**https://intuit.ru/studies/courses/10554/1092/lecture/27095?page=4**

**Мягкие методы блокировки. Модель "Читатели и Писатели"**

Как уже говорилось, *блокировка* с использованием оператора lock полностью закрывает критическую секцию и находящиеся в ней ресурсы. Иногда требуется более гибкий подход, допускающий возможность одновременного чтения ресурса, но блокирующий попытку одновременной записи. Многие ситуации использования общих ресурсов укладываются в схему, получившую название "Читатели и Писатели". В этой модели пользователи общих ресурсов делятся на две категории - читателей и писателей. Когда один из писателей, захватывает *ресурс*, создавая новое *произведение*, то все остальные должны ждать в очереди, ожидая завершения работы. Когда же *ресурс* захвачен читателем, читающим *произведение*, то это не мешает другим читателям использовать этот *ресурс*, - читать *произведение* можно одновременно нескольким читателям.

**Класс ReaderWriterLockSlim**

Класс ReaderWriterLockSlim из пространства имен System.Threading предназначен для поддержки модели "Читатели и Писатели". Он позволяет выделить три группы пользователей ресурса - читателей, писателей и редакторов, - читающих, пишущих и редактирующих ресурс. Для каждой группы класс предлагает свой метод блокировки:

* EnterReadLock(). Если при выполнении этого метода есть очередь потоков со статусом Write, то поток становится в очередь потоков со статусом Read. Потоки из этой очереди смогут начать выполняться, только когда нет ждущих "писателей". Если очередь состоит только из потоков со статусом Upgradeable, то поток входит в критическую секцию, поскольку разрешается в критической секции одновременное присутствие потоков со статусом Read и одного потока со статусом Upgradeable.
* EnterWriteLock(). Если при выполнении этого метода есть очередь потоков со статусом Write, то поток становится в конец этой очереди. Если же очереди нет, то поток дожидается завершения работы потоков, находящихся в критической секции. После чего поток входит в критическую секцию и единолично работает с ресурсом.
* EnterUpgradeableReadLock(). Если при выполнении этого метода есть очередь потоков со статусом Write, то поток становится в очередь потоков со статусом Upgradeable. Если последняя очередь пуста, то поток будет первым в этой очереди. Первый поток из этой очереди может начать свою работу, когда очередь потоков со статусом Write пуста.

Для разблокирования критической секции применяют три соответствующих метода:

* ExitReadLock().
* ExitWriteLock().
* ExitUpgradeableReadLock().

Общая схема организации критической секции выглядит так:

Enter-метод

try {работа с ресурсом}

finally {Exit-метод }

Блок finally всегда будет выполняться, что гарантирует освобождение ресурса.

Поскольку из-за блокировки совместно используемых ресурсов можно войти в ситуацию клинча, то часто применяют более мягкую схему блокировки, заменяя Enter-методы их аналогами:

* bool TryEnterReadLock(int wait).
* bool TryEnterWriteLock(int wait).
* bool EnterUpgradeableReadLock(int wait).

В этом случае поток либо войдет в критическую секцию и начнет выполняться, возвращая в качестве результата Enter-метода значение true, либо выйдет из очереди по истечении времени ожидания wait, возвращая false. В любом случае зависание потока исключается.

У объектов класса ReaderWriterLockSlim есть достаточно много полезных свойств, позволяющих, например, анализировать состояние очередей. В частности, есть группа свойств Waiting, позволяющих узнать число потоков в очередях каждого типа.

**Пример. Разработка программного проекта**

Для демонстрации работы с классом ReaderWriterLockSlim построим пример, моделирующий разработку программного проекта. В нашей модели будем предполагать, что над проектом работают разработчики (писатели), создающие новые версии проекта, тестеры (редакторы), которые могут вносить исправления в проект, и программисты (читатели), использующие проект в собственных разработках. Все три группы пользователей могут работать параллельно.

Начнем построение нашей модели с создания класса Project:

/// <summary>

/// Программный проект, над которым работают

/// (developers), (testers), (programmers)

/// (писатели), (редакторы), (читатели)

/// </summary>

public class Project

{

string program = "";

string Program

{

get { return program; }

}

ReaderWriterLockSlim wer = new ReaderWriterLockSlim();

Этот класс описывает создаваемый проект. Сам проект представлен строкой текста program, вначале пустой, но которая будет меняться в результате действий разработчиков и тестеров. Программистам при каждом обращении будет доступно для чтения текущее значение этой строки.

