**№ 15 Работа с потоками выполнения**

Курносенко Софья

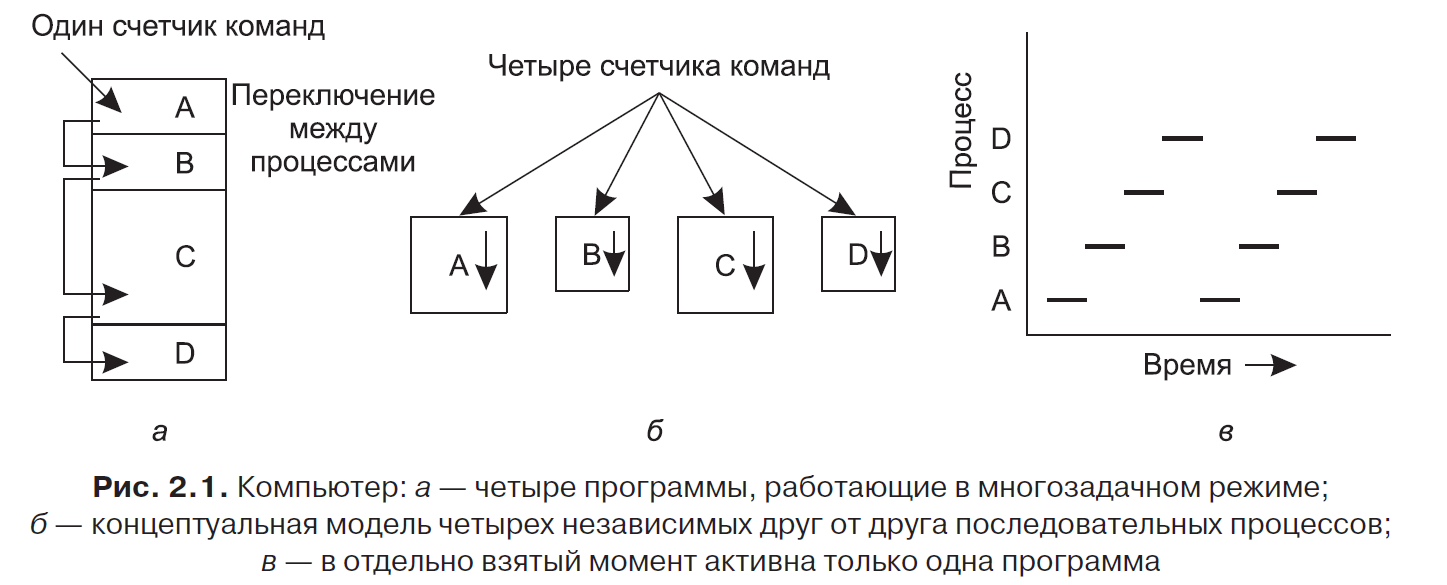
### Процесс

Процессы — это одна из самых старых и наиболее важных абстракций, присущих операционной системе. Они поддерживают возможность осуществления (псевдо) параллельных операций даже при наличии всего одного центрального процессора. Они превращают один центральный процессор в несколько виртуальных.

Процесс (process) - это объект, который создается операционной системой, когда пользователь запускает приложение. Процессу выделяется отдельное адресное пространство, причем это пространство физически недоступно для других процессов. Процесс может работать с файлами или с каналами связи локальной или глобальной сети. Когда вы запускаете текстовый процессор или программу калькулятора, вы создаете новый процесс. Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

В любой многозадачной системе центральный процессор быстро переключается между процессами, предоставляя каждому из них десятки или сотни миллисекунд. При этом, хотя в каждый конкретный момент времени центральный процессор работает только с одним процессом, в течение 1 секунды он может успеть поработать с несколькими из них, создавая иллюзию параллельной работы. Иногда в этом случае говорят о **псевдопараллелизме** в отличие от настоящего аппаратного параллелизма

в **многопроцессорных** системах (у которых имеется не менее двух центральных процессоров, использующих одну и ту же физическую память).



Разница между процессом и программой довольно тонкая, но весьма существенная. Здесь нам, наверное, поможет какая-нибудь аналогия. Представим себе программиста, решившего заняться кулинарией и испечь пирог на день рождения дочери. У него есть рецепт пирога, а на кухне есть все ингредиенты: мука, яйца, сахар, ванильный экстракт и т. д. В данной аналогии рецепт — это программа (то есть алгоритм, выраженный в не-

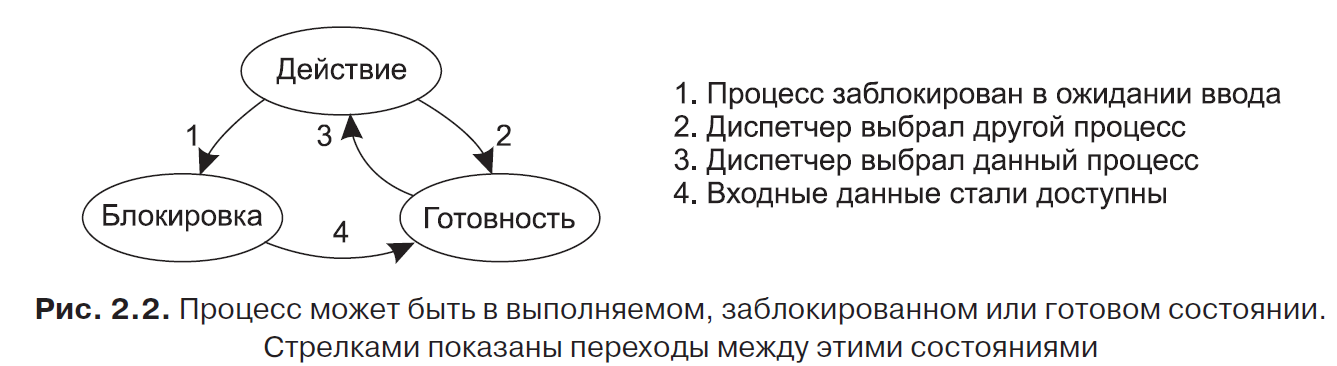
кой удобной форме записи), программист — это центральный процессор, а ингредиенты пирога — это входные данные. Процесс — это действия, состоящие из чтения рецепта нашим кулинаром, выбора ингредиентов и выпечки пирога. Теперь представим, что на кухню вбегает сын программиста и кричит, что его ужалила пчела. Программист записывает, на каком месте рецепта он остановился (сохраняется состояние текущего процесса), достает книгу советов по оказанию первой помощи и приступает к выполнению изложенных в ней инструкций. Перед нами процессор, переключенный с одного процесса (выпечки) на другой процесс, имеющий более высокую степень приоритета (оказание медицинской помощи), и у каждого из процессов есть своя программа (рецепт против справочника по оказанию первой помощи). После извлечения пчелиного жала программист возвращается к пирогу, продолжая выполнять действия с того места, на котором остановился. Ключевая идея здесь в том, что процесс — это своего рода действия. У него есть программа, входные и выходные данные и состояние.

Один процессор может совместно использоваться несколькими процессами в соответствии с неким алгоритмом планирования, который используется для определения того, когда остановить один процесс и обслужить другой. В отличие от процесса программа может быть сохранена на диске

и вообще ничего не делать.

Стоит отметить, что если программа запущена дважды, то считается, что ею заняты два процесса. Например, зачастую возможно дважды запустить текстовый процессор или одновременно распечатать два файла, если одновременно доступны два принтера. Тот факт, что два работающих процесса запущены от одной и той же программы, во внимание не принимается, поскольку это два разных процесса. Операционная система может позволить им использовать общий код, поэтому в памяти будет присутствовать только одна копия этого кода, но это чисто техническая деталь, не меняющая концептуальную ситуацию, касающуюся двух работающих процессов.

**Состояния процессов**



Как показано на рисунке, между этими тремя состояниями могут быть четыре перехода.

Переход *1* происходит в том случае, если операционная система определит, что процесс в данный момент выполняться не может.

Переходы *2* и *3* вызываются планировщиком процессов, который является частью операционной системы, без какого-либо оповещения самого процесса. Переход *2* происходит, когда планировщик решит, что выполняемый процесс продвинулся достаточно далеко и настало время позволить другому процессу получить долю рабочего времени центрального процессора. Переход *3* происходит, когда все другие процессы получили причитающуюся им долю времени и настал момент предоставить центральный процессор первому процессу для возобновления его выполнения. Вопрос планирования, то есть решение, какой именно процесс, когда и сколько времени должен выполняться, играет весьма важную роль и будет рассмотрен в этой главе чуть позже. В попытках сбалансировать конкурирующие требования соблюдения эффективности системы

в целом и справедливого отношения к отдельному процессу было изобретено множество алгоритмов.

Переход *4* осуществляется в том случае, если происходит внешнее событие, ожидавшееся процессом (к примеру, поступление входных данных). Если к этому моменту нет других выполняемых процессов, будет вызван переход *3* и процесс возобновится. В противном случае ему придется немного подождать в состоянии готовности, пока не станет доступен центральный процессор и не придет его очередь.

