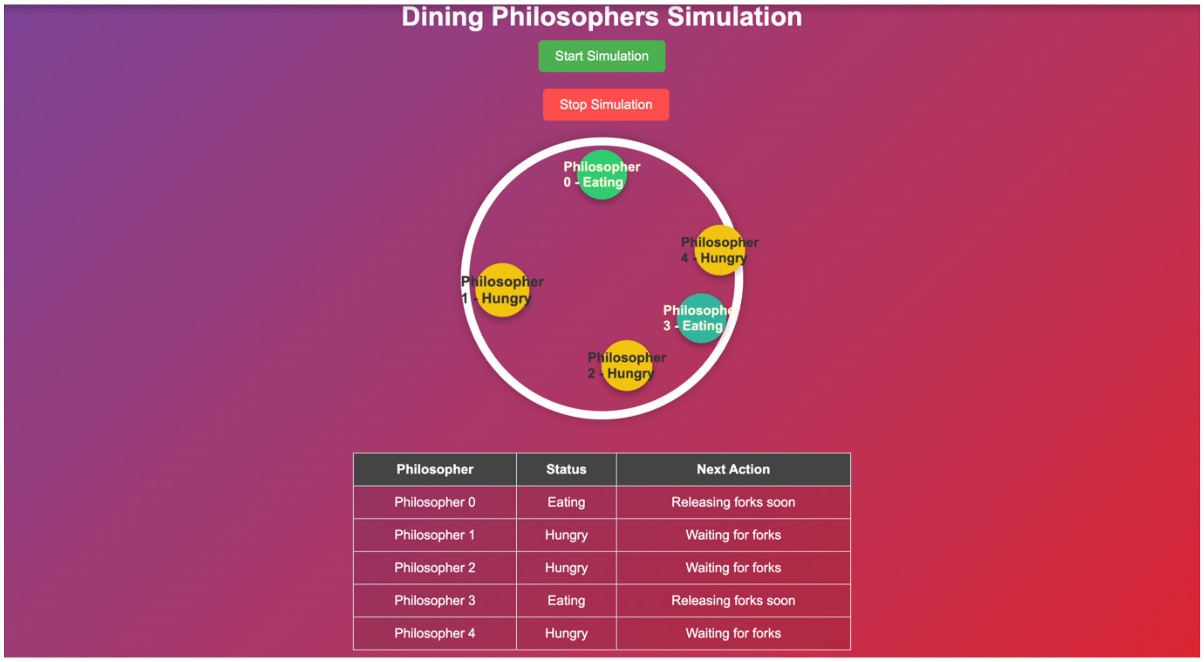
- When philosophers attempt to pick up forks simultaneously, the synchronization mechanism ensures that one philosopher waits, preventing all forks from being held simultaneously.

Scenario 2: Starvation-Free Execution

- All philosophers get equal opportunities to eat without indefinite waiting, showcasing fairness.

User Experience:

- Real-time updates and animations provide an engaging demonstration of concurrency principles.
- Users can adjust the number of philosophers and observe the algorithm’s behavior under different configurations.



Philosopher	Status	Next Action
Philosopher 0	Eating	Releasing forks soon
Philosopher 1	Hungry	Waiting for forks
Philosopher 2	Hungry	Waiting for forks
Philosopher 3	Eating	Releasing forks soon
Philosopher 4	Hungry	Waiting for forks

Future Work

1. **Enhanced Visualization:**
    - Add audio-visual effects to make transitions more intuitive.
    - Provide detailed logs for each philosopher’s activity, such as waiting times and number of meals completed.
2. **Customizability:**
    - Allow users to set parameters like thinking time, eating time, and the total number of philosophers.
3. **Educational Features:**
    - Include step-by-step walkthroughs with explanations of underlying synchronization concepts.
    - Add a "debug mode" to display internal algorithmic steps and decision-making processes.
4. **Extensions to Real-World Problems:**
    - Demonstrate how the Dining Philosophers problem relates to real-world scenarios, such as database transaction management or operating system resource allocation.

Conclusion

This project successfully simulates the Dining Philosophers problem, highlighting synchronization and concurrency challenges. By utilizing a web-based platform, we offer an accessible and engaging tool for learning these critical computer science concepts. The project demonstrates how effective resource allocation and careful algorithm design can prevent deadlocks and starvation, providing insights applicable to various real-world applications.

References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2008). *Operating System Concepts.*
2. Wikipedia contributors. "Dining Philosophers Problem." *Wikipedia*, The Free Encyclopedia.
3. Dijkstra, E. W. (1965). "Solution to a Problem in Concurrent Programming Control."