Introduction

In this assignment, we will explore the classic Dining Philosophers Problem, a synchronization problem that demonstrates challenges in coordinating access to shared resources among multiple concurrent processes (or threads). The problem involves a group of philosophers sitting around a table, each alternating between two states: **thinking** and **eating**. To eat, a philosopher must first pick up two chopsticks (shared resources), one on either side of their plate.

The primary issue this problem presents is ensuring that philosophers can pick up and release chopsticks in a way that avoids **deadlock** (where all philosophers are waiting indefinitely for resources) and **starvation** (where some philosophers are perpetually denied access to the chopsticks). This assignment will help you understand the intricacies of synchronization, thread management, and the use of mutexes and conditional variables to ensure efficient and fair access to shared resources.

The assignment is broken into two parts:

1. **Part 1** focuses on basic thread creation, where each philosopher is represented as a separate thread.
2. **Part 2** adds complexity by introducing multiple mutexes to simulate the picking up and putting down of chopsticks.

Through this exercise, you will gain hands-on experience with thread management, mutual exclusion, and solving real-world synchronization problems using semaphores, mutexes, and threading in C/C++.

Part 1 (5 points): Creation of Simple Threads

In this part, you will create a main program that accepts a single command-line argument (an integer) indicating the number of threads to be created. Follow these steps to complete this part:

1. **Process the Command-Line Argument**:
   - Retrieve the value of nthreads from the command-line argument.
2. **Output Initial Information**:
   - Print your name followed by "Assignment 4: # of threads = " and the value of nthreads.
3. **Thread Creation**:
   - Your main function should invoke the function:

```
void createPhilosophers(int nthreads);
```

where nthreads is the number of threads to be created. Inside this function, each thread will be created by calling the philosopherThread(void * pVoid) function, passing a pointer to an integer that holds the thread index.

4. **Thread Function**:

- Each philosopher thread will execute the function philosopherThread(), which will output the statement: "This is philosopher X", where X is the index of the current thread. Afterward, the thread function should return NULL.

5. **Thread Joining**:
   - In createPhilosophers(), the main thread must wait for all philosopher threads to complete by using pthread_join(). Once all threads are successfully joined, print: "N threads have been completed/joined successfully!" and then return.

File Submission:

- Name the source code file for this part as assign4-part1.c.
- Test your program using nthreads = 5 and nthreads = 20 (representing 5 and 20 philosophers).
- Save the program output in a file named REPORT1.txt.
- Briefly explain how you used pthread_join() in your report under the section labeled "Section: Part 1"

Part 2 (5 points): Using Multiple Mutexes or Binary Semaphores to Manage Chopsticks

In this part, we will implement one of three solutions we learned in the class to solve the deadlock problem in this particular case. Here, you will simulate the activities of philosophers using multiple **mutexes or binary semaphores** to manage the chopsticks ensuring a philosopher can pick up forks **only if both are available**. Each philosopher will go through the cycle: **thinking → picking up chopsticks (if both available) → eating → putting down chopsticks**.

1. **Define the Following Functions**:
   - void thinking();
   - void pickUpChopsticks(int threadIndex);
   - void eating();
   - void putDownChopsticks(int threadIndex);
2. **Chopstick Management**:
   - Each philosopher needs two chopsticks to eat, and each chopstick is shared by two neighboring philosophers. To prevent race conditions, represent each chopstick as a **mutex or binary semaphore**. A philosopher can only eat after successfully locking both the left and right chopsticks (represented by two locks).
3. **Function Implementations**:
   - In pickUpChopsticks(), lock the left and right chopsticks together if available for the philosopher.
   - In **eating()**, simulate the eating phase by pausing the thread using usleep(random_time * 1000), where random_time is a randomly generated value between **1 and 500 milliseconds**. During this phase, print the messages **"Philosopher #X: starts eating"** and **"Philosopher #X: ends eating"**, where **X** represents the philosopher's index.

- Similarly, implement thinking() with random sleep intervals.
- In putDownChopsticks(), release both chopstick locks.

File Submission:

- Name the source code file for this part as assign4-part2.c.
- Test the program with both 5 and 20 threads and include the output in REPORT2.txt
- Run the program 10 times and report if deadlock occurred under "Section: Part 2" in your report.

General Submission Guidelines:

- Submit the C source code files for both parts (assign4-part1.c and assign4-part2.c).
- Include REPORT1.txt, REPORT2.txt files containing the required outputs and explanations.
- The report file containing the explanations (Pdf/Doc/Docx format)
- Ensure that your programs compile and run correctly on a Unix/Linux environment.
- Compress all the files into a zip file and submit the zip file.
- **Note:** Late submissions will incur a flat penalty of 40% deduction if submitted after the due date.