**Реализация процессов**

Для реализации модели процессов операционная система ведет таблицу (состоящую из массива структур), называемую **таблицей процессов**, в которой каждая запись соответствует какому-нибудь процессу. (Ряд авторов называют эти записи **блоками управления процессом**.) Эти записи содержат важную информацию о состоянии процесса, включая счетчик команд, указатель стека, распределение памяти, состояние открытых

им файлов, его учетную и планировочную информацию и все остальное, касающееся процесса, что должно быть сохранено, когда процесс переключается из состояния *выполнения* в состояние *готовности* или *блокировки*, чтобы позже он мог возобновить выполнение, как будто никогда не останавливался.

**Моделирование режима многозадачности**

Режим многозадачности позволяет использовать центральный процессор более рационально. При грубой прикидке, если для среднестатистического процесса вычисления занимают лишь 20 % времени его пребывания в памяти, то при пяти одновременно находящихся в памяти процессах центральный процессор будет загружен постоянно. Но в эту модель заложен абсолютно нереальный оптимизм, поскольку в ней заведомо

предполагается, что все пять процессов никогда не будут одновременно находиться в ожидании окончания какого-нибудь процесса ввода-вывода.

### Поток

Для каждого процесса операционная система создает один главный поток (thread), который является потоком выполняющихся по очереди команд центрального процессора. При необходимости главный поток может создавать другие потоки, пользуясь для этого программным интерфейсом операционной системы.

Все потоки, созданные процессом, выполняются в адресном пространстве этого процесса и имеют доступ к ресурсам процесса. Однако поток одного процесса не имеет никакого доступа к ресурсам потока другого процесса, так как они работают в разных адресных пространствах. При необходимости организации взаимодействия между процессами или потоками, принадлежащими разным процессам, следует пользоваться системными средствами, специально предназначенными для этого.

**Применение потоков**

Зачем нам нужна какая-то разновидность процесса внутри самого процесса? Необходимость в подобных мини-процессах, называемых **потоками**, обусловливается целым рядом причин. Рассмотрим некоторые из них. Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько

последовательных потоков, выполняемых в квазипараллельном режиме.

Мы уже сталкивались с подобными аргументами. Именно они использовались в поддержку создания процессов. Вместо того чтобы думать о прерываниях, таймерах и контекстных переключателях, мы можем думать о параллельных процессах. Но только теперь, рассматривая потоки, мы добавляем новый элемент: возможность использования

параллельными процессами единого адресного пространства и всех имеющихся данных. Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов (с их раздельными адресными пространствами).

Вторым аргументом в пользу потоков является легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами. Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов. Это свойство особенно пригодится, когда потребуется быстро и динамично изменять количество потоков.

Третий аргумент в пользу потоков также касается производительности. Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.

И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений.

Понять, в чем состоит польза от применения потоков, проще всего на конкретных примерах. Рассмотрим в качестве первого примера текстовый процессор. Обычно эти программы отображают создаваемый документ на экране в том виде, в каком он будет выводиться на печать. В частности, все концы строк и концы страниц находятся именно там, где они в результате и появятся на бумаге, чтобы пользователь мог при необходимости их проверить и подправить (например, убрать на странице начальные

и конечные висячие строки, имеющие неэстетичный вид).

Предположим, что пользователь пишет какую-то книгу.

Теперь представим себе, что происходит, когда пользователь вдруг удаляет одно предложение на первой странице 800-страничного документа. Теперь, проверив внесенные изменения, он хочет внести еще одну поправку на 600-й странице и набирает команду, предписывающую текстовому процессору перейти на эту страницу (возможно, посредством поиска фразы, которая только там и встречается). Теперь текстовый процессор вынужден немедленно переформатировать всю книгу вплоть до 600-й страницы, поскольку он не знает, какой будет первая строка на 600-й странице, пока не обработает все предыдущие страницы. Перед отображением 600-й страницы может произойти существенная задержка, вызывающая недовольство пользователя.

И здесь на помощь могут прийти потоки. Предположим, что текстовый процессор написан как двухпоточная программа. Один из потоков взаимодействует с пользователем, а другой занимается переформатированием в фоновом режиме. Как только предложение с первой страницы будет удалено, поток, отвечающий за взаимодействие

с пользователем, приказывает потоку, отвечающему за формат, переформатировать всю книгу. Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью выполняет вычисления. Если немного повезет, то пере-форматирование закончится как раз перед тем, как пользователь запросит просмотр

600-й страницы, которая тут же сможет быть отображена.

Ну раз уж начали, то почему бы не добавить и третий поток? Многие текстовые процессоры обладают свойством автоматического сохранения всего файла на диск каждые несколько минут, чтобы уберечь пользователя от утраты его дневной работы в случае программных или системных сбоев или отключения электропитания.

Первый поток занят только взаимодействием с пользователем. Второй поток по необходимости занимается переформатированием документа. А третий поток периодически сбрасывает содержимое ОЗУ на диск.

Три потока вместо трех процессов используют общую память, таким образом, все они имеют доступ к редактируемому документу. При использовании трех процессов такое было бы невозможно.

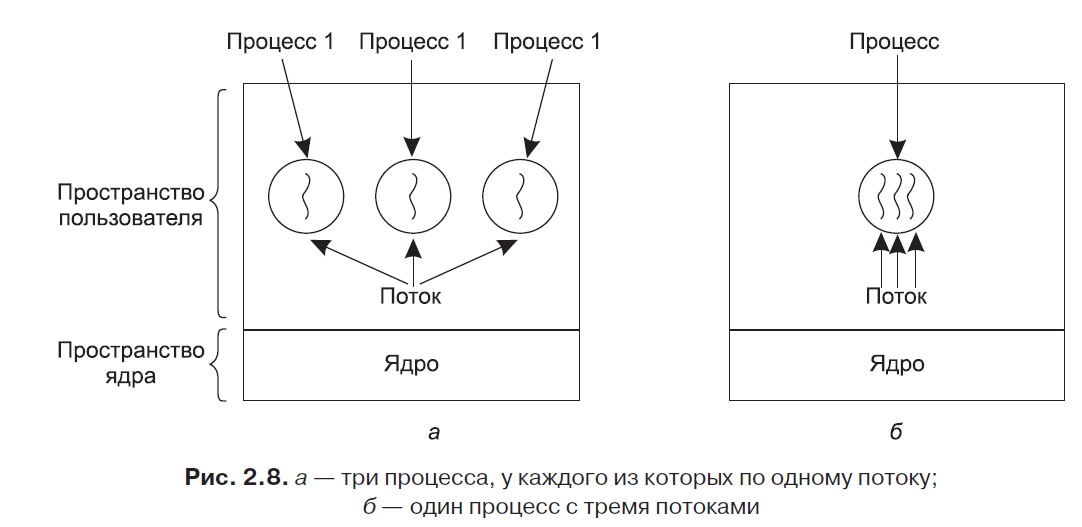
**Классическая модель потоков**

Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса. Наличие нескольких потоков, выполняемых параллельно в рамках одного процесса, является аналогией наличия нескольких процессов, выполняемых параллельно на одном компьютере. В первом случае потоки используют единое адресное пространство и другие ресурсы. А в последнем случае процессы используют общую физическую память, диски, принтеры и другие ресурсы. Поскольку потоки обладают

некоторыми свойствами процессов, их иногда называют **облегченными процессами**.

Термин «**многопоточный режим**» также используется для описания ситуации, при которой допускается работа нескольких потоков в одном и том же процессе.

На рис. 2.8, *а* показаны три традиционных процесса. У каждого из них имеется собственное адресное пространство и единственный поток управления. В отличие от этого, на рис. 2.8, *б* показан один процесс, имеющий три потока управления. Хотя в обоих случаях у нас имеется три потока, на рис. 2.8, *а* каждый из них работает в собственном адресном пространстве, а на рис. 2.8, *б* все три потока используют общее адресное пространство.

