**Project 4: Deadlock**

Duc Vo, Ty Gehrke

Grand Canyon University

CST-315: Operating Systems Lecture and Lab

Professor Citro

March 3rd, 2024

# Team Members

- Ty Gehrke

- Duc Vo

# Project Overview

This project simulates scenarios involving resource contention and the utilization of a timer to address deadlock and starvation situations among multiple processes. The implemented code consists of three distinct phases

## Deadlock Scenario Simulation (`DeadlockCase1.c`)

- Objective: Simulate a deadlock scenario where two resources are contended by two threads, leading to a circular wait.

- Code Overview: Uses two mutex locks (`lock1` and `lock2`) to protect two resources. Threads (`resource1` and `resource2`) acquire locks on these resources in a way that induces a deadlock.

## Resource Contention with Timer (`DeadlockCase2.c`)

- Objective: Simulate a scenario where multiple processes compete for access to a shared resource and a timer is used to handle starvation situations.

- Code Overview: Uses a single mutex (`resourceMutex`) to protect a shared resource. Processes, represented by threads (`process`), attempt to access the resource. A timer mechanism handles situations where a process is starved, and it restarts after a predefined time interval.

## Combined Solution (`DeadlockCase3.c`):

- Objective: Combine the deadlock scenario simulation and the resource contention with a timer to provide a comprehensive scenario.

- Code Overview: Merges the concepts of deadlock simulation and resource contention with timer handling into a single program.

# Scenario

In this scenario, the program addresses concurrent programming challenges where multiple processes contend for access to a shared resource. The shared resource is safeguarded by a mutex to prevent data race conditions, ensuring exclusive access for one thread at a time. The overarching goal is to manage resource access effectively, mitigating potential deadlocks and handling situations where a process might experience starvation due to the unavailability of the shared resource.

# Approach to Implementation (‘DeadlockCase3.c’)

Implemented in C using the thread library, the program begins by initializing a mutex to control access to the shared resource. Multiple threads, each representing a process, are created to simulate resource contention.

Within each thread, a logic block checks for resource availability. If the resource is available, the thread acquires the mutex, accesses the resource, and releases the mutex. In cases where the resource is unavailable, a timer mechanism handles starvation scenarios, allowing controlled restarts after a predefined waiting period.

Synchronization is ensured by waiting for all threads to complete before proceeding, providing a realistic simulation for analyzing potential deadlocks and thread starvation. The program's comprehensive approach integrates mutex synchronization and a timer mechanism, contributing to a robust solution for concurrent resource management.

