Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

 «Пермский национальный исследовательский политехнический университет»

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

Направление 09.03.04 – «Программная инженерия»

**ОТЧЕТ**

по лабораторной работе №1

на тему: «Race condition и Deadlock»

Выполнил: студент группы РИС-20-1б

Шумилов Л.С.

Проверил: доцент кафедры ИТАС

Щапов В.А.

Пермь, 2024

**Цель работы** – познакомиться с явлениями «Состояния гонки» и «Взаимной блокировки»

**Задание**:

1. Продемонстрировать ситуацию «Состояния гонки»;
2. Предложить решение данной проблемы;
3. Продемонстрировать ситуацию «Взаимной блокировки»;
4. Предложить решение данной проблемы.

**Краткие теоретические сведения**

Race Condition (Состояние гонки) — это ситуация, при которой выполнение нескольких операций зависит от их относительного порядка выполнения. В результате, может возникнуть непредсказуемое или неверное поведение системы. Такая ситуация возникает в многозадачных системах, когда несколько процессов или потоков конкурируют за общие ресурсы и могут изменять их состояние.

Deadlock (Взаимная блокировка) — это ситуация, при которой несколько процессов или потоков оказываются в состоянии ожидания, потому что каждый из них ожидает освобождения ресурса, который заблокирован другим процессом или потоком. В результате, ни один из процессов не может продолжить своё выполнение, и система оказывается в неработоспособном состоянии.

**Ход работы**

Для получения состояния гонки, необходимо запустить 2 потока в одно и то же время, которые обращаются к общему ресурсу. Сам факт одновременного обращения продемонстрирует работу Race Condition, как результат – состояние общего ресурса будет отличаться после каждого прогона и не совпадать с ожидаемым результатом.

На листинге 1 представлен код, который демонстрирует работу Race Condition.

Листинг 1. Race Condition

private int \_sharedCounter = 0;

public int SharedCounter => \_sharedCounter;

public object lockObject = new object();

private void NonLockedIncrement(object count)

{

var intCount = (int)count;

for (int i = 0; i < intCount; i++)

{

\_sharedCounter++;

}

}

public void RunNonLockedTwoThreadIncrement(int incrementCount)

{

var firstThread = new Thread(new ParameterizedThreadStart(NonLockedIncrement));

var secondThread = new Thread(new ParameterizedThreadStart(NonLockedIncrement));

firstThread.Start(incrementCount);

secondThread.Start(incrementCount);

firstThread.Join();

secondThread.Join();

}

Для проверки работы Race Condition, были реализованы юнит-тесты, которые в автоматическом режиме сравнивают ожидаемый и фактический результаты после нескольких прогонов. Код представлен на Листинге 3.

Для того, чтобы не допустить ситуацию «Состояния гонки» необходимо при обращении к общему ресурсу осуществлять блокировку самого ресурса, чтобы другие процессы «ждали» пока текущий ресурс не закончит обработку общего ресурса. На Листинге 2 представлен пример решения.

Листинг 2. Исправление Race Condition

private void LockedIncrement(object count)

{

var intCount = (int)count;

for (int i = 0; i < intCount; i++)

{

lock (lockObject)

{

\_sharedCounter++;

}

}

}

public void RunLockedThoThreadIncrement(int incrementCount)

{

var firstThread = new Thread(new ParameterizedThreadStart(LockedIncrement));

var secondThread = new Thread(new ParameterizedThreadStart(LockedIncrement));

firstThread.Start(incrementCount);

secondThread.Start(incrementCount);

firstThread.Join();

secondThread.Join();

}

Листинг 2. Юнит-тесты для проверки Race Condition и его исправления

using System;

using System.Linq;

using Xunit;

namespace XUnitTests

{

public class RaceConditionTesting

{

private void TestRaceCondition(bool useLock)

{

var processor = new SharedVariableProcessor();

var incrementCount = 1000000;

if (useLock)

{

processor.RunLockedThoThreadIncrement(incrementCount);

}

else

{

processor.RunNonLockedTwoThreadIncrement(incrementCount);

}

var expected = incrementCount \* 2;

Assert.Equal(expected, processor.SharedCounter);

}

[Fact] public void RaceCondition1() => TestRaceCondition(useLock: false);

[Fact] public void RaceCondition2() => TestRaceCondition(useLock: false);

[Fact] public void RaceCondition3() => TestRaceCondition(useLock: false);

[Fact] public void FixedCondition1() => TestRaceCondition(useLock: true);

[Fact] public void FixedCondition2() => TestRaceCondition(useLock: true);

[Fact] public void FixedCondition3() => TestRaceCondition(useLock: true);

}

}

На Рисунке 1. представлен результат выполнения тестов с Race Condition и его исправлением (зелёные тесты пройдены успешно, красные – показывают несоответствия вызванные Race Condition)