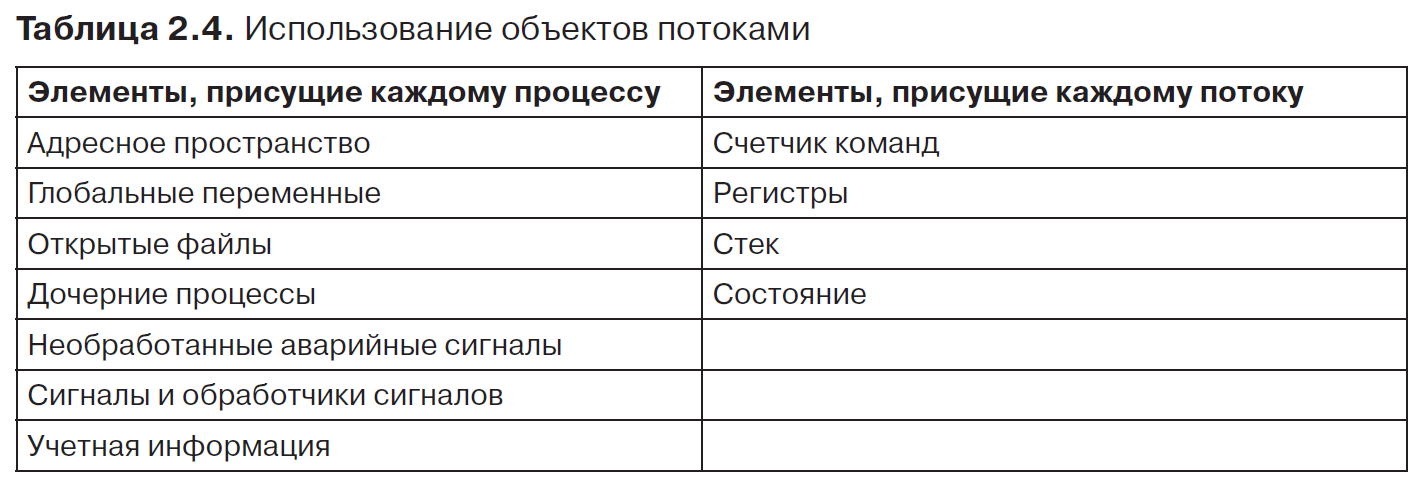


Когда многопоточный процесс выполняется на однопроцессорной системе, потоки выполняются, сменяя друг друга. Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно используют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства процесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости. В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут враждовать друг с другом, один процесс всегда принадлежит одному и тому же пользователю, который, по-видимому, и создал несколько потоков для их

совместной работы, а не для вражды.

В дополнение к использованию общего адресного пространства все потоки, как показано в табл. 2.4, могут совместно использовать одни и те же открытые файлы, дочерние процессы, ожидаемые и обычные сигналы и т. п.

Поэтому структура, показанная на рис. 2.8, *а*, может использоваться, когда все три процесса фактически не зависят друг от друга, а структура, показанная на рис. 2.8, *б*, может применяться, когда три потока фактически являются частью одного и того же задания и активно и тесно сотрудничают друг с другом.



**Процессы в C#**

# Процессы и домены приложения

При запуске приложения операционная система создает для него отдельный процесс, которому выделяется определённое адресное пространство в памяти и который изолирован от других процессов. Процесс может иметь несколько потоков. Как минимум, процесс содержит один - главный поток. В приложении на C# точкой входа в программу является метод Main. Вызов этого метода автоматически создает главный поток. А из главного потока могут запускаться вторичные потоки.

В .NET процесс представлен классом **Process** из пространства имен **System.Diagnostics**. Этот класс позволяет управлять уже запущенными процессами, а также запускать новые. В данном классе определено ряд свойств и методов, позволяющих получать информацию о процессах и управлять ими.

Мы можем вызвать методы:

* Метод **GetProcesses()**: возвращающий массив всех запущенных процессов
* Метод **GetProcessesByName()**: возвращает процессы по его имени. Так как можно запустить несколько копий одного приложения, то возвращает массив
* Метод **Start()**: запускает новый процесс

Примеры получения процессов:

using System;

using System.Diagnostics;

using System.ComponentModel;

namespace MyProcessSample

{

class MyProcess

{

void BindToRunningProcesses()

{

// Get the current process.

Process currentProcess = Process.GetCurrentProcess();

// Get all processes running on the local computer.

Process[] localAll = Process.GetProcesses();

// Get all instances of Notepad running on the local computer.

// This will return an empty array if notepad isn't running.

Process[] localByName = Process.GetProcessesByName("notepad");

// Get a process on the local computer, using the process id.

// This will throw an exception if there is no such process.

Process localById = Process.GetProcessById(1);

// Get processes running on a remote computer. Note that this

// and all the following calls will timeout and throw an exception

// if "myComputer" and 169.0.0.0 do not exist on your local network.

// Get all processes on a remote computer.

Process[] remoteAll = Process.GetProcesses("myComputer");

// Get all instances of Notepad running on the specific computer, using machine name.

Process[] remoteByName = Process.GetProcessesByName("notepad", "myComputer");

// Get all instances of Notepad running on the specific computer, using IP address.

Process[] ipByName = Process.GetProcessesByName("notepad", "169.0.0.0");

// Get a process on a remote computer, using the process id and machine name.

Process remoteById = Process.GetProcessById(2345, "myComputer");

}

static void Main()

{

MyProcess myProcess = new MyProcess();

myProcess.BindToRunningProcesses();

}

}

}

Пример запуска нового процесса:

using System.Diagnostics;

namespace MyProcessSample

{

class Program

{

static void Main()

{

// открываем текстовый файл

Process.Start("D://my.txt");

// запускаем Notepad

Process.Start("C://Program Files (x86)//Notepad++//notepad++.exe");

// запускаем Notepad с выбранным файлом

Process.Start("C://Program Files (x86)//Notepad++//notepad++.exe", "D://contract.txt");

}

}

}

Методы Getprocesses, GetProcessesByName и другие возвращают массив запущенных процессов. Мы можем обратиться к отдельным элементам массивов и получить значения таких свойств как:

* Свойство **Handle**: возвращает дескриптор процесса
* Свойство **Id**: получает уникальный идентификатор процесса в рамках текущего сеанса ОС
* Свойство **MachineName**: возвращает имя компьютера, на котором запущен процесс
* Свойство **Modules**: получает доступ к коллекции ProcessModuleCollection, которая хранит набор модулей (файлов dll и exe), загруженных в рамках данного процесса
* Свойство **ProcessName**: возвращает имя процесса, которое нередко совпадает с именем приложения
* Свойство **StartTime**: возвращает время, когда процесс был запущен
* Свойство **VirtualMemorySize64**: возвращает объем памяти, который выделен для данного процесса
* Метод **CloseMainWindow()**: закрывает окно процесса, который имеет графический интерфейс
* Метод **Kill()**: останавливает процесс

### Потоки процесса

Свойство Threads представляет коллекцию потоков процесса - объект ProcessThreadCollection, каждый поток в которой является объектом **ProcessThread**. В данном классе можно выделить следующие свойства:

* **CurrentPriority**: возвращает текущий приоритет потока
* **Id**: идентификатор потока
* **IdealProcessor**: позволяет установить процессор для обработки потока
* **PriorityLevel**: уровень приоритета потока
* **StartAddress**: адрес в памяти функции, запустившей поток
* **StartTime**: время запуска потока

Например, получим все потоки процесса Visual Studio:

using System.Diagnostics;

using System;

namespace MyProcessSample

{

class Program

{

static void Main()

{

Process proc = Process.GetProcessesByName("devenv")[0];

ProcessThreadCollection processThreads = proc.Threads;

foreach (ProcessThread thread in processThreads)

{

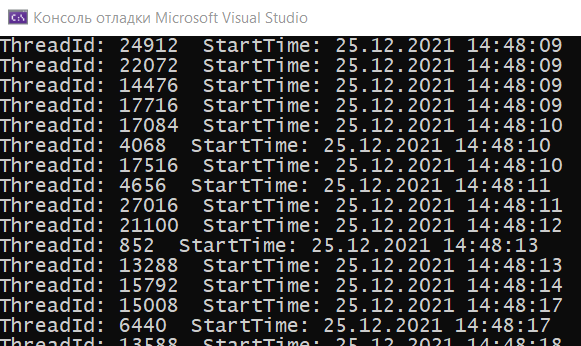
Console.WriteLine($"ThreadId: {thread.Id} StartTime: {thread.StartTime}");

}

}

}

}



### Модули процесса

Одно приложение может использовать набор различных сторонних библиотек и модулей. Для их получения класс Prosess имеет свойство Modules, которое представляет объект ProcessModuleCollection. Каждый отдельный модуль представлен классом **ProcessModule**, у которого можно выделить следующие свойства:

* **BaseAddress**: адрес модуля в памяти
* **FileName**: полный путь к файлу модуля
* **EntryPointAddress**: адрес функции в памяти, которая запустила модуль
* **ModuleName**: название модуля (краткое имя файла)
* **ModuleMemorySize**: возвращает объем памяти, необходимый для загрузки модуля

Получим все модули, используемые Visual Studio:

Process proc = Process.GetProcessesByName("devenv")[0];

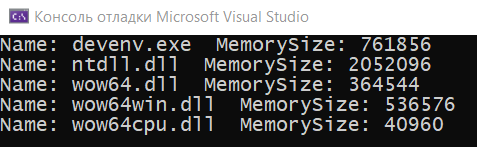
ProcessModuleCollection modules = proc.Modules;

foreach(ProcessModule module in modules)

{

    Console.WriteLine($"Name: {module.ModuleName}  MemorySize: {module.ModuleMemorySize}");

}



### Запуск нового процесса

С помощью статического метода **Process.Start()** можно запустить новый процесс. Например:

using System.Diagnostics;

namespace MyProcessSample

{

class Program

{

static void Main()