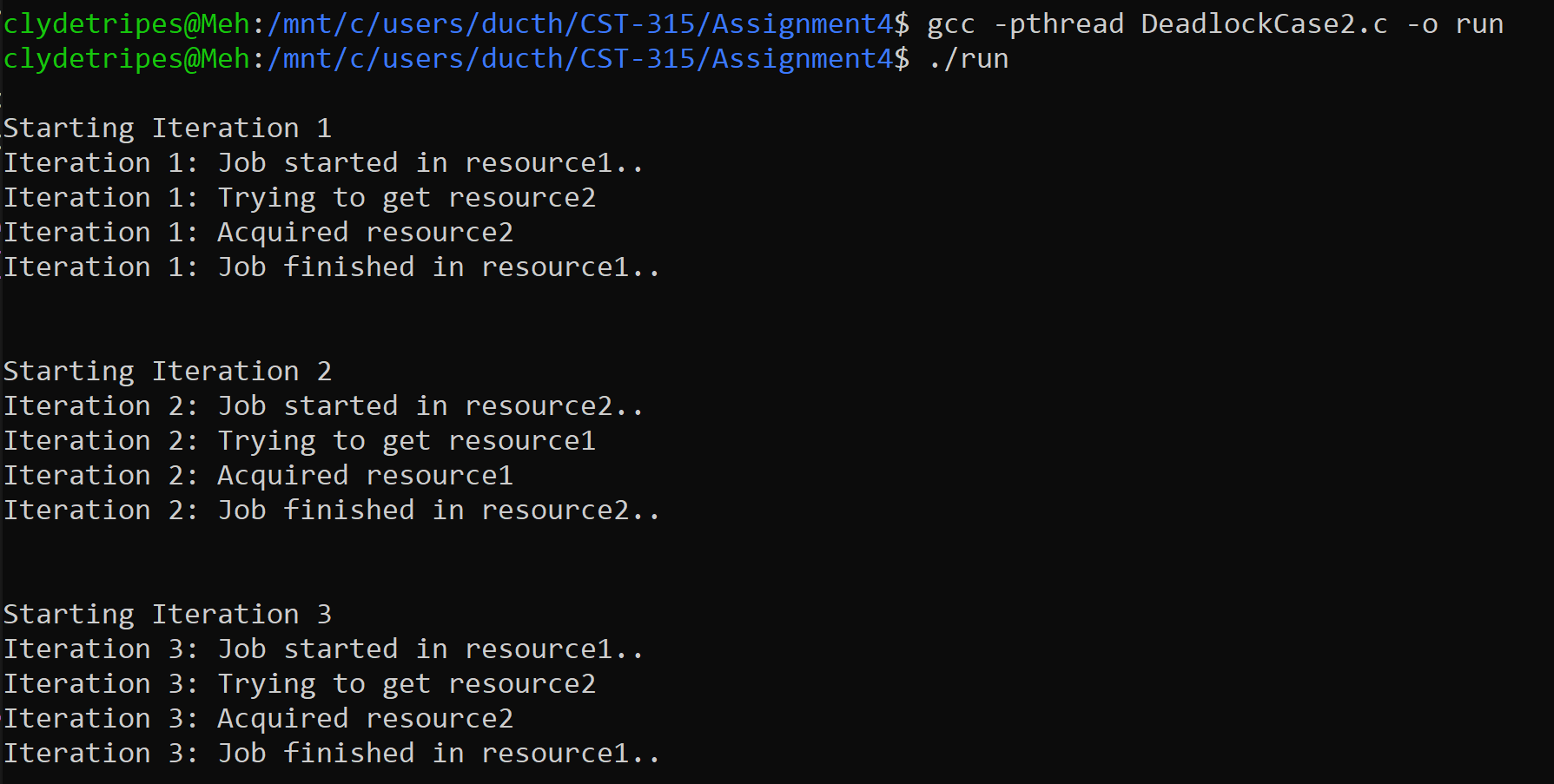
