Date of publication 22 May 2024

Simulating a Party with Limited Resources Using Multithreading in Java

FURKAN BAYTAK[1], FURKAN ÖZKAYA[2],
[1]Department of Computer Engineering, Manisa Celal Bayar University, Manisa, Türkiye
[2]Department of Computer Engineering, Manisa Celal Bayar University, Manisa, Türkiye

This project is a part of the coursework for the "CSE3124 Operating Systems" class at the Manisa Celal Bayar University.

ABSTRACT This paper presents a simulation of a party scenario with limited resources using multithreading in Java. The simulation involves eight guests and a waiter serving "börek," "cake," and "drink" with specific consumption limits and tray capacities. The objective is to ensure all guests consume from each type of food and drink at least once, and the waiter refills the trays when necessary until all resources are exhausted. The implementation demonstrates the coordination between multiple threads representing guests and the waiter, ensuring synchronized access to shared resources.

INDEX TERMS Multithreading, Synchronization, Java, Resource Management, Simulation

I. INTRODUCTION

SIMULATING real-world scenarios involving limited resources and multiple consumers can provide valuable insights into efficient resource management and synchronization. This paper presents a Java-based simulation of a party with eight guests and a waiter. The resources to be consumed include 30 "börek," 15 slices of cake, and 30 glasses of drink, served on trays with limited capacity. Each guest must consume at least one item from each type, with specific consumption limits, while the waiter refills the trays when they are nearly empty. The primary challenge is to coordinate the actions of guests and the waiter using multithreading and synchronization mechanisms in Java.

II. METHODS

The simulation involves three main components: trays, guests, and a waiter. Each component is represented as a class with specific responsibilities and behaviors.

A. TRAY CLASS

The `Tray` class represents a tray with a specific capacity and a maximum number of items that can be served. It includes methods for guests to take items and for the waiter to refill the tray. Synchronization is used to ensure that the tray's state is consistently updated by multiple threads.

- `Tray(String name,`
- `int capacity, int maxTotalItems)`: Constructor to initialize the tray with a given name, capacity, and maximum total items.
- `synchronized boolean takeItem(String guestName)`: Method to allow a guest to take an item from the tray. It checks if there are items available and updates the tray state accordingly.
- `synchronized void refillItem()`: Method to refill the tray with items. It ensures that the tray is refilled only up to its capacity or the remaining items that can be served.
- `synchronized int getRemainingItems()`: Method to get the remaining items that can be served from the tray.
- `synchronized boolean hasItemsLeft()`: Method to check if there are items left to be served from the tray.

B. GUEST CLASS

The `Guest` class represents a guest at the party. Each guest has limits on the number of items they can consume from each tray. The `run` method of the `Guest` class simulates the process of taking items from the trays, with random sleep intervals to mimic the time between consumptions.

- `Guest(String name, Tray borekTray, Tray cakeTray, Tray drinkTray)`: Constructor to initialize the guest with a name and trays for börek, cake, and drink.
- `void run()`: Method that simulates the guest's actions. The guest tries to take items from each tray based on their limits and sleeps for a random duration between actions.

C. WAITER CLASS

The `Waiter` class represents the waiter responsible for refilling the trays. The waiter's `run` method continuously checks the trays and refills them if the number of items falls below a certain threshold. Synchronization ensures that the refilling process does not interfere with guests taking items.

- `Waiter(Tray... trays)`: Constructor to initialize the waiter with trays.
- `void run()`: Method that simulates the waiter's actions. The waiter continuously refills trays while not interrupted and sleeps for 500 milliseconds between checks.

D. SYNCHRONIZATION AND THREAD MANAGEMENT

Java's synchronized blocks and methods are used to manage access to the shared tray resources, ensuring that only one thread can modify the state of a tray at a time. This prevents race conditions and ensures the integrity of the simulation.

Procedure: `Guest::run`
while items available and limits not reached **do**
  if börek available **then**
    Take börek
  end if
  if cake available **then**
    Take cake
  end if
  if drink available **then**
    Take drink
  end if
  Sleep for random duration
end while

Procedure: `Waiter::run`
while not interrupted **do**
  for each tray **do**
    if tray needs refill **then**
      Refill tray
    end if
  end for
  Sleep for 500 ms
end while

E. MAIN METHOD AND TESTING

To demonstrate and test the simulation, a `Main` class is used. This class sets up the trays, guests, and waiter, and initiates their interactions.