{

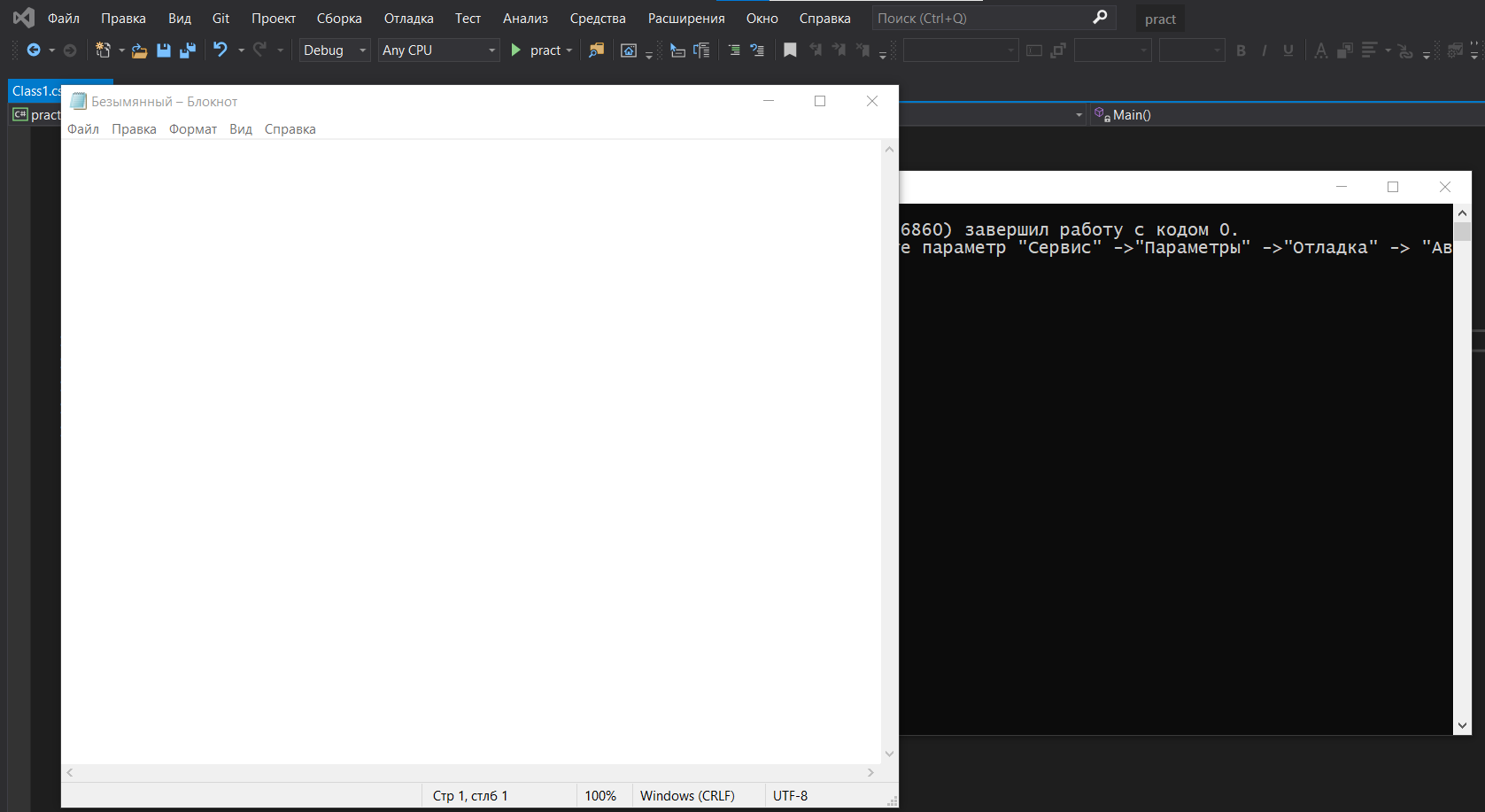
// обращение к исполняемой программе

Process.Start("notepad");

}

}

}



Чтобы отделить настройку параметров запуска от самого запуска можно использовать класс **ProcessStartInfo**:

using System.Diagnostics;

namespace MyProcessSample

{

class Program

{

static void Main()

{

ProcessStartInfo procInfo = new ProcessStartInfo();

// исполняемый файл программы - браузер хром

procInfo.FileName = "C://Program Files (x86)//Google//Chrome//Application//chrome.exe";

// аргументы запуска - адрес интернет-ресурса

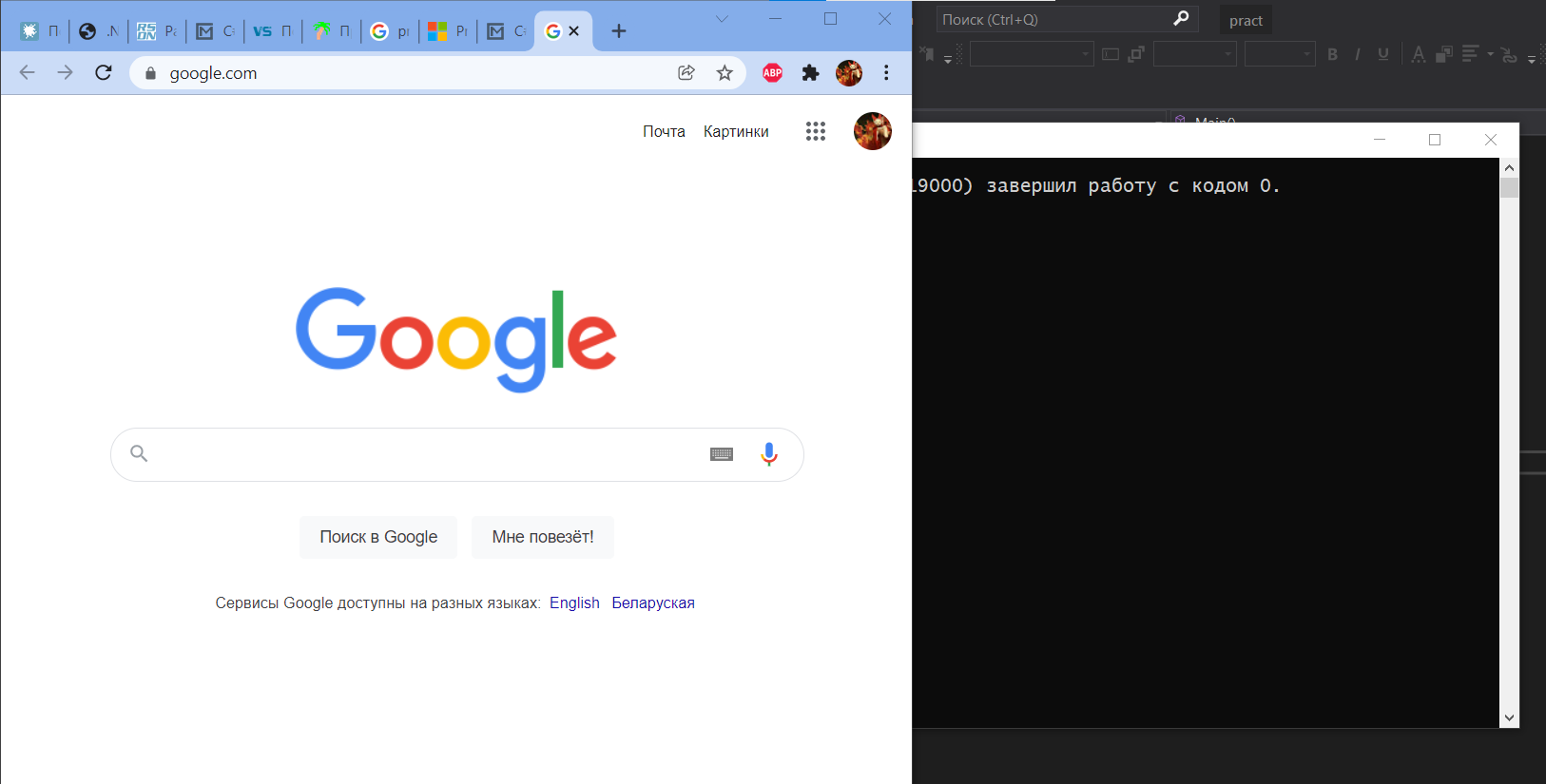
procInfo.Arguments = "http://google.com";

Process.Start(procInfo);

}

}

}



## Домены приложений

В .NET исполняемые файлы не обслуживаются прямо внутри процесса Windows, как это происходит в случае традиционных неуправляемых приложений. Вместо этого они обслуживаются в отдельном логическом разделе внутри процесса, который называется **доменом приложения** (Application Domain — AppDomain). В единственном процессе может содержаться несколько доменов приложений, каждый из которых обслуживает свой исполняемый файл .NET. Такое дополнительное подразделение традиционного процесса Windows предоставляет ряд преимуществ, главные из которых описаны ниже:

→ Домены приложений играют ключевую роль в обеспечении нейтральности платформы .NET по отношению к операционной системе из-за того что такое логическое разделение стирает отличия в способе представления загружаемого исполняемого файла лежащей в основе операционной системой.

