**Cole Watson**

**Operating Systems and Architecture**

**Dr. Park**

**Fall 2024**

**11/19/24**

**TA Simulation Code Write-Up**

**Overview:** The TA Simulation project models a multithreaded environment where a teaching assistant (TA) helps students seeking assistance in an office with limited seating. The simulation addresses real-world problems of resource contention and scheduling using Java concurrency mechanisms, providing a practical example of multithreading and synchronization.

**Code Structure:**

1. **Constants and Variables:** The simulation defines key constants:
   * MAX\_CHAIRS: Limits the number of students who can wait in the "office" at any given time.
   * TOTAL\_STUDENTS: The total number of students in the simulation.
   * MAX\_RETRIES: Limits the number of attempts a student can make to acquire a chair before scheduling an appointment.

Additionally, shared resources like waitingList (for waiting students), appointments (a queue for students who make appointments), and concurrency controls (locks, condition variables, and semaphores) are initialized.

1. **Classes:**
   * **TeachingAssistant (TA):** This thread continuously checks for waiting students and processes appointments. If no students are present, the TA goes to sleep, using a condition variable to be awaken when a student arrives.
   * **Student:** Each student thread tries to acquire a chair. If unsuccessful after multiple attempts, the student schedules an appointment. This simulates a “retry” mechanism that reflects real-world behavior in a constrained resource environment.
2. **Synchronization Mechanisms:**
   * **Locks and Condition Variables:** A lock is used to control access to shared variables, ensuring thread safety when checking and updating waitingList and appointments. A condWakeup condition variable allows the TA to sleep and be woken up when a student arrives, or an appointment is scheduled.
   * **Semaphore (chairSemaphore):** This semaphore limits the number of students allowed to wait in the office to MAX\_CHAIRS. It ensures that no more than the specified number of students can hold a waiting spot at any time.
3. **Appointment System:**
   * Students unable to acquire a chair after the set retries make an appointment, which is stored in a queue. The TA prioritizes helping students with appointments, providing an orderly mechanism for handling excess demand.
4. **Simulation Flow:**
   * **Initialization:** The main method initializes the executor service, which starts the TA thread and all student threads.
   * **Student Arrival and Help Process:** Each student either waits in a chair or makes an appointment after reaching the retry limit. The TA serves students based on the waitingList or appointments queue, simulating realistic office hours and scheduling.
   * **End of Program:** When all students have been processed, the simulation sets an ENDOFPROGRAM flag, signaling the TA to exit.

**Implementation Strategies:**

1. **Controlled Concurrency:** The simulation uses a combination of locks, condition variables, and semaphores to safely manage shared resources. This ensures that the TA and students can operate independently without race conditions or deadlocks.
2. **Retry Mechanism for Fairness:** Students are given multiple attempts to secure a chair. This approach allows students an opportunity to join the queue but falls back on an appointment system if capacity is consistently full. This retry mechanism, coupled with the queue-based appointment system, provides both fairness and efficient scheduling.
3. **Separation of Concerns:** Dividing functionality into distinct classes (TeachingAssistant and Student) makes the code modular and easy to maintain. Each class has a specific role, simplifying both thread management and debugging.
4. **Detailed Logging:** Output statements provide a trace of all major events in the simulation, including student arrivals, TA actions, and waiting/appointment statuses. This logging is essential for debugging and understanding the interaction flow, offering visibility into the synchronization process.
5. **Graceful Shutdown:** The simulation uses the ENDOFPROGRAM flag to signal the TA thread when all students are processed. This ensures that the program exits cleanly without abruptly interrupting any ongoing operations.

