East West University

Department of Computer Science and Engineering

Lab Experiment: Deadlock: Understanding and Avoidance in Operating Systems

Objective:

The objective of this lab is to understand and implement solutions to avoid deadlock using C programming, semaphores, and threads. The Dining Philosophers Problem will serve as an illustrative model for exploring the challenges of resource allocation and synchronization in Operating Systems.

Equipment and Tools:

- A computer system with Linux or any other POSIX-compliant operating system.
- GCC compiler to compile C code.
- Text editor (e.g., VS Code, Vim, or Nano).
- pthread library for thread creation.
- Semaphore library (POSIX semaphores).

Pre-Lab Instructions:

1. Revise basic concepts of Operating Systems, especially synchronization, semaphores, and deadlock.
2. Install necessary development tools on your system (e.g., GCC compiler and pthread library).
3. Read about the Dining Philosophers Problem and its relevance in resource allocation.

Introduction:

Deadlock is a critical issue in Operating Systems where two or more processes are unable to proceed because each is waiting for a resource held by the other. This condition leads to a system halt, as none of the involved processes can make progress. Deadlocks commonly occur in resource-sharing scenarios such as memory, files, or peripherals.

The Dining Philosophers Problem is a classic synchronization problem that demonstrates the occurrence of deadlock in resource allocation. In this scenario, five philosophers sit around a circular table, alternating between thinking and eating. Each philosopher requires two chopsticks (shared resources) to eat. Deadlock can occur if every philosopher picks up one chopstick and waits indefinitely for the second, causing a circular wait condition. This problem serves as a model to understand and address deadlock in multi-process systems.

Theory:

Deadlock

Deadlock arises in a system when a set of processes are blocked because each process is holding a resource and waiting for another resource held by another process. Four conditions must hold simultaneously for deadlock to occur:

1. Mutual Exclusion: Only one process can use a resource at any time.
2. Hold and Wait: Processes holding resources can request additional resources held by others.
3. No Preemption: Resources cannot be forcibly taken from processes holding them.
4. Circular Wait: A circular chain of processes exists, where each process waits for a resource held by the next process in the chain.

Dining Philosophers Problem and Deadlock

The Dining Philosophers Problem illustrates the potential for deadlock in resource allocation:

- Problem Description:
    - There are 5 philosophers and 5 chopsticks.
    - Philosophers alternate between two states: thinking and eating.
    - To eat, a philosopher must simultaneously hold the left and right chopsticks.
- Deadlock Scenario:
    - If all philosophers pick up their left chopsticks simultaneously, no one can pick up their right chopstick, leading to circular waiting and deadlock.

Deadlock Avoidance Strategies

To avoid deadlock in the Dining Philosophers Problem, various strategies can be employed:

1. Semaphore-based Solution:
    - Represent chopsticks as semaphores and allow a philosopher to pick up chopsticks only when both are available.
2. Resource Hierarchy:
    - Impose an order on resource acquisition (e.g., always pick up the lower-numbered chopstick first).
3. Alternating Behavior:
    - Philosophers alternate which chopstick they pick up first (e.g., even-indexed philosophers pick the right chopstick first, and odd-indexed pick the left chopstick first).
4. Limit Resource Requests:
    - Restrict the number of philosophers allowed to pick up chopsticks simultaneously, ensuring at least one can eat.

These strategies help eliminate one or more of the necessary conditions for deadlock, enabling smooth resource allocation.

Lab Implementation:

C Code: Dining Philosophers Problem

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM_PHILOSOPHERS 5

sem_t chopsticks[NUM_PHILOSOPHERS];  // Semaphores to represent
chopsticks

void* philosopher(void* num) {

    int id = *(int *)num;  // Get philosopher's ID

    while (1) {

        printf("Philosopher %d is thinking.\n", id);

        sleep(1);  // Thinking (simulated by sleep)

        // Alternating behavior to avoid deadlock

        if (id % 2 == 0) {

            sem_wait(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);  //
Pick up right chopstick first for even //philosophers

            sem_wait(&chopsticks[id]);  // Pick up left chopstick

        } else {

            sem_wait(&chopsticks[id]);  // Pick up left chopstick
first for odd philosophers

            sem_wait(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);  //
Pick up right chopstick

        }

        printf("Philosopher %d is eating.\n", id);

        sleep(2);  // Eating (simulated by sleep)

        // Put down chopsticks

        sem_post(&chopsticks[id]);

        sem_post(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);
```

```
        printf("Philosopher %d is done eating.\n", id);
    }
}
int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int ids[NUM_PHILOSOPHERS];
    // Initialize semaphores (chopsticks)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }
    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }
    // Join philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }
    // Destroy semaphores
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&chopsticks[i]);
    }
    return 0;
}
```

Explanation:

1. Semaphores:
    - Each chopstick is represented as a semaphore initialized to 1, meaning it's available.
    - A philosopher can only pick up a chopstick if it's available (`sem_wait`).
    - Once a philosopher is done eating, they put the chopstick down (`sem_post`).

2. Philosopher Threads:
    - Each philosopher is a separate thread alternating between thinking and eating:
        - Thinking: The philosopher spends time thinking (`sleep(1)`).
        - Eating: The philosopher tries to pick up two chopsticks (one to their left and one to their right) before eating. Once done, they release the chopsticks.
3. Chopstick Indexing:
    - Since the philosophers are seated in a circle, each philosopher picks up their own chopstick and the one on their right.
    - The right chopstick is accessed with `(id + 1) % NUM_PHILOSOPHERS` to handle the circular nature.
4. Deadlock Avoidance:
    - Alternating which chopstick to pick up first (based on philosopher index) reduces the chance of deadlock by breaking circular wait.

Example Output:

When you run the program, the philosophers will alternately think and eat. The output might look like this:

Philosopher 0 is thinking.

Philosopher 1 is thinking.

Philosopher 2 is thinking.

Philosopher 3 is thinking.

Philosopher 4 is thinking.

Philosopher 0 is eating.

Philosopher 0 is done eating.

Philosopher 4 is eating.

Philosopher 4 is done eating.

Philosopher 2 is eating.

Philosopher 2 is done eating.

…..

Enhancing Deadlock Avoidance:

To enhance deadlock avoidance, we can modify the behavior so that philosophers alternate which chopstick they pick up first, based on their index. Here's the modification:

```
if (id % 2 == 0) {
    sem_wait(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);  // Pick up right chopstick first for even philosophers
    sem_wait(&chopsticks[id]);  // Pick up left chopstick
} else {
    sem_wait(&chopsticks[id]);  // Pick up left chopstick first for odd philosophers
    sem_wait(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);  // Pick up right chopstick
}
```

This alternating behavior reduces the chance of all philosophers simultaneously picking up the same chopstick first, helping to prevent deadlock. By alternating the order of chopstick acquisition based on the philosopher's index, the circular wait condition is minimized, significantly enhancing the robustness of the solution.

Conclusion:

The Dining Philosophers Problem serves as an excellent model for understanding the complexities of resource allocation, synchronization, and deadlock in multi-process systems. By leveraging semaphores and implementing alternating resource acquisition strategies, we effectively address the potential for deadlock while maintaining the philosophers' ability to think and eat. This solution highlights the importance of systematic resource management in Operating Systems, ensuring efficiency and preventing system halts due to deadlock. Through this lab, students gain practical insights into real-world synchronization issues and the techniques to resolve them effectively.