→ Домены приложений являются гораздо менее дорогостоящими в плане потребления вычислительных ресурсов и памяти по сравнению с полноценными процессами. Благодаря этому CLR-среде удается загружать и выгружать домены приложений намного быстрее, чем формальные процессы, и тем самым значительно улучшать масштабируемость серверных приложений.

→ Домены приложений обеспечивают более глубокий уровень изоляции для обслуживания загружаемого приложения. В случае выхода из строя какого-то одного домена приложения внутри процесса, остальные домены приложений все равно остаются работоспособными.

**Класс System.AppDomain**

Платформа .NET позволяет программно осуществлять мониторинг доменов приложений, создавать новые домены приложений (или выгружать их) во время выполнения, загружать в домены приложений различные сборки и решать целый ряд других задач с применением класса AppDomain из пространства имен System, которое находится в сборке mscorlib.dll. Ниже перечислены наиболее полезные методы этого класса:

**CreateDomain**()

Этот статический метод позволяет создавать новый домен приложения в текущем процессе

**CreateInstance**()

Этот метод позволяет создавать экземпляр типа из внешней сборки после загрузки соответствующей сборки в вызывающий домен приложения

**ExecuteAssembly**()

Этот метод позволяет запускать сборку \*.ехе внутри домена приложения за счет предоставления имени ее файла

**GetAssemblies**()

Этот метод позволяет узнать, какие сборки .NET были загружены в данный домен приложения (двоичные файлы СОМ и С игнорируются)

**GetCurrentThreadId**()

Этот статический метод возвращает идентификатор потока, который является активным в текущем домене приложения

**Load**()

Этот метод применяется для динамической загрузки сборки в текущий домен приложения

**Unload**()

Этот статический метод позволяет выгрузить определенный домен приложения из конкретного процесса

Платформа .NET не позволяет производить выгрузку конкретной сборки из памяти. Единственным способом для осуществления выгрузки библиотек программным образом является разрушение обслуживающего домена приложения с помощью метода Unload().

Кроме того, класс AppDomain имеет свойства и поддерживает набор событий, которые могут быть полезны для проведения мониторинга за каким-то доменом приложения. Наиболее интересные свойства и события такого рода кратко описаны ниже:

**BaseDirectory**

Позволяет извлечь путь к каталогу, который преобразователь адресов использует для поиска сборок

**CurrentDomain**

Представляет собой статическое свойство и позволяет узнать домен приложения, используемый для выполняющегося в текущий момент потока

**FriendlyName**

Позволяет получить дружественное имя текущего домена приложения

MonitoringlsEnabled

Позволяет получить или установить значение, указывающее, должна ли работать функция мониторинга за использованием ресурсов ЦП и памяти для текущего процесса. После включения функции мониторинга для процесса отключить ее нельзя

**SetupInformation**

Позволяет извлечь детали конфигурации определенного домена приложения, которые предоставляются в виде объекта AppDomainSetup



**Пример "Информация о домене приложения" (имя текущего домена, перечисляем сборки)**

using System;

using System.Reflection;

namespace Laba15

{

partial class Program

{

static void Main(string[] args)

{

AppDomain curDom = AppDomain.CurrentDomain;

string curDomInfo = $"Имя домена приложения: {curDom.FriendlyName}\n" +

$"Каталог домена: {curDom.BaseDirectory}\n" +

$"Сборки домена:\n";

Assembly[] assemblies = curDom.GetAssemblies();

foreach(Assembly asm in assemblies)

{

curDomInfo += asm.GetName().Name + "\n";

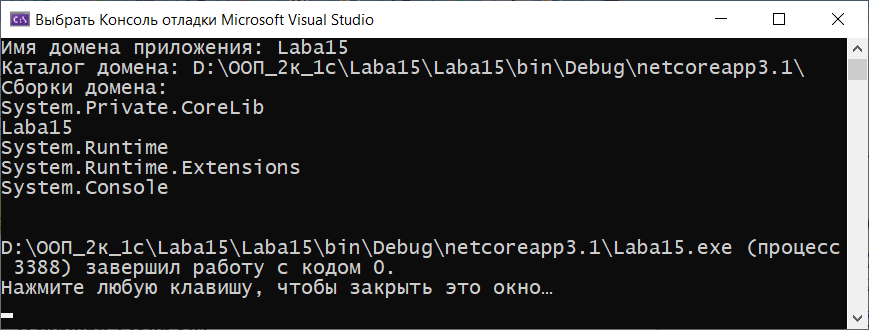
}

Console.WriteLine(curDomInfo);

}

}

}



**Попытка создать домен:**

using System;

namespace ConsoleApplication1

{

class Program

{

static void Main()

{

MakeNewDomain();

}

static void MakeNewDomain()

{

// Создадим новый домен приложения

AppDomain newD = AppDomain.CreateDomain("ProfessorWebAppDomain");

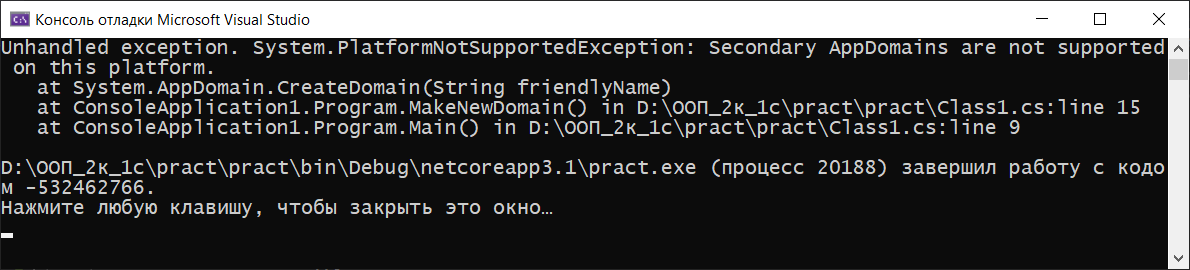
// Уничтожение домена приложения

AppDomain.Unload(newD);

}

}

}



Короче, в моей версии .NET создавать домены нельзя.

# Многопоточность

## Введение в многопоточность. Класс Thread

Одним из ключевых аспектов в современном программировании является **многопоточность**. Ключевым понятием при работе с многопоточностью является поток. Поток представляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности.

# Что такое потоки в C#

Если говорить простым языком, то **поток** – это некая независимая последовательность инструкций для выполнения того или иного действия в программе. В одном конкретном потоке выполняется одна конкретная последовательность действий.  
Совокупность таких потоков, выполняемых в программе параллельно называется **многопоточностью** программы.

Следует также запомнить, что *в действительности потоки выполняются всё-таки не совсем параллельно*. Дело в том, что процессор физически не может обрабатывать параллельно несколько инструкций или процессов. Однако его вычислительной мощи хватает настолько, что он может выполнять все операции по небольшому фрагменту по очереди, отводя на каждый такой фрагмент по очень маленькому кусочку времени, настолько, что кажется, будто все процессы в компьютере выполняются параллельно.

Точно такая же ситуация происходит и с потоками. Если в программе имеется 3 потока, то сначала выполняется кусочек кода из одного потока, потом кусочек кода из другого, затем – из третьего, после чего процессор снова переходит к какому-либо из двух других потоков. Выбор, какой поток необходимо назначить для выполнения в данный момент остаётся за процессором. Происходит это в доли миллисекунд, поэтому происходит ощущение параллельной работы потоков.

Стандартно в проектах Visual Studio существует только один **основной поток – в методе Main**. Всё, что в нём выполняется – выполняется последовательно строка за строкой. Но при необходимости можно “распараллелить” выполняемые процессы при помощи потоков.

# Реализация потоков в C#

### Как создавать потоки в C#

Перво-наперво для работы с потоками в C# необходимо подключить специальную директиву:

using System.Threading;

Именно она позволяет нам реализовывать необходимые потоки и их настройку.

Дальше стоит понять, что **любой поток в C# должен обязательно происходить в каком-либо методе или функции**, поэтому для работы с новым потоком необходимо сначала создать для него, например, метод, который и будет точкой входа для этого потока.

Для примера создадим пустой метод под именем **fornewthread**, который ничего не возвращает:

static void fornewthread()

{

//в действительности здесь должны находиться инструкции, которые будут выполняться в нашем новом потоке

}

Теперь мы можем создать сам поток (например, в главном методе **Main**). Назовём его **mythread**:

Thread mythread = new Thread(fornewthread);

Как вы видите, для создания потока нужно вызвать делегат Thread, а также передать конструктору адрес метода (в скобках).

