The lock statement - ensure exclusive access to a shared resource

Article • 05/07/2024

The `lock` statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released. The `lock` statement ensures that at maximum only one thread executes its body at any time moment.

The `lock` statement takes the following form:

C#

```
lock (x)
{
    // Your code...
}
```

The variable `x` is an expression of System.Threading.Lock type, or a reference type. When `x` is known at compile-time to be of the type System.Threading.Lock, it's precisely equivalent to:

C#

```
using (x.EnterScope())
{
    // Your code...
}
```

The object returned by Lock.EnterScope() is a ref struct that includes a `Dispose()` method. The generated using statement ensures the scope is released even if an exception is thrown with the body of the `lock` statement.

Otherwise, the `lock` statement is precisely equivalent to:

C#

```
object __lockObj = x;
bool __lockWasTaken = false;
```

```
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Since the code uses a try-finally statement, the lock is released even if an exception is thrown within the body of a `lock` statement.

You can't use the await expression in the body of a `lock` statement.

Guidelines

Beginning with .NET 9 and C# 13, lock a dedicated object instance of the System.Threading.Lock type for best performance. In addition, the compiler issues a warning if a known `Lock` object is cast to another type and locked. If using an older version of .NET and C#, lock on a dedicated object instance that isn't used for another purpose. Avoid using the same lock object instance for different shared resources, as it might result in deadlock or lock contention. In particular, avoid using the following instances as lock objects:

- `this`, as callers might also lock `this`.
- Type instances, as they might be obtained by the typeof operator or reflection.
- string instances, including string literals, as they might be interned.

Hold a lock for as short time as possible to reduce lock contention.

Example

The following example defines an `Account` class that synchronizes access to its private `balance` field by locking on a dedicated `balanceLock` instance. Using the same instance for locking ensures that two different threads can't update the `balance` field by calling the `Debit` or `Credit` methods simultaneously. The sample uses C# 13 and the new `Lock` object. If you're using an older version of C# or an older .NET library, lock an instance of `object`.

C#

```
using System;
using System.Threading.Tasks;

public class Account
{
    // Use `object` in versions earlier than C# 13
    private readonly System.Threading.Lock _balanceLock = new();
    private decimal _balance;

    public Account(decimal initialBalance) => _balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit
amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (_balanceLock)
        {
            if (_balance >= amount)
            {
                _balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit
amount cannot be negative.");
        }

        lock (_balanceLock)
        {
            _balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (_balanceLock)
        {
```

```
            return _balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = [0, 2, -3, 6, -2, -1, 8, -5, 11, -6];
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

C# language specification

For more information, see The lock statement section of the C# language specification.

See also

- System.Threading.Monitor
- System.Threading.SpinLock

- System.Threading.Interlocked
- Overview of synchronization primitives
- Introduction to System.Threading.Channels