Project 2: Synchronization

Course: CECS 326 SEC 02 - Operating Systems
Project: Synchronization
Team Members:

- Teammate 1: Phu Quach (ID: 029475548)
- Teammate 2: Ryan Tran (ID: 031190716)

Date: October 19, 2025

I . Summary

Our team successfully implemented a solution to the Dining Philosophers problem using POSIX mutex locks and condition variables in C. The implementation prevents deadlock through state management and uses condition variables to avoid busy waiting. We utilize a fork ownership tracking system that provides visual feedback on resource allocation.

YouTube Demo: https://www.youtube.com/watch?v=R09sOSwCJVw
GitHub: https://github.com/harvest7777/proj2

II. Project Implementation

1. Problem: In circular table is surrounded by five philosophers, numbered 0 through 4. There are five forks, and one between each pair of philosophers who are next to each other. The challenge is from the dining protocol: each philosopher requires both their left and right forks simultaneously to eat. The challenge is deadlock, which can occur if all five philosophers pick up their left fork and then wait indefinitely for their right fork to become available. Since each philosopher holds one fork and needs another, a circular wait condition emerges, resulting in system deadlock. Additionally, no philosopher is denied access to the forks. Race conditions present another concern, as multiple threads attempting to access shared resources simultaneously could lead to an inconsistency.
2. Solution :

- There's one main mutex (**pthread_mutex_t mutex**) that secures every critical section so updates happen atomically. Each philosopher has their own condition variable in pthread_cond_t cond[N], which lets a thread wait only when it can't move forward. We track what everyone's doing in int state[N] (thinking, hungry, or eating).
- Due to the sample output from the professor, we created **the time_cal() function** that uses the gettimeofday() system call to capture timestamps with milliseconds.
- Function :
    - We created the **test() function** where our synchronization logic is located. The ability of a philosopher to move from the hungry state to the eat state is determined by this function.

It checks the critical condition that a philosopher can only begin eating if they are hungry and both of their neighbors are currently not eating. As a result, when these conditions are valid, the function updates the philosopher's state to eat, assigns ownership of both required forks to that philosopher, and wakes them from waiting.

- The **pickup_forks()** function sets the philosopher's state to hungry after locking the mutex to enter the critical region. The test() function is called right away to see if the philosopher is ready to start eating. The philosopher goes into a wait loop on their condition variables if the requirements are not fulfilled. Importantly, **pthread_cond_wait()** releases the mutex while waiting and locks it again when signaled, it also lets others make progress and avoiding deadlock.
- The **return_forks()** function manages the release of resources after a philosopher finishes eating. It begins by acquiring the mutex lock, then updates the philosopher's state back to think. The fork ownership array is updated to show both forks as available by setting their values to -1. The function then tests both neighboring philosophers to see if they can now eat, waking them if they were waiting. It ensures that waiting philosophers are notified when resources become available, preventing starvation.
- The **print_forks()** function iterates through the fork ownership array and displays which philosopher currently holds each fork.
- Each philosopher runs as an independent thread executing the **philosopher() function**. First, the philosopher thinks for a random duration between one and three seconds, simulating contemplation. The randomization prevents synchronization patterns that could lead to starvation. After thinking, the philosopher attempts to acquire both forks by calling pickup_forks(). Once the forks are successfully acquired, the philosopher eats for another random duration between one and three seconds. After eating, the philosopher returns both forks, and the cycle continues. The thread displays timing information for each activity and shows fork ownership status after each eating session.

- Our program was designed to run continuously (according to the Teaching Assistant Grading Criteria), allowing time to observe multiple eating cycles for all philosophers. We observed that all five philosophers received multiple opportunities to eat during the infinite execution period. While the monitor solution allows starvation (as noted in the lecture slides), our testing showed that starvation does not occur in practice during our execution window

III. Contribution

1. **Ryan:**

- Developed the timing using **gettimeofday()** and implemented a fork ownership tracking.
- Created the **philosopher thread function** that coordinates the thinking-eating cycle and added the output formatting
- Writing documentation for the project, including **code comments**, the **README file**, the **makefile**, and **Record the Video**

2. **Phu Quach :**

- Implementing the core synchronization logic, including **the test(), pickup_forks(), and return_forks() functions**, with the support of the lecture slide. I also designed the **state management system** that prevents deadlock.
- **Writing the report**

IV. Conclusion

The solution shows deadlock-free, runs numerous tests without any instance of system freeze or infinite waiting. While the monitor solution can theoretically allow starvation, our implementation with random timing ensures all philosophers get opportunities to eat in practice. During all test runs, every philosopher successfully completed multiple eat-think cycles without any philosopher being starved. The addition of fork ownership tracking provides visibility into the synchronization process. The timing measurements confirm that the system operates efficiently with synchronization.

V. References

1. Lecture Slide Synchronization (Monitor Solution to Dining Philosophers)
2. Stack Overflow: "Trying to understand pthread_cond_lock and pthread_cond_signal"
    - https://stackoverflow.com/questions/52960662/
3. Stack Overflow: "What do the pthread_mutex_lock and pthread_mutex_unlock do"
    - https://stackoverflow.com/questions/72094630/
4. GeeksforGeeks: "Thread functions in C/C++"
    - https://www.geeksforgeeks.org/c/thread-functions-in-c-c/