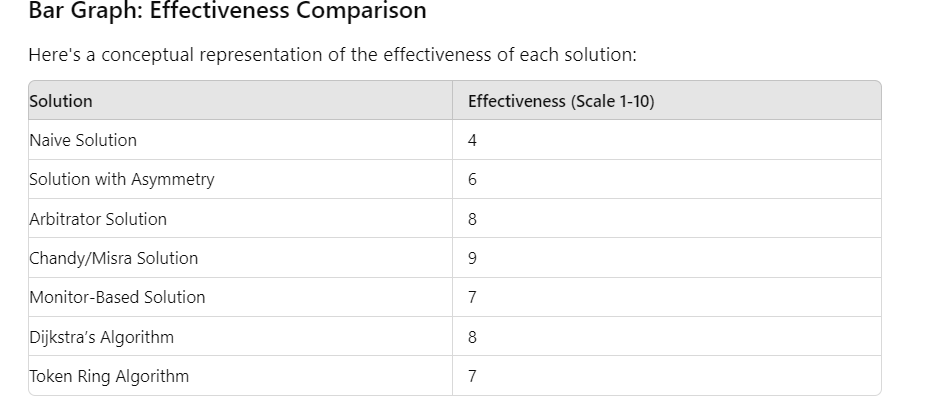
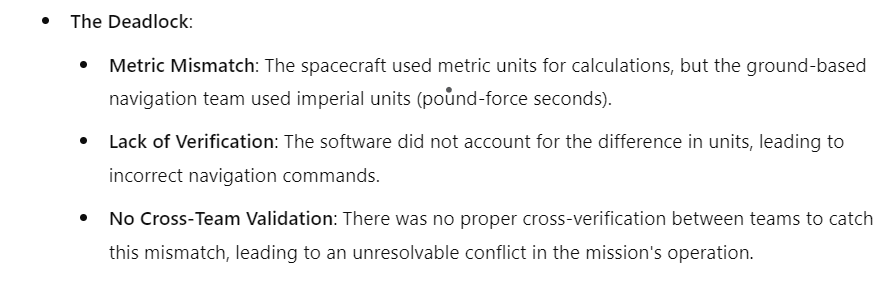


Mars Climate Orbiter deadlock scenario  


Decentralized Temporal Resource Allocation with Adaptive Priority Queuing (DTRA-APQ)  
(SRIYA’S ALGORITHM)

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.Random;

class Philosopher extends Thread {

private int id;

private Lock leftFork;

private Lock rightFork;

private int timeSlot;

private Random random;

public Philosopher(int id, Lock leftFork, Lock rightFork, int timeSlot) {

this.id = id;

this.leftFork = leftFork;

this.rightFork = rightFork;

this.timeSlot = timeSlot;

this.random = new Random();

}

public void run() {

try {

while (true) {

think();

tryToEat();

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

private void think() throws InterruptedException {

int thinkingTime = random.nextInt(2000) + 1000; // 1 to 3 seconds

System.out.println("Philosopher " + id + " is thinking for " + thinkingTime + " milliseconds.");

Thread.sleep(thinkingTime);

}

private void tryToEat() throws InterruptedException {

long currentTime = System.currentTimeMillis();

// Wait until the philosopher's time slot is active

if (currentTime % 10000 < timeSlot) {

System.out.println("Philosopher " + id + " is attempting to eat during time slot " + timeSlot + ".");

boolean acquiredLeft = leftFork.tryLock();

boolean acquiredRight = rightFork.tryLock();

if (acquiredLeft && acquiredRight) {

eat();

} else {

if (acquiredLeft) {

leftFork.unlock();

}

if (acquiredRight) {

rightFork.unlock();

}

System.out.println("Philosopher " + id + " failed to acquire both forks. Retrying in next time slot.");

}

} else {

System.out.println("Philosopher " + id + " is waiting for their time slot " + timeSlot + ".");

}

}

private void eat() throws InterruptedException {

int eatingTime = random.nextInt(3000) + 2000; // 2 to 5 seconds

System.out.println("Philosopher " + id + " is eating for " + eatingTime + " milliseconds.");

Thread.sleep(eatingTime);

leftFork.unlock();

rightFork.unlock();

}

}

public class DiningPhilosophersDTRA {

public static void main(String[] args) {

int numberOfPhilosophers = 5;

Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];

