Semmle 1.21

Double-checked lock is not thread-safe

Created by Documentation team, last modified on Mar 28, 2019

Name: Double-checked lock is not thread-safe

Description: A repeated check on a non-volatile field is not thread-safe on some platforms, and could result in unexpected behavior.

ID: cs/unsafe-double-checked-lock

Kind: problem

Severity: error

Precision: medium

Query: UnsafeLazyInitialization.ql Expand source

Double-checked locking requires that the underlying field is `volatile`, otherwise the program can behave incorrectly when running in multiple threads, for example by computing the field twice.

Recommendation

There are several ways to make the code thread-safe:

1. Avoid double-checked locking, and simply perform everything within the lock statement.
2. Make the field volatile using the `volatile` keyword.
3. Use the `System.Lazy` class, which is guaranteed to be thread-safe. This can often lead to more elegant code.
4. Use `System.Threading.LazyInitializer`.

Example

The following code defines a property called `Name`, which calls the method `LoadNameFromDatabase` the first time that the property is read, and then caches the result. This code is efficient but will not work properly if the property is accessed from multiple threads, because `LoadNameFromDatabase` could be called several times.

```
string name;

public string Name
{
    get
    {
        // BAD: Not thread-safe
        if (name == null)
            name = LoadNameFromDatabase();
        return name;
    }
}
```

A common solution to this is *double-checked locking*, which checks whether the stored value is `null` before locking the mutex. This is efficient because it avoids a potentially expensive lock operation if a value has already been assigned to `name`.

```
string name;    // BAD: Not thread-safe

public string Name
{
    get
    {
        if (name == null)
        {
            lock (mutex)
            {
                if (name == null)
                    name = LoadNameFromDatabase();
            }
        }
        return name;
    }
}
```

However this code is incorrect because the field `name` isn't volatile, which could result in `name` being computed twice on some systems.

The first solution is to simply avoid double-checked locking (Recommendation 1):

```
string name;

public string Name
{
    get
    {
        lock (mutex)    // GOOD: Thread-safe
        {
            if (name == null)
                name = LoadNameFromDatabase();
            return name;
        }
    }
}
```

Another fix would be to make the field volatile (Recommendation 2):

```
volatile string name;    // GOOD: Thread-safe

public string Name
{
    get
    {
        if (name == null)
        {
            lock (mutex)
            {
                if (name == null)
                    name = LoadNameFromDatabase();
            }
        }
        return name;
    }
}
```

It may often be more elegant to use the class `System.Lazy`, which is automatically thread-safe (Recommendation 3):

```
Lazy<string> name;    // GOOD: Thread-safe

public Person()
{
    name = new Lazy<string>(LoadNameFromDatabase);
}

public string Name => name.Value;
```

References

- MSDN: Lazy<T> Class.
- MSDN: LazyInitializer.EnsureInitialized Method.
- MSDN: Implementing Singleton in C#.
- MSDN Magazine: The C# Memory Model in Theory and Practice.
- MSDN, C# Reference: volatile.
- Wikipedia: Double-checked locking.
- Common Weakness Enumeration: CWE-609.

correctness concurrency cwe cwe-609 alert problem