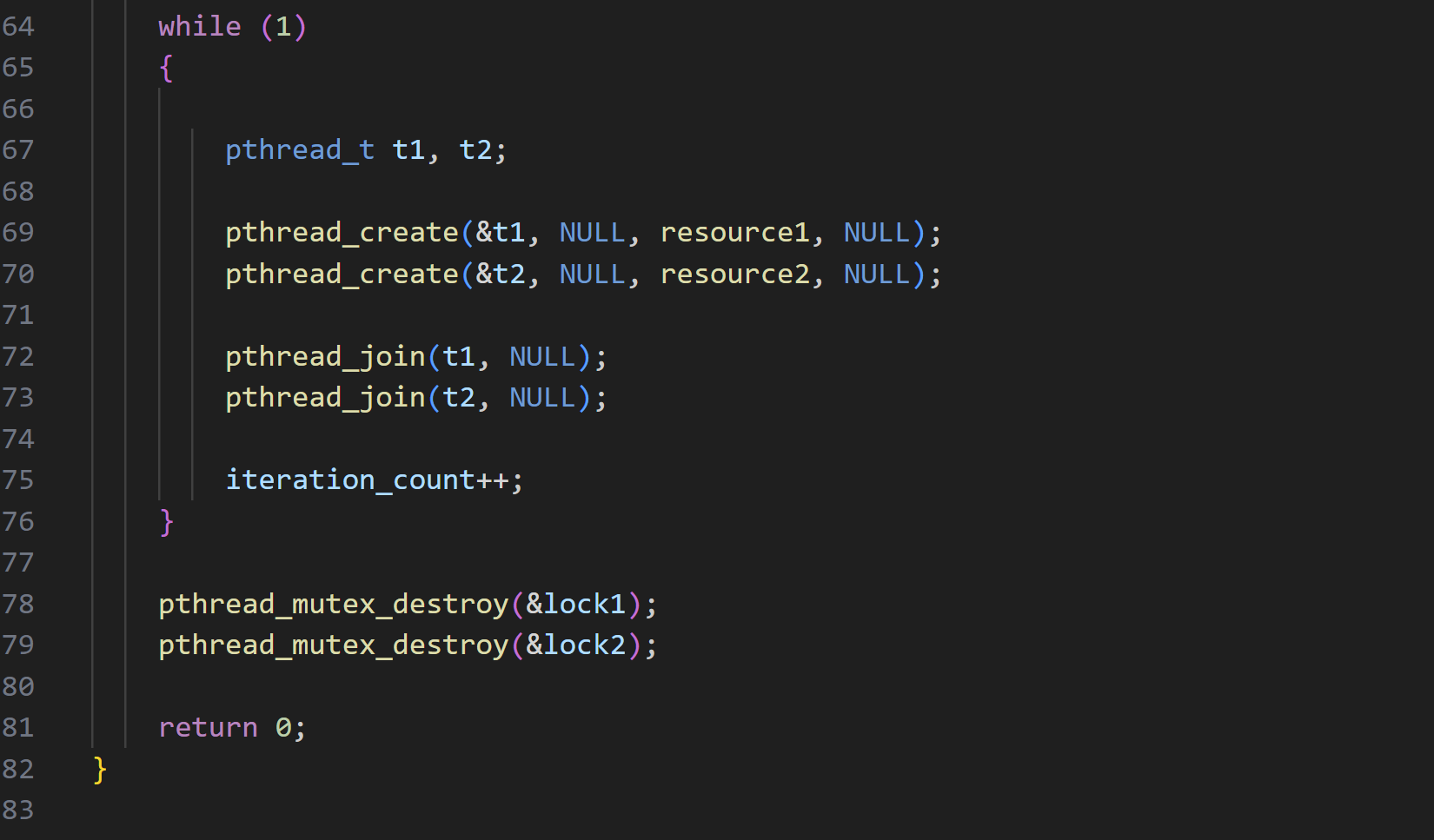