Объект wer класса ReaderWriterLockSlim будет использоваться для организации блокировок при работе каждой из групп пользователей проекта.

Добавим теперь в наш класс метод, который будут использовать разработчики, создавшие новую версию проекта:

/// <summary>

///For Writers

/// </summary>

/// <param name="prog">новая версия программы</param>

public void WriteNewProgram(string prog)

{

wer.EnterWriteLock();

try

{

program = prog;

Thread.Sleep(10);

readers = wer.WaitingReadCount;

}

finally

{

wer.ExitWriteLock();

}

}

Параметр prog задает новую версию проекта. Вход в критическую секцию закрывается со статусом "write", что гарантирует отсутствие каких-либо других изменений переменной program до окончания работы метода и освобождения ресурса.

Аналогично устроен метод, предназначенный для тестеров. Основное отличие в том, каким статусом закрывается критическая секция:

/// <summary>

///For Editors

/// </summary>

/// <param name="prog">новое изменение программы</param>

public void EditNewProgram(string patch, ref int writers)

{

wer.EnterUpgradeableReadLock();

try

{

if(program != "")

program += patch;

Thread.Sleep(10);

if (wer.WaitingWriteCount > writers)

writers = wer.WaitingWriteCount;

}

finally

{

wer.ExitUpgradeableReadLock();

}

}

Методу передается два параметра. Первый из них - patch - отражает вносимые изменения в проект. Поскольку в задачу тестеров входит отслеживание процесса работы над проектом, то второй обновляемый параметр - writers - позволяет следить за числом писателей, стоящих в очереди. Он характеризует интенсивность работы разработчиков проекта.

Метод, предназначенный для читателей проекта, имеет вид:

/// <summary>

///For Readers

/// </summary>

/// <param name="prog">чтение программы</param>

public string ReadingProgram()

{

string p;

wer.EnterReadLock();

try

{

p = program;

Thread.Sleep(10);

}

finally

{

wer.ExitReadLock();

}

return p;

}

Построим теперь класс Model, моделирующий ход процесса работы над проектом. Вначале определим поля этого класса:

/// <summary>

/// Моделирование работы над программным проектом

/// </summary>

public class Model

{

int ndev, ntes, nprog, n;

string programs;

int writers;

Project project;

const int rep = 5;

const string nl = "\r\n";

Random rnd = new Random();

Thread[] threads;

Целочисленные переменные ndev, ntes, nprog задают число участников каждой группы, работающей над проектом, n - суммарное число участников. Переменная programs будет содержать некоторую историю разработки проекта, а writers - характеризует интенсивность разработки. Каждый раз, когда одному из участников необходимо выполнить свою работу будет создаваться поток, выполняющий эту задачу. Можно было бы иметь одну переменную класса Thread, каждый раз формируя ее новое значение. Но для гарантирования синхронизации удобнее иметь массив потоков threads.

Как обычно, для чтения закрытых полей создаются методы-свойства:

public string Programs

{

get { return programs;}

}

public int Writers

{

get { return writers; }

}

Конструктор класса позволяет формировать значения полей класса:

public Model(int nd, int nt, int np)

{

n = nd + nt + np;

this.ndev = nd;

this.ntes = nt;

this.nprog = np;

programs = "";

project = new Project();

threads = new Thread[n \* rep];

}

Основным методом класса является метод ModelWorkWithProject, задающий сценарий работы с проектом. Идея сценария такова: формируется некоторый цикл, на каждом шаге этого цикла происходит некоторое событие - один из участников работы над проектом выполняет очередное задание. Разработчик проекта задает новую версию, тестер вносит изменения, программист использует текущее состояние проекта. Для реализации задания создается новый поток, в котором выполняется соответствующий метод из класса Project. Вот как выглядит код этого метода:

/// <summary>

/// Реализация сценария работы с проектом

/// </summary>

public void ModelWorkWithProject()

{

int num = 0;

int version = 1;

int patch = 1;

for (int i = 0; i < n \* rep; i++)

{

//Генерация случайного события

num = rnd.Next(n);

if (num < ndev)