### Как запускать потоки в C#

Потоки в C# начинают выполняться не сразу после их инициализации. Каждый из созданных потоков необходимо сначала запустить. Делается это следующим образом: ***имя\_потока.Start();***. В случае из нашего примера строка запуска будет такой:

mythread.Start();

Мы записали данный код, также как и предыдущий, в методе **Main** сразу после инициализации самого потока. Теперь, при запуске программы, как только в **Main** выполнится метод **Start()**, начнёт работать вновь созданный поток. После того, как выполнился **Start()**, работа программы “распараллеливается”: один поток начинает выполнять код из метода **fornewthread**, а второй поток продолжает выполнять операции, которые остались ещё не выполненными в методе **Main** (после **Start()**). Естественно, это всё произойдёт, если в **fornewthread** или в **Main** имеется дополнительный код. Если же один из потоков выполнит работу раньше другого, то первый будет ожидать окончания выполнения работы последнего. А делать это он будет по причине, описанной выше: так как оба потока являются основными, то завершение какого-либо одного потока не влияет на завершение другого. Если бы один из наших потоков был фоновым и “задержался” бы с работой, то при окончании работы основного потока, принудительно завершился бы и он.

### Как приостановить потоки в C#

Иногда бывают такие сценарии, когда необходимо бывает приостановить поток, например для того, чтобы пропустить другие потоки и не мешать им выполнять свою работу, либо для снижения потребления процессорного времени.

Есть несколько способов остановки потоков, но в данной статье мы рассмотрим наиболее употребляемый:

Thread.Sleep(100);

В этой строке мы указали, что поток, к которому будет относиться данная инструкция, будет приостановлен на 100 миллисекунд. Как нетрудно догадаться после такого объяснения, значение в скобках указывает, на какой период времени в миллисекундах будет приостановлен поток. Как только заданное время пройдёт, поток снова возобновит свою работу и продолжит с места, на котором остановился. Кстати говоря, число, заданное в скобках – это количество миллисекунд для задержки в идеале, на практике поток может возобновиться на несколько миллисекунд позже или раньше, это зависит не только от программы, но и от характеристик операционной системы.

Поток приостановится на положенный ему интервал времени (мы же помним, что все потоки выполняются по кусочку и по очереди, и при этом на каждый такой кусочек выделяется очень небольшое количество времени, но достаточное для того, чтобы выполнить некоторое количество действий), который он “отдаёт” любому другому готовому к выполнению потоку с таким же приоритетом, как и у него (про приоритеты поговорим ниже). Если же не удастся найти ни один такой “ожидающий” поток, то выполнение текущего потока не будет приостановлено, и он продолжит свою работу.

**Остановку нужно прописывать в методе, который будет выполнять заданный поток.**

### Приоритеты потоков в C#

Когда в программе фигурирует несколько потоков, выбор процессором следующего потока для выполнения не является рандомным. Дело в том, что у каждого потока имеется значение приоритета, чем выше приоритет потока, тем важнее для процессора предоставить время и ресурсы для выполнения именно ему. Если же приоритет потока не слишком высокий, значит он может и подождать в очереди, пока выполняются более приоритетные потоки.

CLR считывает и анализирует значение приоритета и на его основании выделяет данному потоку то или иное количество времени.

Всего существует пять вариантов приоритетов потоков в C#:

* **Highest**- самый высокий
* **AboveNormal**- выше среднего
* **Normal**- стандартный
* **BelowNormal**- ниже среднего
* **Lowest**- самый низкий

Как уже было сказано, чем выше приоритет, тем быстрее он выполнится, опережая потоки с меньшим значением. Если у потоков одинаковые приоритеты, они выполняются по очереди.

По умолчанию все создаваемые потоки в C# имеют стандартный приоритет **Normal**. Однако приоритеты потоков можно и менять, делается это так:

**имяпотока.Priority = ThreadPriority.вариантприоритета;**

Допустим мы решили присвоить описанному выше потоку **mythread** самый высокий приоритет. В таком случае нам надо записать (где-нибудь перед методом **Start**):

mythread.Priority = ThreadPriority.Highest;

***Примечание***: приоритет метода Main изменить нельзя. Он всегда будет в позиции Normal.

### Изменение типов потоков в C#

Как мы рассказывали выше, потоки в C# бывают основными и фоновыми. Также мы говорили о том, что при создании потока он по умолчанию становится основным. Однако его тип можно, также как и приоритет, поменять. Для того, чтобы сделать основной поток фоновым, нужно изменить его свойство IsBackground: ***имя\_потока.IsBackground = true;***

mythread.IsBackground = true;

Если установить данное свойство в значение ***true***, то поток будет работать как фоновый, если в значение ***false***- как основной.

Однако, если в программе используется несколько потоков, то не всегда можно понять, какой поток к какому типу относится. Но, к счастью, всегда можно легко это узнать:

bool a = mythread.IsBackground;

Console.WriteLine(a);

Надо всего-навсего присвоить переменной типа **bool** значение данного свойства, и, если необходимо, вывести полученное значение на экран. Суть такая же: если выведется ***true***, значит поток фоновый, иначе – основной.

# Примеры работы потоков в C#

В данном разделе мы на практике увидим, как работают потоки в C# и создадим для наглядного примера пару простеньких программок.

### Программа №1: как работают потоки в C#

В первой программе мы просто реализуем работу нескольких параллельных потоков и особое внимание обратим на выходные данные. Для примера создадим проект C# с тремя потоками, каждый из которых будет выводить числа от 0 до 9.  
Вот его код:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static void mythread1()

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Поток 1 выводит " + i);

}

}

static void mythread2()

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Поток 2 выводит " + i);

}

}

static void Main(string[] args)

{

Thread thread1 = new Thread(mythread1);

Thread thread2 = new Thread(mythread2);

thread1.Start();

thread2.Start();

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Поток 3 выводит " + i);

}

Console.ReadLine();

}

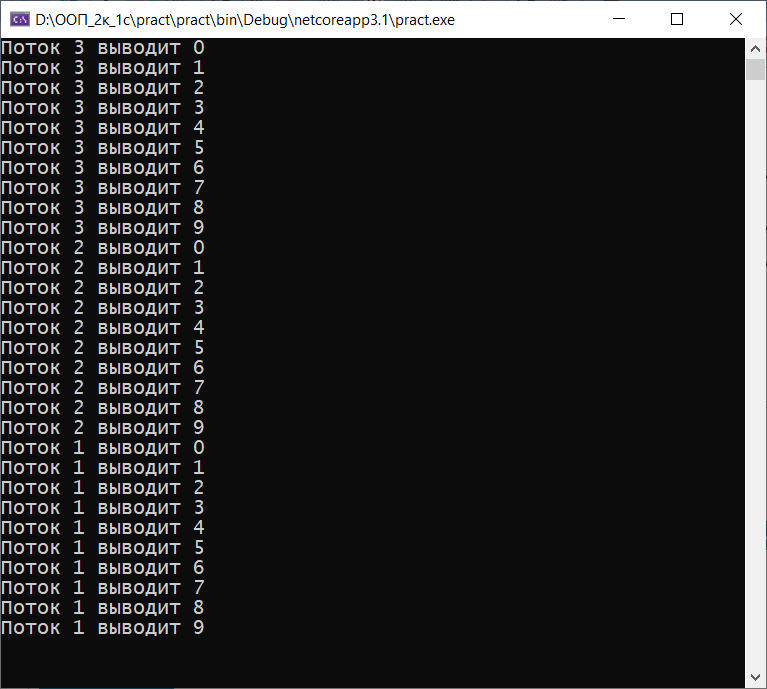
}

}

Итак, как мы помним, для того, чтобы инициализировать потоки в C#, нам необходимы методы, в которых они будут выполняться.

Поэтому мы создали два дополнительных метода **mythread1** и **mythread2**, в которых будут выполняться соответственно потоки **thread1** и **thread2**. Третий поток у нас будет работать непосредственно в методе **Main**.

Мы инициализировали все потоки (третий поток заработал с самого запуска программы, его не надо инициализировать и начинать, естественно), затем запускаем их. В каждой функции происходит одно и то же действие – вывод в консоль цифр от 0 до 9, и при этом каждый поток получил от нас нумерацию от 1 до 3 для удобства восприятия.  
Пора запустить программу и посмотреть, что она нам выведет:



На примере данного скриншота мы можем увидеть “последовательность” многопоточности. Как уже несколько раз было сказано выше, процессор выполняет по кусочку от одного потока, затем кусочек от другого, потом от третьего и так далее.

Здесь мы видим, что процессор сначала предоставил работу третьему потоку (тот вывел первые 7 цифры), затем перешёл к первому потоку (вывел все 10 цифр и прекратил работу), потом выделил ресурсы для второго (успел вывести 9 цифр), затем опять вернул управление третьему потоку (вывел оставшиеся 3 цифры, прекратил работу), а потом опять вернулся ко второму (вывел последнюю цифру). Если мы запустим программу ещё раз, то распределение ресурсов среди потоков может уже измениться и вывод будет происходить как-нибудь иначе, в третий раз – опять по-другому (стоит заметить, что такой “рандом” возможен во многом потому, что все три потока имеют один приоритет – **Normal**, и, кроме того, все они являются основными потоками).

### Программа №3: типы потоков в C#

Как мы рассказывали выше, потоки в C# бывают приоритетными и фоновыми. Разница между ними в том, что если основной поток будет завершен, то и вложенные в него фоновые потоки также будут завершены принудительно. Если же фоновый поток завершится раньше основного, то на основной поток это не повлияет, и он продолжит свою работу.

