CONCURRENT PROGRAMMING

Assignment -1- Sleeping Barber Problem

## Balaji Sivakumar - 23260267

## Section 1: Import Statements

Import statements for required Java packages and classes, including “ExecutorService”, “TimeUnit”, “Date”, “LinkedList”, “Random”, “Scanner”, and “AtomicInteger”.

import static java.util.concurrent.TimeUnit.SECONDS;

import java.util.Date;

import java.util.LinkedList;

import java.util.List;

import java.util.Random;

import java.util.Scanner;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.atomic.AtomicInteger;

## Section 2: Main Class - “SleepingBarber”

The main class “SleepingBarber” contains the “main” method where the program starts execution.

public class SleepingBarber {

public static void main (String a[]) throws InterruptedException {

// **(Variable declarations and user inputs)**

}

}

## Section 3: Input Section

Input section where the user is prompted to enter the number of barbers (“numOfBarbers”) and the number of waiting chairs (“numOfChairs”). The default values are provided for “numOfBarbers” and “numOfCustomers”, but they can be overridden by user input.

int numOfBarbers=2, customerId=1, numOfCustomers=20, numOfChairs;

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of Barbers[M] : ");

numOfBarbers=sc.nextInt();

System.out.println("Enter the number of waiting Chairs[N] : ");

numOfChairs=sc.nextInt();

## Section 4: ExecutorService and Barber Shop Initialization

Creation of an “ExecutorService” (“exec”) with a fixed thread pool size of 12, an instance of the “BarberShop” class (“shop”) with the specified number of barbers and chairs, and a “Random” object (“r”) for introducing random delays.

ExecutorService exec = Executors.newFixedThreadPool(12);

BarberShop shop = new BarberShop(numOfBarbers, numOfChairs);

Random r = new Random();

## Section 5: Barber Shop Simulation

Simulation of the barber shop: Threads are created for each barber (“Barber” class) and customer (“Customer” class) and submitted to the “ExecutorService”. Random delays are introduced to simulate the time between customer arrivals.

for(int i=1; i<=numOfBarbers;i++) {

Barber barber = new Barber(shop, i);

Thread thbarber = new Thread(barber);

exec.execute(thbarber);

}

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

Customer customer = new Customer(shop);

customer.setInTime(new Date());

Thread thcustomer = new Thread(customer);

customer.setcustomerId(customerId++);

exec.execute(thcustomer);

try {

double val = r.nextGaussian() \* 2000 + 2000;

int millisDelay = Math.abs((int) Math.round(val));

Thread.sleep(millisDelay);

}

catch(InterruptedException iex) {

iex.printStackTrace();

}

}

## Section 6: ExecutorService Shutdown and Await Termination

Shutdown of the “ExecutorService” after all tasks are submitted and waiting for all threads to complete.

exec.shutdown();

exec.awaitTermination(12, SECONDS);

## Section 7: Output and Statistics

Outputting statistics about the barber shop, including the total number of customers, the number of customers served, and the number of customers lost.

long elapsedTime = System.currentTimeMillis() - startTime;

System.out.println("\nBarber shop is closed now");

System.out.println("\nTotal no of customers: "+numOfCustomers+

"\nTotal no of customers served: "+shop.getTotalHairCutsDone()

+"\nTotal no of customers lost: "+shop.getCustomerLost());

sc.close();

## Section 8: Barber Class - “Barber”

The `Barber` class implements the `Runnable` interface. Each barber continuously calls the `cutHair` method in the barber shop.

class Barber implements Runnable {

BarberShop shop;

int barberId;

public Barber(BarberShop shop, int barberId) {

this.shop = shop;

this.barberId = barberId;

}

public void run() {

while(true) {

shop.cutHair(barberId);

}

}

}

## Section 9: Customer Class - “Customer”

The “Customer” class implements the “Runnable” interface. Each customer enters the shop, and the “getForHairCut” method is called.

class Customer implements Runnable {

int customerId;

Date inTime;

BarberShop shop;

public Customer(BarberShop shop) {

this.shop = shop;

}

**// Variables Declarations**

**// Getter and Setter Methods comes here**

public void run() {

getForHairCut();

}

private synchronized void getForHairCut() {

shop.add(this);

}

}

## Section 10: BarberShop Class - “BarberShop”

class BarberShop {

**// Variable declarations**

public BarberShop(int numOfBarbers, int numOfChairs){

**// Constructor**

}

**// Getter Methods**

public void cutHair(int barberId)

{

**// Method to simulate hair cutting**

}

public void add(Customer customer) {

**// Method to simulate a customer entering the shop**

}

The “BarberShop” class manages the state of the barber shop, including the number of barbers, chairs, and customers. It contains methods for cutting hair (“cutHair”) and adding customers to the shop (`add`).

## Conclusion

Random delays are introduced between customer arrivals. The program simulates the Sleeping Barber problem, where customers may get a haircut or leave if no barbers or chairs are available. The use of synchronization ensures proper handling of shared data structures. AtomicInteger counters are used for tracking the total number of haircuts and the number of customers who left.

# 

# 

# Description of where a solution to the sleeping barber(s) problem is used in practice

* Task Scheduling in Operating Systems:

Operating systems use scheduling algorithms to manage tasks competing for CPU time. The principles of fairness, avoiding starvation, and efficient resource utilization draw inspiration from problems like the Sleeping Barber.

* Database Management Systems:

Concurrency control mechanisms in database systems prevent the conflicts and ensure consistency when multiple transactions access and modify shared data concurrently. Techniques such as locking and transaction isolation levels address similar challenges.

* Resource Allocation in Cloud Computing:

In cloud computing environments, where multiple virtual machines or containers share physical resources, the principles of resource allocation and avoiding contention are crucial. Solutions to the Sleeping Barber problem are analogous to ensuring efficient resource utilization in cloud environments.

* Parallel Processing and High-Performance Computing:

In parallel computing systems and supercomputers, efficient scheduling of tasks across multiple processors or cores is essential for maximizing performance. Concepts from the Sleeping Barber problem can be applied to avoid bottlenecks and contention for shared resources.

* Distributed Systems:

Distributed systems involve multiple nodes communicating and collaborating to achieve a common goal. Solutions to problems like the Sleeping Barber are relevant for designing distributed algorithms that maintain consistency and avoid conflicts in a distributed environment.

* Network Resource Management:

In networking scenarios, where multiple devices or applications contend for network resources, solutions inspired by concurrency control can help avoid congestion, packet loss, and ensure fair access to the network.

* Concurrency in Software Development:

In software development, especially in concurrent programming, avoiding race conditions, deadlocks, and ensuring thread safety are critical.