Lock[] forks = new ReentrantLock[numberOfPhilosophers];

for (int i = 0; i < numberOfPhilosophers; i++) {

forks[i] = new ReentrantLock();

}

for (int i = 0; i < numberOfPhilosophers; i++) {

int timeSlot = (i + 1) \* 2000; // Assign time slots (in milliseconds)

philosophers[i] = new Philosopher(i + 1, forks[i], forks[(i + 1) % numberOfPhilosophers], timeSlot);

philosophers[i].start();

}

}

}  
  
  
  
solution 2:   
import java.util.Random;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

// Shared Fork Resource

class Fork {

private final Lock lock = new ReentrantLock();

public boolean pickUp() {

return lock.tryLock();

}

public void putDown() {

lock.unlock();

}

}

// Philosopher Class

class Philosopher implements Runnable {

private final Fork leftFork;

private final Fork rightFork;

private final TimeSlotManager timeSlotManager;

private final PhilosopherCommunication communication;

private int philosopherId;

private int priority;

public Philosopher(int id, Fork leftFork, Fork rightFork, TimeSlotManager manager, PhilosopherCommunication comm) {

this.philosopherId = id;

this.leftFork = leftFork;

this.rightFork = rightFork;

this.timeSlotManager = manager;

this.communication = comm;

this.priority = 1; // Initial priority

}

@Override

public void run() {

try {

while (true) {

// Wait for the allocated time slot

Thread.sleep(timeSlotManager.getTimeSlot(philosopherId));

// Attempt to pick up forks

if (tryToEat()) {

// Successfully eating

System.out.println("Philosopher " + philosopherId + " is eating.");

Thread.sleep(new Random().nextInt(1000)); // Simulate eating

putDownForks();

timeSlotManager.successfulEat(philosopherId); // Inform manager

} else {

// Failed to pick up forks

System.out.println("Philosopher " + philosopherId + " failed to eat. Waiting for the next time slot.");

timeSlotManager.adjustTimeSlot(philosopherId); // Adjust time slot

}

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.out.println("Philosopher " + philosopherId + " was interrupted.");

}

}

private boolean tryToEat() {

if (leftFork.pickUp()) {

if (rightFork.pickUp()) {

return true; // Successfully picked up both forks

} else {

leftFork.putDown(); // Put down the left fork if the right is unavailable

}

}

return false;

}

private void putDownForks() {

rightFork.putDown();

leftFork.putDown();

}

}

// Time Slot Manager

class TimeSlotManager {

private final int[] timeSlots;

private final int baseTimeSlot = 500;

public TimeSlotManager(int numPhilosophers) {

timeSlots = new int[numPhilosophers];

for (int i = 0; i < numPhilosophers; i++) {

timeSlots[i] = baseTimeSlot + i \* 100; // Initial time slot allocation

}

}

public int getTimeSlot(int philosopherId) {

return timeSlots[philosopherId];

}

public void successfulEat(int philosopherId) {

// Reduce time slot delay after a successful eating

timeSlots[philosopherId] = Math.max(baseTimeSlot, timeSlots[philosopherId] - 50);

}

public void adjustTimeSlot(int philosopherId) {

// Increase time slot delay after a failed attempt

timeSlots[philosopherId] += 50;

}

}

// Philosopher Communication (Simulated)

class PhilosopherCommunication {

// Simulate communication by adjusting priorities, handling errors, etc.

public void communicate(int philosopherId, int newPriority) {

// Implementation of communication protocol (e.g., priority adjustments)

}

}

// Main class to start the simulation

public class DiningPhilosophersDTRA {

public static void main(String[] args) {

int numPhilosophers = 5;

Fork[] forks = new Fork[numPhilosophers];

Philosopher[] philosophers = new Philosopher[numPhilosophers];

TimeSlotManager timeSlotManager = new TimeSlotManager(numPhilosophers);

PhilosopherCommunication communication = new PhilosopherCommunication();

for (int i = 0; i < numPhilosophers; i++) {

forks[i] = new Fork();

}

for (int i = 0; i < numPhilosophers; i++) {

Fork leftFork = forks[i];

Fork rightFork = forks[(i + 1) % numPhilosophers];

philosophers[i] = new Philosopher(i, leftFork, rightFork, timeSlotManager, communication);

new Thread(philosophers[i]).start();

}

}

}  
  
  