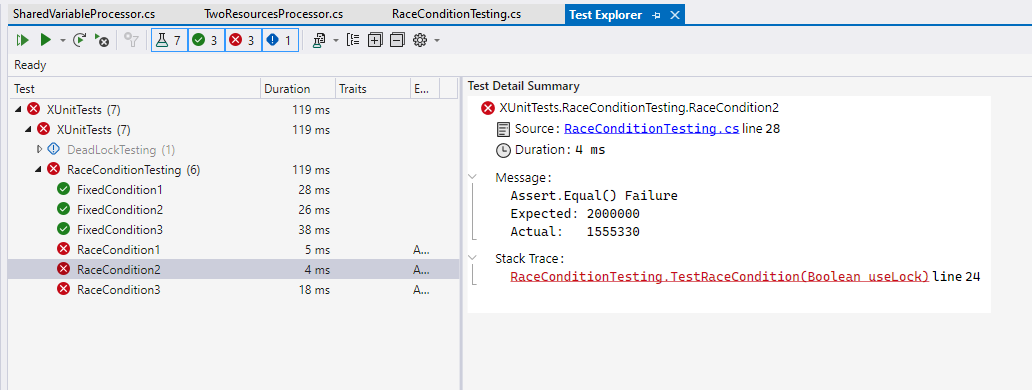


Рисунок 1. Тестирование Race Condition и его исправления

Для симуляции Deadlock необходимо сформировать 2 потока, которые обращаются к 2-м общим ресурсам так, чтобы первый поток ожидал второй, пока второй ожидает первый. Именно таким образом формируется взаимная блокировка неограниченная по времени. На Листинге 4 представлен пример.

Листинг 4. Deadlock

public IntResource FirstResource { get; }

public IntResource SecondResource { get; }

private void TransferFromFirstToSecond(object value)

{

var count = (int)value;

lock (FirstResource)

{

Thread.Sleep(1000);

lock (SecondResource)

{

FirstResource.Value -= count;

SecondResource.Value += count;

}

}

}

private void TransferFromSecondToFirst(object value)

{

var count = (int)value;

lock (SecondResource)

{

Thread.Sleep(1000);

lock (FirstResource)

{

FirstResource.Value += count;

SecondResource.Value -= count;

}

}

}

Как результат, при запуске теста, сам тест будет выполняться бесконечно долго – это и есть явление взаимной блокировки. Код будет «висеть» до тех пор, пока не будет выполнена принудительная остановка со стороны пользователя или операционной системы. Пример на Рисунке 2.

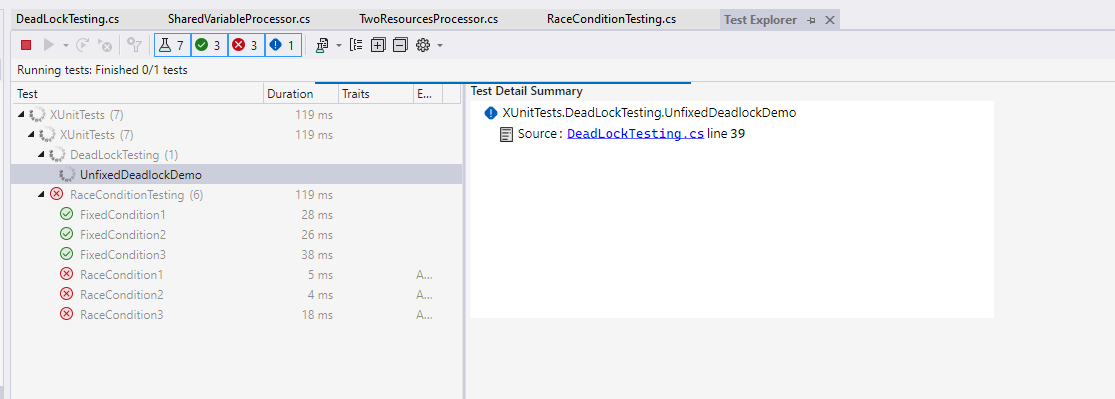


Рисунок 2. Демонстрация работы Deadlock

Исправление представляет собой изменённую последовательность блокировки ресурсов – оба ресурса блокируют 2 общих ресурса в одном порядке. Так удастся избежать взаимной блокировки, т.к. второй поток не сможет заблокировать второй ресурс, потому что первый уже заблокирован, и поток ждёт его освобождения. Пример на листинге 5.

Листинг 5. Исправленный Deadlock

private void NonDeadlockTransferFromSecondToFirst(object value)

{

var count = (int)value;

// решение - поменять lock местами, в таком случае взаимной блокировки не будет

lock (FirstResource)

{

Thread.Sleep(1000);

lock (SecondResource)

{

FirstResource.Value += count;

SecondResource.Value -= count;

}

}

}

На Рисунке 3 предоставлен пример успешного выполнения исправленного кода.

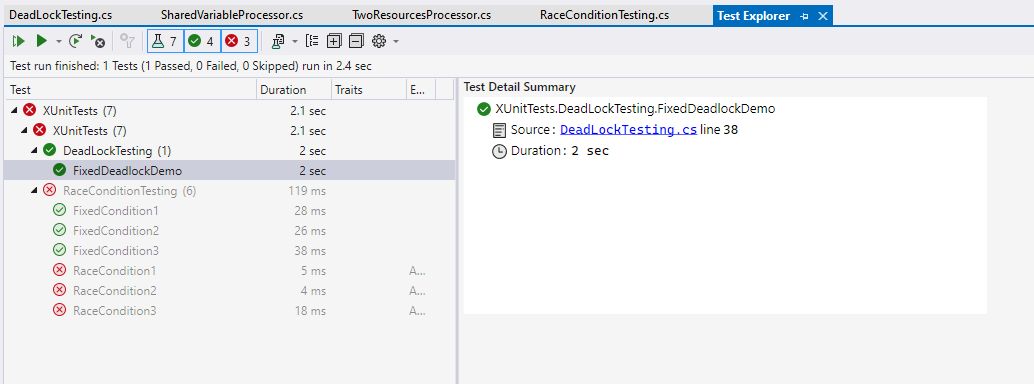


Рисунок 3. Исправленный Deadlock успешно выполнен