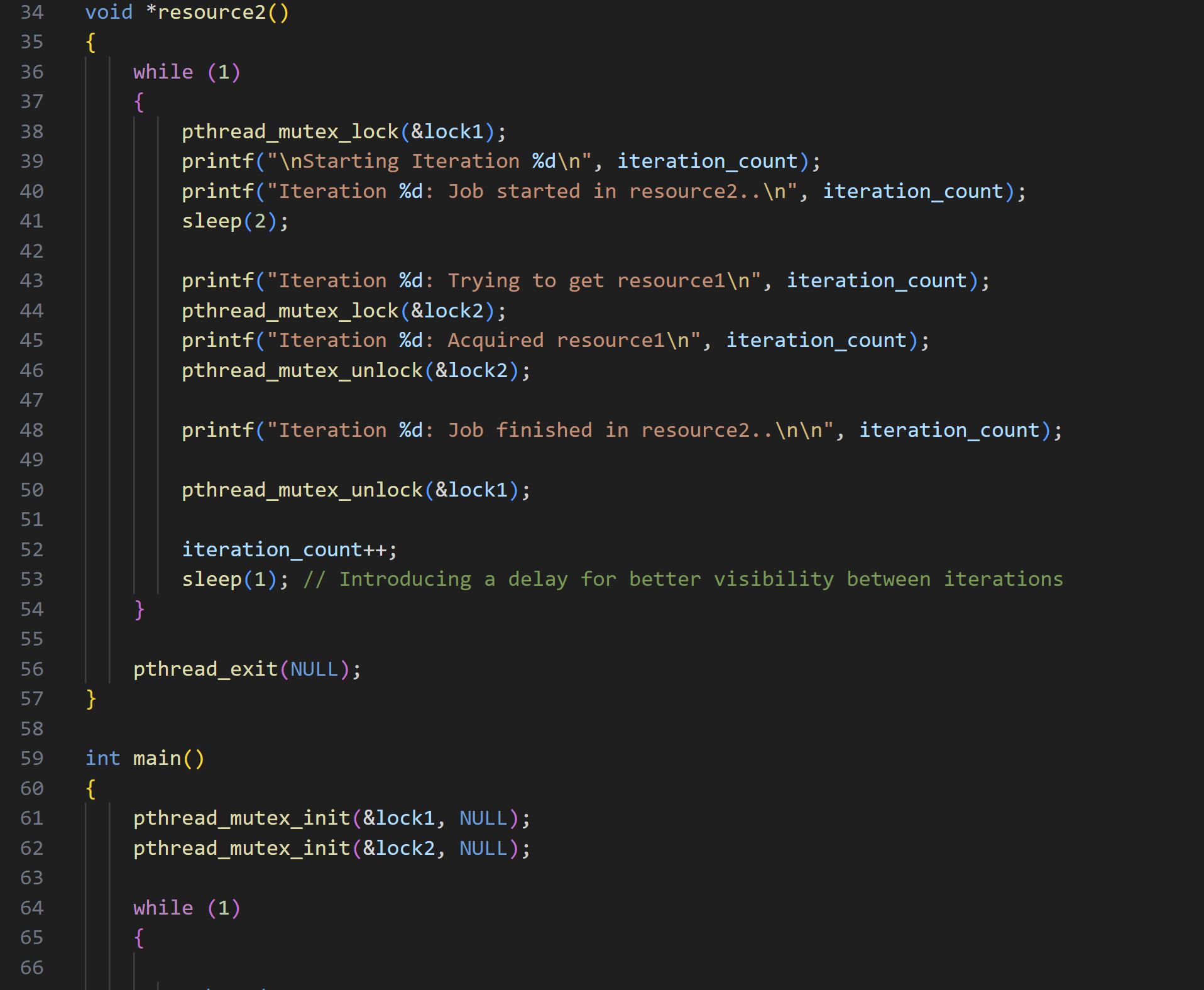