- `public static void main(String[] args)`: This method initializes the trays for börek, cake, and drink with specified capacities and maximum total items. It creates a waiter responsible for refilling the trays and starts the waiter thread. Eight guest threads are created and started, each attempting to consume items from the trays according to their limits. The main method waits for all guest threads to finish and then interrupts the waiter thread, ensuring all threads complete their tasks.

The testing involves observing the output of the program, which includes messages indicating when a guest takes an item and when the waiter refills a tray. The final output shows the remaining items on each tray, verifying that the simulation correctly managed the resources and synchronized access.

III. CONCLUSION

This paper presents a practical example of using multithreading and synchronization in Java to simulate a party scenario with limited resources. The simulation demonstrates the challenges and solutions for coordinating multiple consumers and a single producer in a concurrent environment. The implementation was tested thoroughly to ensure accurate synchronization and correct behavior of the simulation. The successful implementation of this simulation provides insights into resource management and synchronization techniques applicable to various real-world scenarios.

```
Guest 2 took 1 drink. Items left on tray: 2
Guest 6 took 1 borek. Items left on tray: 1
Guest 6 took 1 drink. Items left on tray: 1
Guest 1 took 1 borek. Items left on tray: 0
Guest 1 took 1 cake. Items left on tray: 0
Guest 1 took 1 drink. Items left on tray: 0
Waiter refilled the borek. Items now: 5
Waiter refilled the drink. Items now: 5
Guest 6 took 1 borek. Items left on tray: 4
Guest 6 took 1 drink. Items left on tray: 4
Guest 4 took 1 borek. Items left on tray: 3
Guest 4 took 1 drink. Items left on tray: 3
Guest 7 took 1 borek. Items left on tray: 2
Guest 7 took 1 drink. Items left on tray: 2
Waiter refilled the borek. Items now: 5
Waiter refilled the drink. Items now: 5
Guest 8 took 1 borek. Items left on tray: 4
Guest 8 took 1 drink. Items left on tray: 4
Guest 2 took 1 borek. Items left on tray: 3
Guest 2 took 1 drink. Items left on tray: 3
Guest 7 took 1 borek. Items left on tray: 2
Guest 3 took 1 borek. Items left on tray: 1
Guest 3 took 1 drink. Items left on tray: 2
Guest 1 took 1 borek. Items left on tray: 0
Guest 1 took 1 drink. Items left on tray: 1
Waiter refilled the borek. Items now: 1
Waiter refilled the drink. Items now: 2
Guest 5 took 1 borek. Items left on tray: 0
Guest 5 took 1 drink. Items left on tray: 1
Guest 5 took 1 drink. Items left on tray: 0
Waiter finished his job.
All food and drink is consumed.
Remaining borek: 0
Remaining cake: 0
Remaining drink: 0
```

FIGURE 1. Simulation results showing the consumption of items by guests and refilling actions by the waiter.

REFERENCES

[1] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts Essentials, 2nd Edition*.

[2] *https://www.w3schools.com/java/java_threads.asp*.

