Task 1 [30 minutes]

Consider a scenario where two types of threads, named “P”, and “C”, share a common, finite-sized buffer (or queue) as a means of communication. The challenge is to ensure that “P” does not produce items into the buffer when it'sfull and that “C” does not consume items from the buffer when it's empty. It is a typical producer-consumer problem that you have already implemented in one of your previous tasks. You have to use semaphores/ mutex to solve this problem.

Task 2 [30 minutes]

Write a program that performs matrix multiplication using threads to improve performance on multi-core processors. The program should take two input matrices, A and B, and calculate their product, C. The matrices A and B should have dimensions suitable for multiplication. The program should utilize a specified number of threads (equal to number of rows of matrix A) to parallelize the multiplication process. Each thread should be responsible for computing a portion of the result matrix, and the result should be stored in matrix C. Ensure that the program correctly divides the task among threads. After multiplication, the program should print the resulting matrix C. Adjust the dimensions of the matrices and the number of threads as needed. Perform synchronization as well. The resultant matrix must be correct.

Task 3 [60 minutes]

Implement Dining Philosophers problem using semaphores, we can assign a semaphore to each fork and limit the number of philosophers allowed to pick up forks simultaneously to prevent deadlock.

Here's a scenario illustrating this:

1. Create an array of n semaphores, where n is the number of philosophers. Each semaphore represents a fork. Initialize all semaphores to 1, indicating that the forks are initially available.
2. Each philosopher is represented by a separate process or thread. The main goal is to ensure that no two neighboring philosophers can pick up the same fork simultaneously.
3. Each philosopher follows the following steps:
    - Think: The philosopher spends some time thinking.
    - Pick up forks: To start eating, the philosopher attempts to acquire the semaphores representing the forks on their left and right.
    - If both forks are available, the philosopher acquires them by decrementing the respective semaphores. If either or both forks are not available (semaphore values are 0), the philosopher waits until they are available.
    - Eat: Once the philosopher has acquired both forks, they can start eating for a certain period.

- Put down forks: After finishing eating, the philosopher releases the forks by incrementing the respective semaphores, making them available to other philosophers.

By using semaphores to control access to the forks, we ensure that only a limited number of philosophers (equal to the number of available forks) can pick up forks simultaneously, preventing deadlock. This solution guarantees that each philosopher can eat without creating a circular dependency.