Многопоточное программирование.

Многопоточность. Thread. Синхронизация. Пул потоков.

[О причинах многопоточности](#_3znysh7)

[Как это работает на платформе .Net](#_qrcviznymi1v)

[Класс Thread](#_tyjcwt)

[Синхронизация потоков](#_1t3h5sf)

[Пул потоков. ThreadPool](#_4d34og8)

[Применение потоков в приложении "Рассыльщик"](#_2s8eyo1)

[Домашнее задание](#_3rdcrjn)

[Дополнительные материалы](#_26in1rg)

[Используемая литература](#_lnxbz9)

# 

# 

# О причинах многопоточности

Говоря о многопоточности, мы подразумеваем возможность приложения выполнять несколько задач одновременно. Для чего это необходимо? Во-первых, задачи, требующие потенциально большого времени на выполнение, будет исполняться в отдельном потоке, не будут задерживать исполнение иных задач, что необходимо для пользовательского интерфейса, который остаётся активным. Повышается масштабируемость. В качестве задач, требующих применения многопоточности, например, можно привести:

* Обращение к серверу, когда ответ от него может занять определенное время.
* “Тяжелый” запрос к базе данных либо файлу на диске.
* Операции, относящиеся, к парсингу большого объёма данных.
* Преобразование данных, например, пересчёт размеров HD изображения и т.д.

Использование многопоточности как даёт неоспоримые преимущества, так и создает определённые проблемы. Это связано с некоторыми ограничениями на использованием потоков со стороны ОС, которое не гарантирует запуск потока немедленно, так и проблемой разделяемых данных, связанную с доступом к одним и тем же данным из разных потоков. Последняя стоит более остро и будет разбираться ниже.

Отдельно хочется подчеркнуть различие потока и процесса. Если процесс - это экземпляр исполняющегося приложения, имеющего выделенные системные ресурсы, как память, время ЦП, то поток является определённым путём исполнения, внутри процесса.

## Как это работает на платформе .Net

При запуске какого-либо приложения - системой создаётся для него процесс. Он, как минимум, содержит один поток и этот поток называется главным. Точкой входа в приложение является метод Main (не важно WPF, WinForms или консольное приложение), его вызов ведёт к созданию основного потока, который, в свою очередь, может порождать вторичные потоки.

Процесс, содержащий единственный поток, является потокобезопастным, но в тоже время операции, выполняющиеся в этом потоке, вынуждены ожидать выполнения друг друга, что не лучшим образом сказывается на производительности. Это связано и с тем, что поток выполнения UI запускается в главном потоке, что делает интерфейс неотзывчивым при длительных операциях, и у пользователя возникает ощущение, что программа “зависла”.

С другой стороны, бездумное увеличение количества потоков выполнения, тоже может сказаться негативно на производительности.



# Класс Thread

В .Net есть несколько вариантов, предоставляющих возможность работы с потоками. Самым простым из них является класс Thread, расположенный в пространстве имён System.Threading.

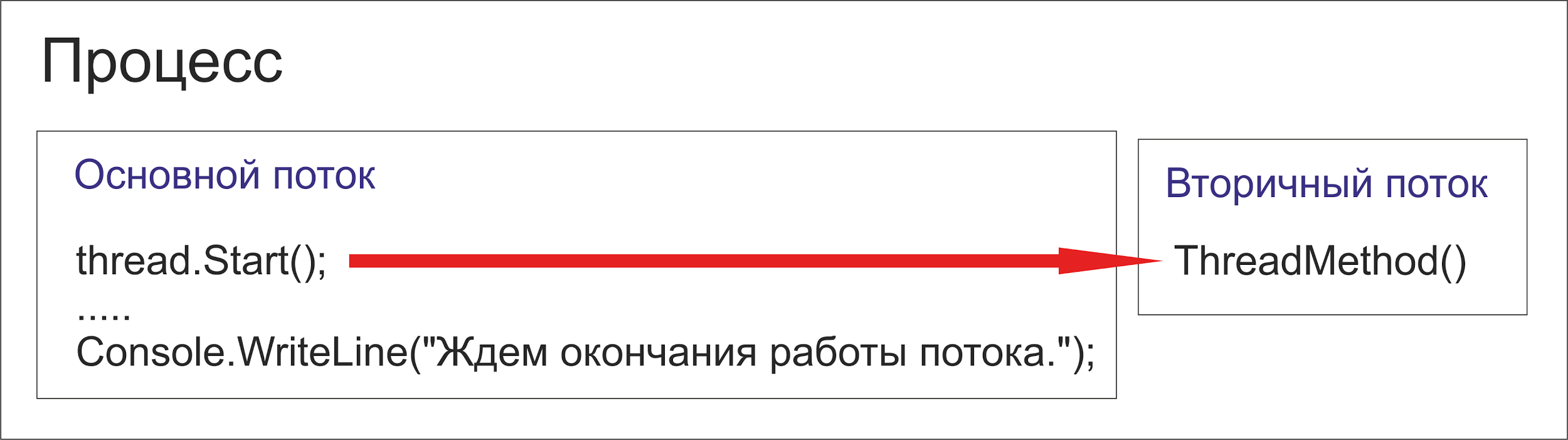
В общем, создание потоков с помощью Thread следует алгоритму:

