15.3-Managing Concurrent Processes

Program

```
using System;
using System.Threading;

namespace Test2
{
    internal class Program
    {
        static int ticketsAvailable = 100;
        static void Main()
        {
            for (int i = 1; i <= 10; i++)
            {
                Thread t = new Thread(BookTicket);
                t.Start(i);
            }

            Console.ReadLine();
        }

        static void BookTicket(object userId)
        {
            Monitor.Enter(typeof(Program));

            try
```

```
{
    if (ticketsAvailable > 0)
    {
        ticketsAvailable--;
        Console.WriteLine($"User {userId} booked a ticket. " +
            $"Tickets available: {ticketsAvailable}");
    }
    else
    {
        Console.WriteLine($"User {userId} couldn't book a ticket. " +
            $"No tickets available.");
    }
}
finally
{
    Monitor.Exit(typeof(Program));
}

Thread.Sleep(1000);

Monitor.Enter(typeof(Program));

try
{
    ticketsAvailable++;
    Console.WriteLine($"User {userId} canceled the ticket. " +
        $"Tickets available: {ticketsAvailable}");
```

```
            }
            finally
            {
                Monitor.Exit(typeof(Program));
            }
        }
    }
}
```

Exercise

Write a C# program that simulates a ticket booking system where multiple users attempt to book tickets concurrently. The program initializes with 100 tickets available for booking. It creates ten threads, each representing a user, and attempts to book a ticket. If a ticket is available, the user books it, and the number of available tickets decreases. If no tickets are available, the user receives a message indicating that booking was unsuccessful. After a brief pause, each user cancels their booking, and the available ticket count increases accordingly.

The core functionality revolves around the BookTicket method, which is executed by each thread. This method utilizes Monitor.Enter and Monitor.Exit to ensure that only one thread at a time can access the shared resource, which is the ticketsAvailable variable representing the number of available tickets. This prevents race conditions and ensures thread safety. Additionally, the program utilizes Thread.Sleep to simulate a delay in ticket booking and cancellation.

Hint

Shared Resource: Identify the shared resource in the program, which is the ticketsAvailable variable representing the number of available tickets for booking.

Thread Creation: threads are created in the Main method using a loop to simulate multiple users attempting to book tickets concurrently. Each thread is started with the BookTicket method as the entry point.

Critical Section: Recognize the critical section of the program, where access to the shared resource (ticketsAvailable) is synchronized using Monitor.Enter and Monitor.Exit. This ensures that only one thread can modify the shared resource at a time to prevent race conditions.

Booking Tickets: Create logic within the BookTicket method, where each user thread attempts to book a ticket if tickets are available. If successful, it decrements the ticketsAvailable count. If no tickets are available, it displays a message indicating the unavailability of tickets.

Cancellation of Tickets: pause (simulated by Thread.Sleep), each user cancels their ticket reservation. This operation also requires synchronization to access the shared resource.

Explanation

This program program simulates a scenario where multiple users attempt to book tickets concurrently for an event. The program maintains a shared resource, represented by the ticketsAvailable variable, which tracks the number of available tickets. Here's an explanation of the program's functionality:

The Main method initializes the program by starting a loop that creates and starts ten threads, each representing a user trying to book a ticket. Inside the loop, a new thread t is created using the Thread class, with the BookTicket method as the entry point. Each thread is then started with a unique user ID ranging from 1 to 10.

The BookTicket method represents the behavior of each user thread attempting to book a ticket. Access to the shared resource (ticketsAvailable) is synchronized using the Monitor.Enter and Monitor.Exit statements, ensuring that only one thread can modify Within the BookTicket method's critical section, the program checks if there are available tickets (ticketsAvailable > 0). If so, it decrements the ticketsAvailable count to indicate that a ticket has been booked by the current user. If no tickets are available, a message is displayed indicating the unavailability of tickets for the

current user.

After booking a ticket (or indicating the inability to do so), the thread pauses for one second (simulated by Thread.Sleep(1000)) to represent some time passing before the user cancels their ticket reservation.

Upon cancellation, the thread re-enters the critical section to increment the ticketsAvailable count, as the canceled ticket becomes available again for booking by other users.