[3] *https://www.javatpoint.com/how-to-create-a-thread-in-java*.


```java
// Tray class representing a tray with a specific capacity and maximum total items
class Tray {
    // Variables to track item count, capacity, maximum total items, total served items, and name of the tray
    // itemCount: current number of items on the tray
    // capacity: maximum capacity of the tray
    // maxTotalItems: maximum total items that can be served from the tray
    // totalServedItems: total number of items served from the tray
    // name: name of the tray
    private int itemCount = 0;
    private final int capacity;
    private final int maxTotalItems;
    private int totalServedItems = 0;
    private final String name;

    // Constructor to initialize the tray with a given name, capacity, and maximum total items
    public Tray(String name, int capacity, int maxTotalItems) {
        this.name = name;
        this.capacity = capacity;
        this.maxTotalItems = maxTotalItems;
    }

    // Method to allow a guest to take an item from the tray
    public synchronized boolean takeItem(String guestName) {
        if (this.itemCount > 0 && this.totalServedItems < this.maxTotalItems) {
            this.itemCount--;
            this.totalServedItems++;
            System.out.println(guestName + " took 1 " + this.name + ". Items left on tray: " + this.itemCount);
            return true;
        }
        return false;
    }
     // Method to refill the tray with items
     public synchronized void refillItem() {
         if (this.itemCount < this.capacity && this.totalServedItems < this.maxTotalItems) {
             this.itemCount = Math.min(this.capacity, this.maxTotalItems - this.totalServedItems);
             System.out.println("Waiter refilled the " + this.name + ". Items now: " + this.itemCount);
         }
     }

     // Method to get the remaining items that can be served from the tray
     public synchronized int getRemainingItems() {
         return maxTotalItems - totalServedItems;
     }

     // Method to check if there are items left to be served from the tray
     public synchronized boolean hasItemsLeft() {
         return totalServedItems < maxTotalItems;
     }
}
```

```java
// Guest class representing a guest that can take items from different trays
class Guest extends Thread {
    // Tray objects for borek, cake, and drink
    private Tray borekTray, cakeTray, drinkTray;
    // Limits for the number of items each guest can take
    private int borekLimit = 4, cakeLimit = 2, drinkLimit = 4;
    // Counters for the number of items taken from each tray
    private int borekTaken = 0, cakeTaken = 0, drinkTaken = 0;

    // Constructor to initialize the guest with a name and trays for borek, cake, and drink
    public Guest(String name, Tray borekTray, Tray cakeTray, Tray drinkTray) {
        super(name);
        this.borekTray = borekTray;
        this.cakeTray = cakeTray;
        this.drinkTray = drinkTray;
    }

    @Override
    public void run() {
        try {
            // Guest continues to take items while there are items left on any tray and the limits have not been reached
            while ((borekTray.hasItemsLeft() || cakeTray.hasItemsLeft() || drinkTray.hasItemsLeft()) &&
                   (borekTaken < borekLimit || cakeTaken < cakeLimit || drinkTaken < drinkLimit)) {
                // Attempt to take items from each tray based on limits
                // Sleep for a random duration to simulate time between taking items
                if (borekTray.hasItemsLeft() && borekTaken < borekLimit && borekTray.takeItem(getName())) borekTaken++;
                if (cakeTray.hasItemsLeft() && cakeTaken < cakeLimit && cakeTray.takeItem(getName())) cakeTaken++;
                if (drinkTray.hasItemsLeft() && drinkTaken < drinkLimit && drinkTray.takeItem(getName())) drinkTaken++;
                Thread.sleep((long) (Math.random() * 1500));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```java
// Main class to demonstrate the interaction between guests and trays
public class Main {
    Run | Debug
    public static void main(String[] args) throws InterruptedException {
        // Create trays for borek, cake, and drink
        Tray borekTray = new Tray(name:"borek", capacity:5, maxTotalItems:30);
        Tray cakeTray = new Tray(name:"cake", capacity:5, maxTotalItems:15);
        Tray drinkTray = new Tray(name:"drink", capacity:5, maxTotalItems:30);

        // Create a waiter with the trays
        Waiter waiter = new Waiter(borekTray, cakeTray, drinkTray);
        waiter.start();

        // Create guests and start their threads
        Guest[] guests = new Guest[8];
        for (int i = 0; i < guests.length; i++) {
            guests[i] = new Guest("Guest " + (i + 1), borekTray, cakeTray, drinkTray);
            guests[i].start();
        }

        // Wait for all guests to finish and then interrupt the waiter
        for (Guest guest : guests) {
            guest.join();
        }
        waiter.interrupt();
        waiter.join();

        // Print remaining items on each tray after all guests have finished
        System.out.println(x:"All food and drink is consumed.");
        System.out.println("Remaining borek: " + borekTray.getRemainingItems());
        System.out.println("Remaining cake: " + cakeTray.getRemainingItems());
        System.out.println("Remaining drink: " + drinkTray.getRemainingItems());
    }
}
```