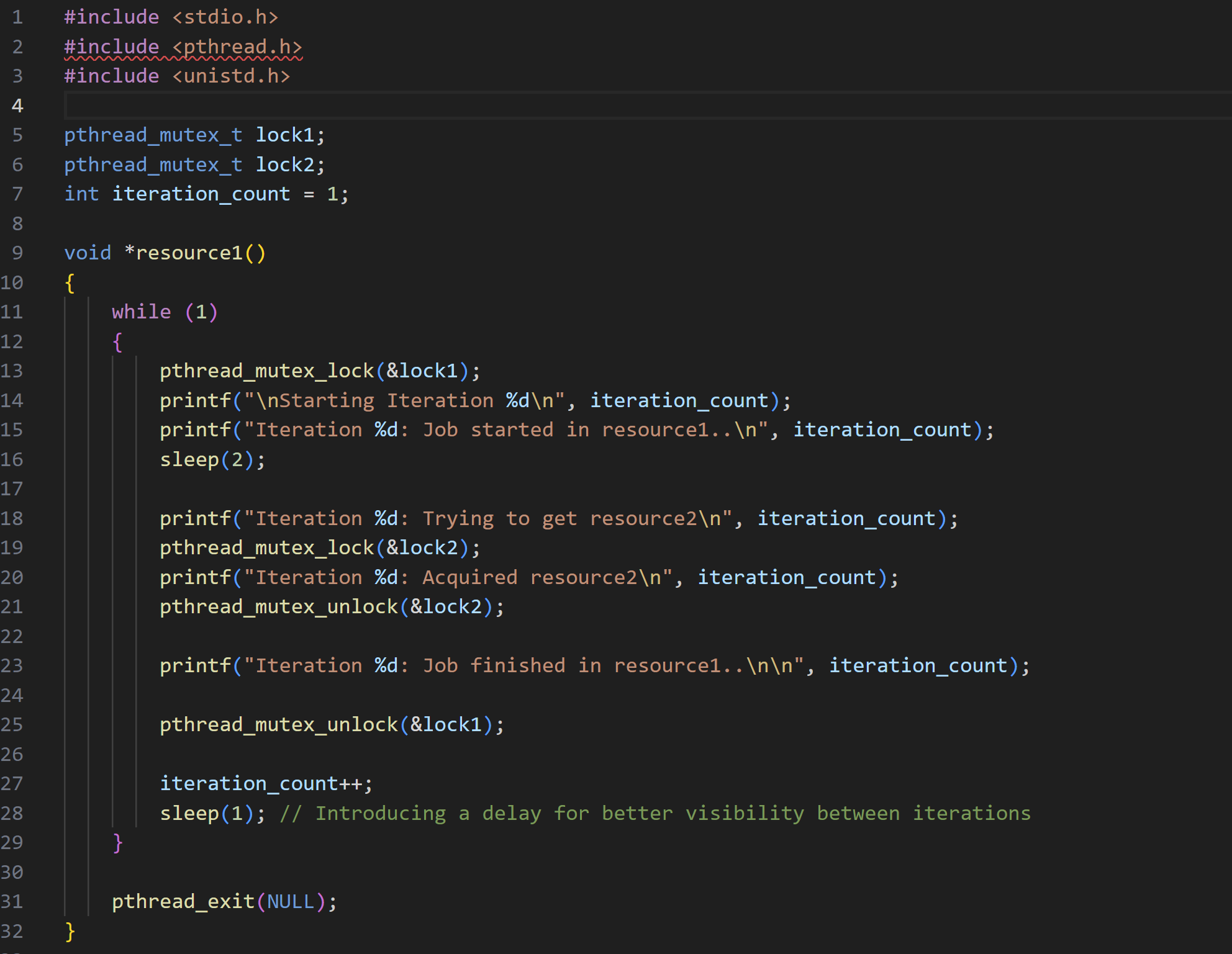
# Code Structure