{

//Создается поток разработчиков

threads[i] = new Thread(() =>

{

project.WriteNewProgram(String.Format(

"version {0} from Devoleper {1} ", version, num));

});

threads[i].Start();

version++;

}

else

if (num < ndev + ntes)

{

//Создается поток тестеров

threads[i] = new Thread(() =>

{

project.EditNewProgram(String.Format(

" patch {0} from Tester {1}", patch, num),

ref writers);

});

threads[i].Start();

patch++;

}

else

{

//Создается поток программистов

threads[i] = new Thread(() =>

{

programs += nl + project.ReadingProgram();

});

threads[i].Start();

}

Thread.Sleep(5);

}

for (int i = 0; i < n \* rep; i++)

{

threads[i].Join();

}

}

При создании потоков используются анонимные методы, в которых объект project вызывает соответствующий метод класса Project. В зависимости от того, чье задание нужно выполнить - разработчика, тестера или программиста, - создаваемый поток будет реализовывать разную стратегию работы с общим ресурсом.

Заметьте, программисты, читающие изменяющееся во времени состояние проекта, сохраняют эту информацию в переменной programs, играющую роль своеобразного архива. Тестеры в переменной writers сохраняют информацию о максимальном числе писателей, стоящих в очереди на запись новой версии.

Классы Project и Model позволили решить главные задачи. Теперь осталось сделать заключительный шаг - в интерфейсном классе запустить сценарий и вывести результаты работы. Вот как выглядит процедура Main в консольном проекте, инициирующая запуск нашей модели:

static void Main(string[] args)

{

Model model = new Model(2, 3, 5);

model.ModelWorkWithProject();

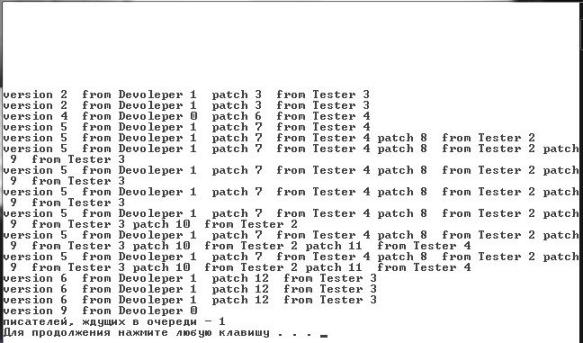
Console.WriteLine(model.Programs);

Console.WriteLine("писателей, ждущих в очереди - "

+ model.Writers);

}

Создается объект model, моделирующий работу над проектом двух разработчиков, трех тестеров и пятерых программистов. Запускается соответствующий сценарий работы, в котором выполняется 50 заданий случайным образом распределенных между всеми участниками проекта. Все задания выполняются в отдельных потоках. При работе потоков применяется мягкая блокировка, при которой читатели (программисты в нашем примере) не блокируют друг друга. Вот как выглядят результаты одного сеанса работы нашего приложения:

[](https://intuit.ru/EDI/28_11_18_2/1543357168-6234/tutorial/1163/objects/5/files/05_05.jpg)

[увеличить изображение](https://intuit.ru/EDI/28_11_18_2/1543357168-6234/tutorial/1163/objects/5/files/05_05.jpg)  
**Рис. 5.5.**Результаты моделирования задачи о программном проекте

Поясним результаты, приведенные на [рис. 5.5](https://intuit.ru/studies/courses/10554/1092/lecture/27095?page=4#image.5.5). Число строк соответствует числу заданий программистов. Максимальный номер версии показывает, сколько раз разработчики обновляли проект. Максимальный номер заплатки говорит о работе тестеров. Суммарно эти три числа должны быть меньше общего числа заданий, равного 50, поскольку некоторые этапы проекта не фиксировались программистами. Например, отсутствуют данные о работе программистов с третьей и седьмой версиях проекта. Одинаковые записи говорят о том, что с одной и той же текущей версией работали несколько программистов, Это является косвенным свидетельством одновременной работы потоков чтения. Из двух писателей одному приходилось стоять в очереди. В целом результаты соответствуют ожидаемым результатам работы. Играя со временами задержки работы потоков, можно получить и другие интересные результаты. Можно считать, что модель свою задачу выполнила.