Assignment 2

PRG 281

Daniel Mattheus Johannes Adendorff
577624

Scenario

You are required to provide answers to the following questions. These answers need to be documented in a word processing application like Microsoft Word, and later converted to a PDF for submission. You should submit a PDF file only. Do not zip the file. These questions will help you in properly understanding the basic concepts that are very important in this module. Your assignment should not exceed 6 pages including the cover page.

Answer for a:

1. Synchronization Mechanisms:

Choose appropriate synchronization mechanisms to control access to Object A's critical section. The most common mechanisms are locks and semaphores. These prevent multiple threads from accessing the critical section at the same time.

2. Testing and Debugging:

Thoroughly test the multi-threaded application under various scenarios to identify any race conditions, deadlocks, or performance bottlenecks. Tools like thread analyzers can help detect and diagnose the issues.

3. Monitor Performance:

Keep an eye on the application performance after implementing thread safety measures. If you overuse locks or improper locking strategies it can lead to performance struggleing due to increased contention between threads.

4. Locking Strategy:

Decide whether to use a simple lock for the entire critical section or a more fine-grained locking approach if Object A has multiple distinct resources or operations. Fine-grained locking can improve concurrency by allowing different threads to access different parts of the object concurrently.

5. Member Access Control:

Restrict access to Object A's internal data members by encapsulating them and providing controlled methods to interact with them. Make sure that any access or modification to these data members is done through synchronized methods.

Answer for b:

When multiple threads attempt to access and modify an object simultaneously, several issues can happen because of the lack of proper synchronization and coordination. These issues can lead to data corruption, inconsistent behavior, and unexpected outcomes. Here are some problems that may occur:

1. Race Conditions:

Race conditions happen when the outcome of a program depends on the relative timing of thread execution. In Object A a race condition might result in incorrect data being read or modified because of the threads interleaving their operations. For example, if two threads increment at the same time a counter in Object A the final count might not be what was expected.

2. Data Corruption:

Concurrent modification of shared data in Object A without proper synchronization can lead to data corruption. If multiple threads modify the same data simultaneously, the final state might be a combination of their changes, resulting in unexpected or incorrect values.

3. Inconsistent State:

Threads accessing Object A concurrently can lead to an inconsistent state of the object. If one thread is in the process of updating an object state while another thread reads it, the person reading it might see a partially updated state, leading to unexpected behaviour.

4. Deadlocks:

Deadlocks occur when two or more threads are unable to proceed because they're waiting for each other to release resources like locks. If one thread holds a lock that another thread needs the application can become stuck.

5. Livelocks:

Livelocks are situations where threads are trying to resolve a resource contention, but they're unable to make progress because their actions keep causing problems. This can lead to the threads being stuck in an endless loop of trying to resolve the contention.

Answer for c:

Thread Synchronization:

Thread synchronization is the process of coordinating the execution of multiple threads in a multi-threaded program to ensure correct and consistent behaviour. It involves controlling access to shared resources in a way that prevents race conditions, data corruption, and other related issues. Thread synchronization mechanisms ensure that only one thread can access a critical section of code or data at a time, thus maintaining data integrity and preventing conflicts.

Example of Thread Synchronization:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Thread Synchronization
{
    using System;
    using System.Threading;

    public class BankAccount
    {
        private double balance;
        private object lockObj = new object();

        public BankAccount(double initialBalance)
        {
            this.balance = initialBalance;
        }

        public void Withdraw(double amount)
        {
            lock (lockObj)
            {
                if (amount > 0 && balance >= amount)
                {

                    Thread.Sleep(100);

                    balance -= amount;
                    Console.WriteLine($"Withdrawal of ${amount} successful. New 
balance: ${balance}");
                }
                else
                {
                    Console.WriteLine("Insufficient funds for withdrawal.");
                }
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            BankAccount account = new BankAccount(420);

            Thread thread1 = new Thread(() => account.Withdraw(200));
            Thread thread2 = new Thread(() => account.Withdraw(300));
            thread1.Start();
            thread2.Start();

            thread1.Join();
            thread2.Join();

            Console.WriteLine("All withdraws completed.");

            Console.ReadLine();
        }
    }
}
```

Example explanation:

Because there are no comments in the code I will explain what I did.
I am defining the BankAccount class with a balance field and the object lockObj.
The Withdraw method is synchronized using the lock statement and the lockObj object.
This ensures that only one thread can execute this method at a time.
I created two threads that attempts to withdraw amounts concurrently from the same BankAccount instance.
I started both threads and used Join to ensure it completes before displaying the "All withdrawals completed." message.

Answer for d:

Using locks and thread-safe data structures are two different approaches to managing concurrent access to shared resources like Object A. Each approach has its own trade-offs and considerations. Let's explore the trade-offs for both:

Locks:

Pros:

Fine-grained Control: Locks provide precise control over which sections of code are synchronized. This can be useful if only specific parts of an object need protection.
Compatibility: Locks can be used with any data structure, including non-thread-safe ones.
Flexibility: Locks can handle complex synchronization scenarios where multiple resources need coordination.
Low-Level Control: Locks allow you to fine-tune synchronization strategies and manage aspects like fairness and re-entrancy.

Cons:

Complexity: Managing locks can be complex, especially in large and intricate systems. Locks introduce the potential for deadlocks and require careful design to avoid.
Performance Overhead: Locks can introduce contention and serialization, leading to performance degradation when multiple threads contend for the same lock.
Scalability: Locks might not scale well when the number of threads increases, as they can introduce bottlenecks.

Using Thread-Safe Data Structures:

Pros:

Simplified Implementation: Thread-safe data structures encapsulate synchronization logic internally, reducing the complexity of your code.
Reduced Risk of Deadlocks: Thread-safe data structures are designed to handle synchronization internally, reducing the risk of deadlocks compared to manual locking.
Performance: Well-designed thread-safe data structures can optimize synchronization to reduce contention and improve performance.
Scalability: Thread-safe data structures can be more scalable than using locks in scenarios with high thread counts.

Cons:

Limited Applicability: Thread-safe data structures are designed for specific use cases, and you might not find one that does exactly what you need.
Less Control: Thread-safe data structures might not allow you to fine-tune synchronization strategies or handle complex synchronization scenarios.
Compatibility: Switching to thread-safe data structures might require changes to your codebase if you were previously using non-thread-safe structures.

What to choose:

The choice between locks and thread-safe data structures depends on the complexity of your code, the performance requirements, and the level of control you need over synchronization. In modern programming, using higher-level abstractions like thread-safe data structures is often favoured due to how easy it is to use and potential for better performance in heavily concurrent scenarios.