## DeadlockCase1.c

## 

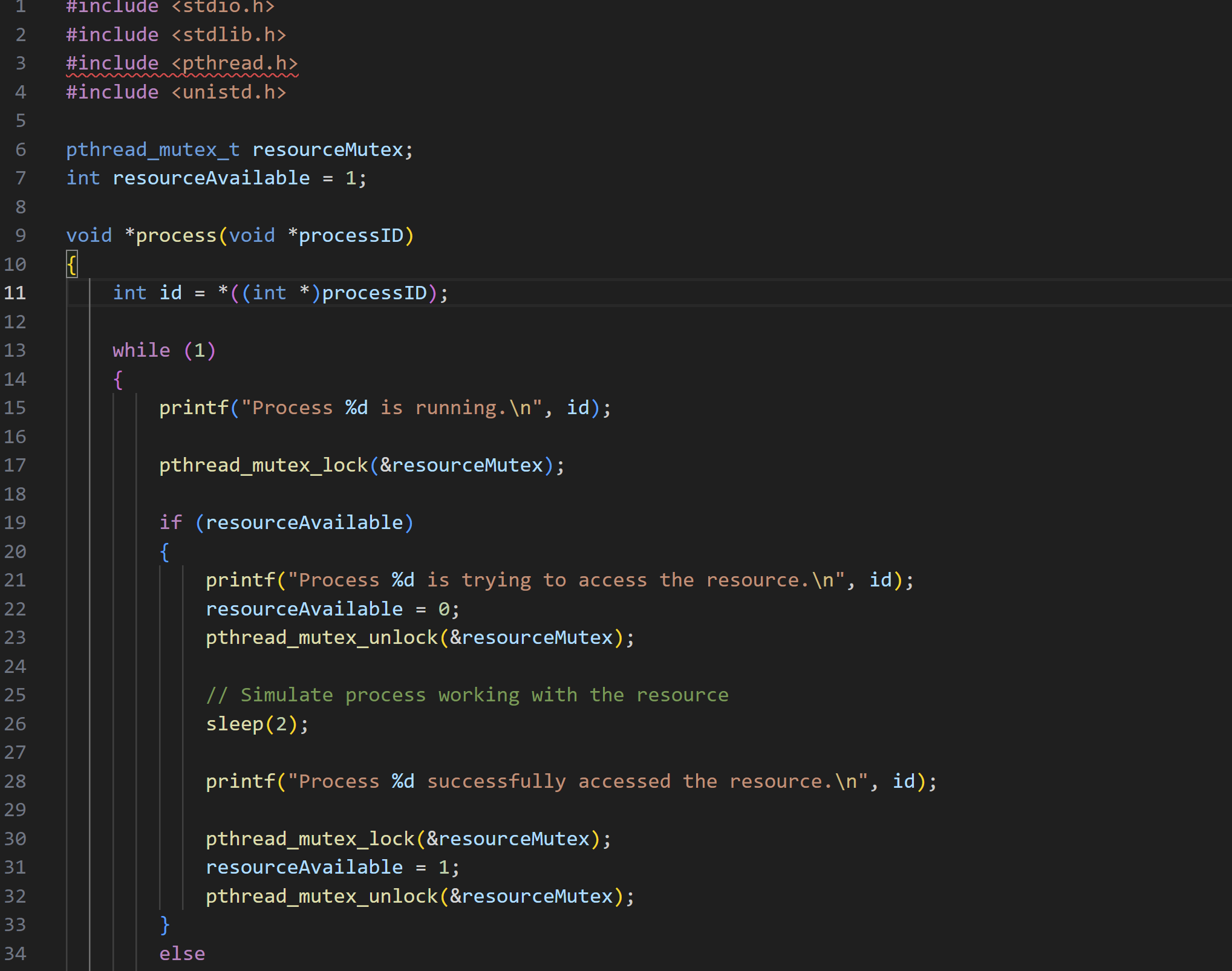
The DeadlockCase1.c code simulates a classic deadlock scenario where two threads contend for two resources, leading to a circular wait. Using two mutex locks (lock1 and lock2), the code demonstrates a scenario where both threads attempt to acquire locks on two resources, creating a deadlock situation.