Мы также описывали выше, как поменять тип потока в программе, как как по умолчанию все потоки создаются приоритетными. Теперь давайте на практике посмотрим, чем отличаются фоновые потоки в C# от основных.

Мы создали программу, которая проливает свет на влияние типов потоков. Код нашего приложения довольно небольшой:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static void mythread1()

{

for (int i = 0; i < 1000000; i++)

{

Console.WriteLine("Фоновый поток выводит " + i);

}

}

static void Main(string[] args)

{

Thread thread1 = new Thread(mythread1);

thread1.IsBackground = true;

thread1.Start();

for (int i = 0; i < 80; i++)

{

Console.WriteLine("Приоритетный поток выводит " + i);

}

}

}

}

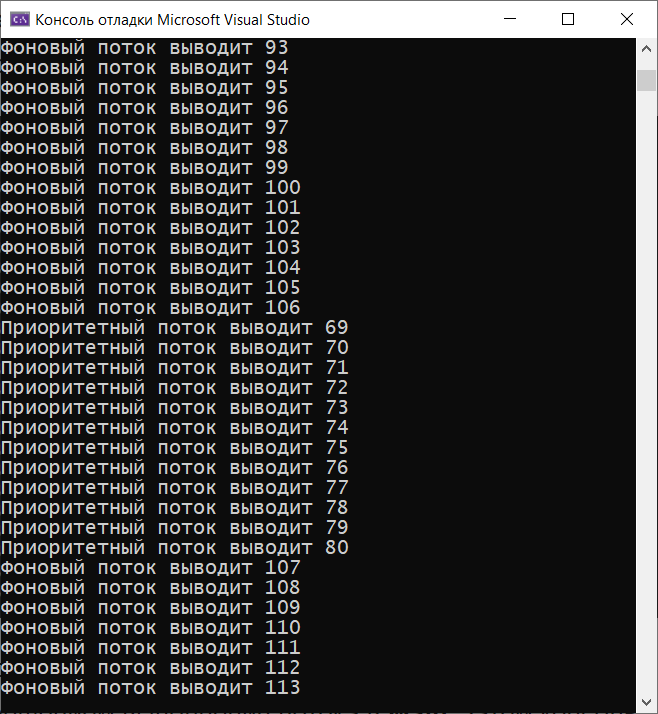
Здесь у нас имеется два потока – **thread1** и поток в из метода **Main**. Изначально они являются приоритетными, следовательно, работают независимо друг от друга, и, пока не закончится выполняться один поток, второй поток нельзя будет закончить принудительно. Однако в нашей программе мы присвоили потоку **thread1** другой тип – фоновый, в строке:

|  |
| --- |
| thread1.IsBackground = true; |

А так как данный поток вызывается в методе **Main**, значит он будет полностью зависеть от потока в этом методе.

Смысл нашей программы в следующем: поток **thread1** вызывается из метода **Main** и начинает работу в методе **mythread1**, который должен выводить на экран числа от 0 до 999999. Однако загвоздка в том, что практически после старта работа метода **Main** прекращается. Поток **thread1** никак не успеет вывести все 999999 чисел за столь короткий промежуток времени, но так как в нашей программе он является фоновым, то принудительно завершится вместе с завершением потока в методе **Main**.

Итак, мы видим, что, как только приоритетный поток закончил вывод, то вскоре после этого фоновый также завершился. Вот последние строки в консоли:



## Класс Thread. Его методы и свойства

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

* Статическое свойство **CurrentContext** позволяет получить контекст, в котором выполняется поток
* Статическое свойство **CurrentThread** возвращает ссылку на выполняемый поток
* Свойство **IsAlive** указывает, работает ли поток в текущий момент
* Свойство **IsBackground** указывает, является ли поток фоновым
* Свойство **Name** содержит имя потока
* Свойство **Priority** хранит приоритет потока - значение перечисления ThreadPriority
* Свойство **ThreadState** возвращает состояние потока - одно из значений перечисления ThreadState

Некоторые методы класса Thread:

* Статический метод **GetDomain** возвращает ссылку на домен приложения
* Статический метод **GetDomainID** возвращает id домена приложения, в котором выполняется текущий поток
* Статический метод **Sleep** останавливает поток на определенное количество миллисекунд
* Метод **Abort** уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
* Метод **Interrupt** прерывает поток, который находится в состоянии WaitSleepJoin
* Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
* Метод **Start** запускает поток

### Получение информации о потоке

Используем вышеописанные свойства и методы для получения информации о потоке:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

// получаем текущий поток

Thread t = Thread.CurrentThread;

//получаем имя потока

Console.WriteLine($"Имя потока: {t.Name}");

t.Name = "Метод Main";

Console.WriteLine($"Имя потока: {t.Name}");

Console.WriteLine($"Запущен ли поток: {t.IsAlive}");

Console.WriteLine($"Приоритет потока: {t.Priority}");

Console.WriteLine($"Статус потока: {t.ThreadState}");

// получаем домен приложения

Console.WriteLine($"Домен приложения: {Thread.GetDomain().FriendlyName}");

Console.ReadLine();

}

}

}

## 

### Статус потока

Статусы потока содержатся в перечислении **ThreadState**:

* **Aborted**: поток остановлен, но пока еще окончательно не завершен
* **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
* **Background**: поток выполняется в фоновом режиме
* **Running**: поток запущен и работает (не приостановлен)
* **Stopped**: поток завершен
* **StopRequested**: поток получил запрос на остановку
* **Suspended**: поток приостановлен
* **SuspendRequested**: поток получил запрос на приостановку
* **Unstarted**: поток еще не был запущен
* **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

### Приоритеты потоков

Приоритеты потоков располагаются в перечислении **ThreadPriority**:

* **Lowest**
* **BelowNormal**
* **Normal**
* **AboveNormal**
* **Highest**

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

## Синхронизация потоков

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static int x = 0;

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

x++;

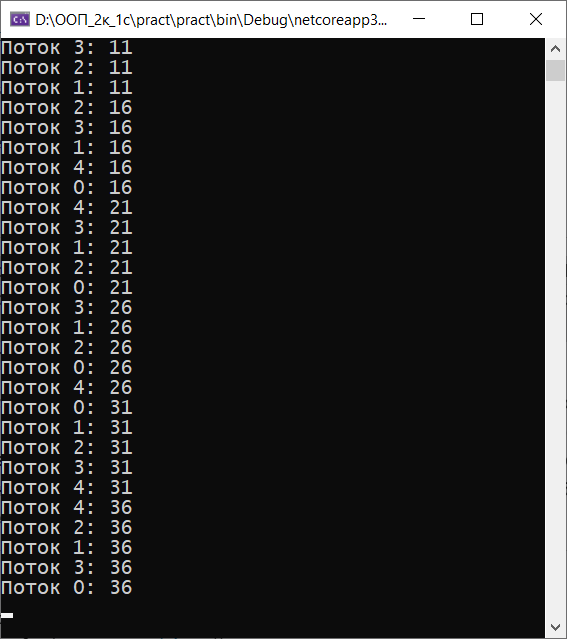
Thread.Sleep(100);

}

}

}

}



Здесь у нас запускаются пять потоков, которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком.

**Синхронизация потоков** – координация действий потоков для получения предсказуемого результата.

**Критическая секция** – участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения.

**Основная причина применения синхронизации** - необходимость разделять среди двух или более потоков общий ресурс (разделяемый ресурс), который может быть одновременно доступен только одному потоку.

**В основу синхронизации положено понятие блокировки**, посредством которой организуется управление доступом к кодовому блоку (критической секции). Когда доступный для каждого из потоков объект (объект синхронизации доступа) заблокирован одним потоком, остальные потоки не могут получить доступ к заблокированному кодовому блоку (критической секции). Когда же блокировка снимается одним потоком, объект (объект синхронизации доступа) становится доступным для использования в другом потоке.

**Объектом синхронизации доступа к разделяемому ресурсу** считается такой объект, который представляет синхронизируемый ресурс. В некоторых случаях им оказывается экземпляр самого ресурса или же произвольный экземпляр класса, используемого для синхронизации.