**Conclusion:** The TA Simulation provides a well-rounded example of Java multithreading and synchronization, effectively modeling resource sharing in a constrained environment. By combining retry mechanisms, an appointment system, and robust concurrency controls, the project demonstrates practical strategies for managing thread interaction in an orderly and efficient manner.

import java.util.LinkedList;

import java.util.Queue;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Semaphore;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class TAAssistantSimulation {

private static final int MAX\_CHAIRS = 3;

private static final int TOTAL\_STUDENTS = 10;

private static final int MAX\_RETRIES = 3;

private static boolean ENDOFPROGRAM = false;

private int waitingList = 0;

private final Queue<Integer> appointments = new LinkedList<>();

private final Lock lock = new ReentrantLock();

private final Condition condWakeup = lock.newCondition();

private Semaphore chairSemaphore = new Semaphore(MAX\_CHAIRS, true);

class TeachingAssistant implements Runnable {

public void run() {

System.out.println("[TA THREAD] TA is in the office.");

try {

while (true) {

lock.lock();

try {

// TA sleeps if no students or appointments are waiting

while (waitingList == 0 && appointments.isEmpty() && !ENDOFPROGRAM) {

System.out.println("[TA THREAD] TA is sleeping.");

condWakeup.await();

}

// Break the loop and end if the program should end and no students are being helped

if (ENDOFPROGRAM && waitingList == 0 && appointments.isEmpty()) break;

// Serve students with appointments first

if (!appointments.isEmpty()) {

int studentId = appointments.poll();

System.out.println("[TA THREAD] TA is helping student " + studentId + " with an appointment.");

} else if (waitingList > 0) {

// Help the next waiting student from the hallway

System.out.println("[TA THREAD] TA is helping a student from the hallway.");

waitingList--;

chairSemaphore.release();

}

// Status after helping a student

System.out.println("[TA THREAD] TA is done with the student!");

System.out.println("[TA THREAD] Waiting students = " + waitingList + ", Student in the office = " + (waitingList > 0 || !appointments.isEmpty() ? 1 : 0));

} finally {

lock.unlock();

}

// Simulate time helping a student

Thread.sleep((long) (Math.random() \* 2000) + 1000);

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

System.out.println("[TA THREAD] TA has left the office.");

}

}

class Student implements Runnable {

private final int id;

public Student(int id) {

this.id = id;

}

public void run() {

int attempts = 0;

while (!ENDOFPROGRAM && attempts < MAX\_RETRIES) {

lock.lock();

try {

System.out.println("[SEMAPHORE] Student " + id + " gets the lock!");

System.out.println("[STUDENT THREAD] Student " + id + " is coming!");

if (chairSemaphore.tryAcquire()) {

waitingList++;

int chairNumber = MAX\_CHAIRS - waitingList + 1;

System.out.println("[STUDENT THREAD] Student " + id + " is waiting.");

System.out.println("[STUDENT THREAD] Student " + id + " is seating on the waiting chair #" + chairNumber + ".");

System.out.println("[STUDENT THREAD] Waiting students = " + waitingList + ", Student in the office = 1");

condWakeup.signal(); // Wake up TA if sleeping

return; // Student is now waiting for help, so exit loop

} else {

System.out.println("[STUDENT THREAD] Student " + id + " finds all chairs taken and leaves.");

attempts++;

}

} finally {

System.out.println("[SEMAPHORE] Student " + id + " releases the lock!");

lock.unlock();

}

// Wait a random time before retrying

try {

Thread.sleep((long) (Math.random() \* 2000));

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

// Make an appointment after 3 unsuccessful attempts

if (attempts >= MAX\_RETRIES) {

lock.lock();

try {

appointments.offer(id);

System.out.println("[STUDENT THREAD] Student " + id + " makes an appointment and will come back later.");

condWakeup.signal(); // Notify TA about the appointment and wake up if sleeping

} finally {

lock.unlock();

}

}

}

}

public void start() {

ExecutorService executor = Executors.newFixedThreadPool(TOTAL\_STUDENTS + 1);

// Start TA thread

TeachingAssistant ta = new TeachingAssistant();

executor.execute(ta);

// Start student threads

for (int i = 0; i < TOTAL\_STUDENTS; i++) {

Student student = new Student(i + 1);

executor.execute(student);

try {

Thread.sleep((long) (Math.random() \* 1000));

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

// End simulation and signal the TA to finish

lock.lock();

try {

ENDOFPROGRAM = true;

condWakeup.signal(); // Wake up TA to finish

} finally {

lock.unlock();

}

executor.shutdown();

while (!executor.isTerminated()) {

// Wait for all threads to finish

}

System.out.println("All threads have finished.");

}

public static void main(String[] args) {

new TAAssistantSimulation().start();

}

}