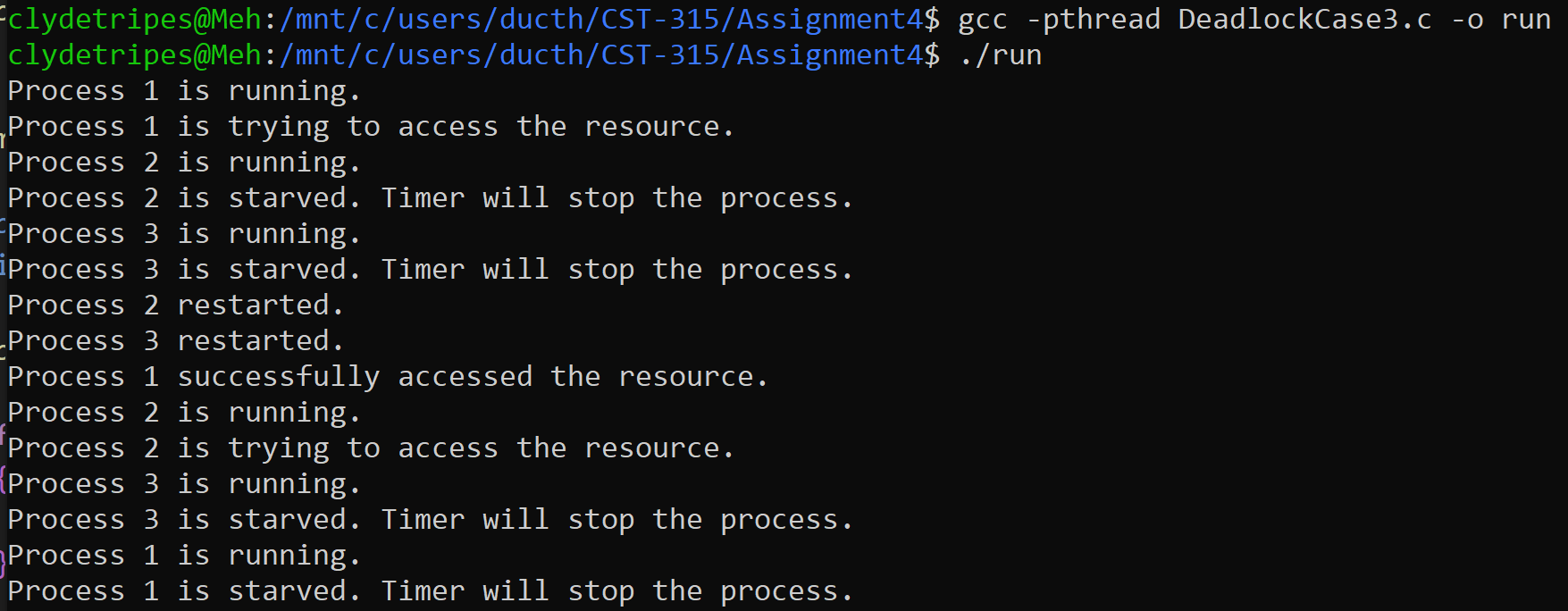
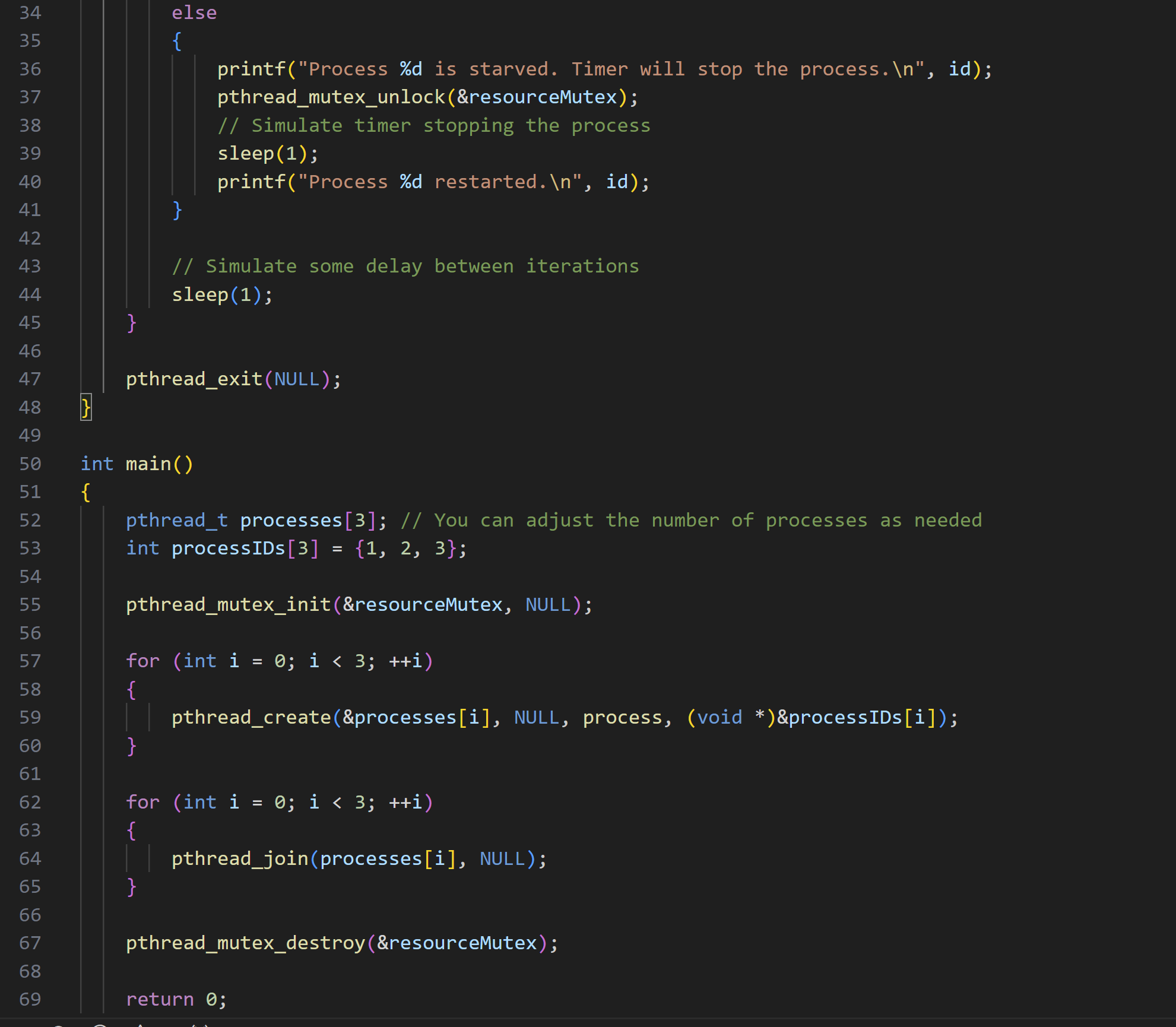
## DeadlockCase2.c