► **Оператор lock** определяет блок кода, внутри которого весь кодблокируется и становится недоступным для другихпотоков до завершения работы текущего потока

И мы можем переделать предыдущий пример следующим образом:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static int x = 0;

static object locker = new object();

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

lock (locker)

{

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

x++;

Thread.Sleep(100);

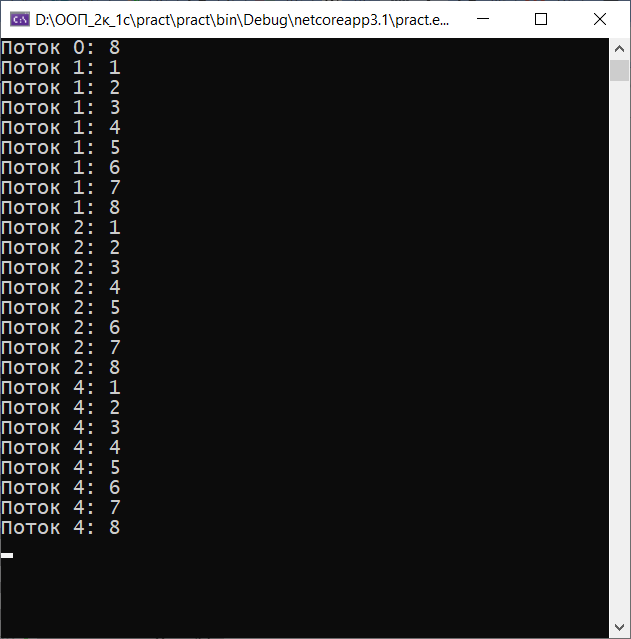
}

}

}

}

}



Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

► **Мониторы**

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом **System.Threading.Monitor**. Оператор C# lock на самом деле является лишь ключевым словом, обозначающим использование типа класса System.Threading.Monitor.

Monitor.Enter() – вход в критическую секцию,

Monitor.Exit() – выход из секции.

Вход и выход должны выполняться в одном и том же потоке.

Рассмотренный в прошлой теме пример будет эквивалентен следующему коду:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static int x = 0;

static object locker = new object();

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = $"Поток {i}";

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

bool acquiredLock = false;

try

{

Monitor.Enter(locker, ref acquiredLock);

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

x++;

Thread.Sleep(100);

}

}

finally

{

if (acquiredLock) Monitor.Exit(locker);

}

}

}

}

Метод Monitor.Enter принимает два параметра - объект блокировки и значение типа bool, которое указывает на результат блокировки (если он равен true, то блокировка успешно выполнена). Фактически этот метод блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, если блокировка осуществлена успешно, он становится доступным для других потоков.



Кроме блокировки и разблокировки объекта класс Monitor имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправил сигнал, то поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

## ► Класс AutoResetEvent

Класс AutoResetEvent также служит целям синхронизации потоков. Этот класс является оберткой над объектом ОС Windows "событие" и позволяет переключить данный объект-событие из сигнального в несигнальное состояние. Так, пример из предыдущей темы мы можем переписать с использованием AutoResetEvent следующим образом:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static AutoResetEvent waitHandler = new AutoResetEvent(true);

static int x = 0;

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = $"Поток {i.ToString()}";

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

waitHandler.WaitOne();

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

x++;

Thread.Sleep(100);

}

waitHandler.Set();

}

}

}

Во-первых, создаем переменную типа AutoResetEvent. Передавая в конструктор значение true, мы тем самым указываем, что создаваемый объект изначально будет в сигнальном состоянии.

Когда начинает работать поток, то первым делом срабатывает определенный в методе Count вызов waitHandler.WaitOne(). Метод WaitOne указывает, что текущий поток переводится в состояние ожидания, пока объект waitHandler не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

После завершения работы вызывается метод waitHandler.Set, который уведомляет все ожидающие потоки, что объект waitHandler снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

Так как в конструкторе AutoResetEvent мы указываем, что объект изначально находится в сигнальном состоянии, то первый из очереди потоков захватывает данный объект и начинает выполнять свой код.

Но если бы мы написали AutoResetEvent waitHandler = new AutoResetEvent(false), тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом waitHandler.WaitOne() до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов методы **WaitAll** и **WaitAny**, которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Так, мы тоже можем использовать WaitAll в вышеприведенном примере. Для этого надо строку waitHandler.WaitOne(); заменить на следующую:

|  |  |
| --- | --- |
|  | AutoResetEvent.WaitAll(new WaitHandle[] {waitHandler}); |

► **Мьютексы**

Еще один инструмент управления синхронизацией потоков представляет класс **Mutex**, также находящийся в пространстве имен System.Threading. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс". Перепишем пример из прошлой темы, используя мьютексы:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static Mutex mutexObj = new Mutex();

static int x = 0;

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = $"Поток {i}";

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

mutexObj.WaitOne();

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

x++;

Thread.Sleep(100);

}

mutexObj.ReleaseMutex();

}

}

}

Сначала создаем объект мьютекса: Mutex mutexObj = new Mutex().

Основную работу по синхронизации выполняют методы **WaitOne()** и **ReleaseMutex()**. Метод mutexObj.WaitOne() приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс mutexObj.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода mutexObj.ReleaseMutex()

Таким образом, когда выполнение дойдет до вызова mutexObj.WaitOne(), поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

► Семафоры

Еще один инструмент, который предлагает нам платформа .NET для управления синхронизацией, представляют **семафоры**. Семафоры позволяют ограничить доступ определенным количеством объектов.

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что единовременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

using System;

using System.Threading;

namespace ConsoleApplication1

{

namespace SemaphoreApp

{

class Program

{

static void Main(string[] args)

{

for (int i = 1; i < 6; i++)

{

Reader reader = new Reader(i);

}

Console.ReadLine();

}

}

class Reader

{

// создаем семафор

static Semaphore sem = new Semaphore(3, 3);

Thread myThread;

int count = 3;// счетчик чтения

public Reader(int i)

{

myThread = new Thread(Read);

myThread.Name = $"Читатель {i}";

myThread.Start();

}

public void Read()

{

while (count > 0)

{

sem.WaitOne();

Console.WriteLine($"{Thread.CurrentThread.Name} входит в библиотеку");

Console.WriteLine($"{Thread.CurrentThread.Name} читает");

Thread.Sleep(1000);

Console.WriteLine($"{Thread.CurrentThread.Name} покидает библиотеку");

sem.Release();

count--;

Thread.Sleep(1000);

}

}

}

}

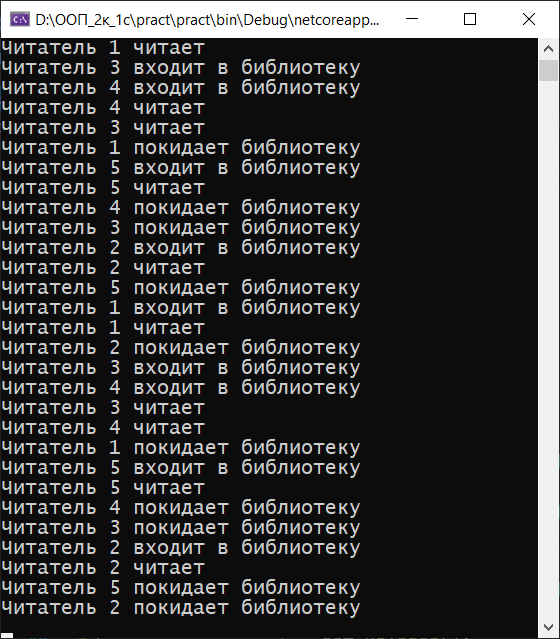
}

В данной программе читатель представлен классом Reader. Он инкапсулирует всю функциональность, связанную с потоками, через переменную Thread myThread.

Для создания семафора используется класс **Semaphore**: static Semaphore sem = new Semaphore(3, 3);. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

Основной функционал сосредоточен в методе Read, который и выполняется в потоке. В начале для ожидания получения семафора используется метод sem.WaitOne(). После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия. После окончания чтения мы высвобождаем семафор с помощью метода sem.Release(). После этого в семафоре освобождается одно место, которое заполняет другой поток.

А в методе Main нам остается только создать читателей, которые запускают соответствующие потоки.



## Таймеры

Одним из важнейших классов, находящихся в пространстве имени System.Threading, является класс **Timer**. Таймер — программный механизм, позволяющий через определенное время выполнять нужный нам участок кода, делая это заданное количество раз. Например, нам нужно запускать резервное копирование папки каждые 10 минут или делать запись в файл журнала каждую секунду.

Например, нам надо запускать какой-нибудь метод через каждые 2000 миллисекунд, то есть раз в две секунды:

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int num = 0;

// устанавливаем метод обратного вызова

TimerCallback tm = new TimerCallback(Count);

// создаем таймер

Timer timer = new Timer(tm, num, 0, 2000);

Console.ReadLine();

}

public static void Count(object obj)

{

int x = (int)obj;

for (int i = 1; i < 9; i++, x++)

{

Console.WriteLine($"{x \* i}");

}

}

}

}

Первым делом создается объект делегата **TimerCallback**, который в качестве параметра принимает метод. Причем данный метод должен в качестве параметра принимать объект типа object.

И затем создается таймер. Данная перегрузка конструктора таймера принимает четыре параметра:

* объект делегата TimerCallback
* объект, передаваемый в качестве параметра в метод Count
* количество миллисекунд, через которое таймер будет запускаться. В данном случае таймер будет запускать немедленно после создания, так как в качестве значения используется 0
* интервал между вызовами метода Count

И, таким образом, после запуска программы каждые две секунды будет срабатывать метод Count.

Если бы нам не надо было бы использовать параметр obj у метода Count, то при создании таймера мы могли бы указывать в качестве соответствующего параметра значение null: Timer timer = new Timer(tm, null, 0, 2000);