The code simulates a scenario where multiple processes compete for access to a shared resource, and a timer is introduced to handle situations where a process is starved. The code utilizes a single mutex (resourceMutex) to protect the shared resource and employs a timer mechanism to stop and restart processes, ensuring efficient resource management.

## DeadlockCase3.c





The final solution employs a mutex-based approach to manage shared resources among multiple threads. The initialization phase sets up a mutex to ensure exclusive access, preventing data race conditions. Threads, representing distinct processes, contend for the shared resource through a loop that initializes each thread and executes a logic block simulating resource contention. If the resource is available, the thread acquires the mutex, accesses the resource, and releases it, ensuring proper synchronization. In cases where the resource is unavailable, a timer mechanism is introduced to simulate a waiting period, effectively handling scenarios of thread starvation before the process restarts. The program meticulously waits for all threads to complete their execution before destroying the mutex, ensuring clean resource management.

This solution not only addresses the complexities of concurrent programming with shared resources but also integrates a realistic timer-based approach to handle situations where threads face resource unavailability. The combination of mutex synchronization and timer handling provides a comprehensive platform for studying and analyzing the challenges associated with effective resource management in concurrent scenarios.

# Conclusion

The implemented scenarios demonstrate the complexities of managing shared resources among multiple processes. The analysis helps evaluate the efficiency of the timer in addressing deadlock and starvation situations. Considerations for scalability and limitations are crucial for real-world applications.

**References**

“Introduction of Deadlock in Operating System.” GeeksforGeeks, GeeksforGeeks, 11 Feb. 2024, www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/.