1. Создать метод, являющийся точкой входа для нового потока.
2. Создать делегат ThreadStart или ParametrizedThreadStart, передав в конструктор в качестве параметра метод, определённый на этапе 1.
3. Создать объект Thread, передав делегат в качестве параметра в конструктор.
4. Вызвать метод Thread.Start(), что запустит поток.

На примере:

|  |
| --- |
| static void Main()  {  Thread thread = new Thread(new ThreadStart(ThreadMethod));  thread.Name = "Вторичный поток";  thread.Start();  Console.WriteLine("Ждём окончания работы потока.");  Console.ReadKey();  }  static void ThreadMethod()  {  Thread.Sleep(2000);  Console.WriteLine($"{Thread.CurrentThread.Name} завершён.");  } |





В данном консольном приложении метод ThreadMethod() будет выполнятся во вторичном потоке. Метод простой, не принимает никаких параметров, а в качестве действия просто происходит приостановка выполнения потока на 2 секунды с помощью метода Thread.Sleep(). Далее метод выведет в консоль сообщение о том, что поток завершён.

В методе Main(), как и говорилось ранее, в начале объявляем переменную thread типа Thread и ей в конструкторе передаём делегат типа ThreadStart, указывающий на метод ThreadMethod.

|  |
| --- |
| Thread thread = new Thread(new ThreadStart(ThreadMethod)); |

Далее свойству Name потока присваиваем имя - “Вторичный поток” (по умолчанию имя инициализируется пустой строкой) и запускаем поток на выполнение.

|  |
| --- |
| thread.Name = "Вторичный поток";  thread.Start(); |

Как результат работы приложения:

* Вывод сообщения “Ожидание окончания работы потока.”
* Ожидание завершения вторичного потока.
* Вывод сообщения “Вторичный поток завершен.”

Помимо свойства Name, у потоков есть ещё несколько интересных свойств, например, Priority, отвечающее за приоритет выполнения потока и являющийся перечислением типа ThreadPriority:

|  |
| --- |
| public enum ThreadPriority  {  Lowest,  BelowNormal,  Normal,  AboveNormal,  Highest  } |

Например, задать потоку высший приоритет выполнения в нашем приложении возможно таким образом:

|  |
| --- |
| Thread thread = new Thread(new ThreadStart(ThreadMethod))  {  Priority = ThreadPriority.Highest, // Задаём высший приоритет  Name = "Вторичный поток"  };  thread.Start(); |

По умолчанию все создаваемые потоки, имеют приоритет ThreadPriority.Normal.

Хорошо, как создавать потоки ясно, но как передавать в поток параметры при необходимости. Для этого служит делегат ParametrizedThreadStart.

Изменим пример выше следующим образом:

|  |
| --- |
| static void Main()  {  int sleepTime = 2000; // величина приостановки потока  Thread thread = new Thread(new ParameterizedThreadStart(ThreadMethod));  thread.Name = "Вторичный поток";  thread.Start(sleepTime);  Console.WriteLine("Ждем окончания работы потока.");  }  static void ThreadMethod(object sleepTime)  {  Thread.Sleep((int)sleepTime);  Console.WriteLine($"{Thread.CurrentThread.Name} завершен.");  } |



Вывод аналогичен предыдущему примеру.

В данном случае объявляем переменную sleepTime типа int, содержащую величину приостановки, и которую мы будем передавать в поток. Теперь, при создании потока, в конструктор передаётся делегат типа ParametrizedThreadStart:

|  |
| --- |
| Thread thread = new Thread(new ParameterizedThreadStart(ThreadMethod)); |

Само значение передаётся в поток при его запуске:

|  |
| --- |
| thread.Start(sleepTime); |

Здесь необходимо сделать важное замечание - делегат ParametrizedThreadStart может указывать на метод, принимающий в качестве параметра один объект типа object. Далее, в теле метода, он приводится к int, для дальнейшего использования.

Резонный вопрос, что делать, когда необходимо передать в поток несколько разнотипных параметров?

Допустим, нам необходимо передать величину приостановки и сообщение, выводимое при завершении выполнения потока. Используем подход, основанный на использовании класса, содержащим необходимые нам поля:

|  |
| --- |
| public class ThreadClass  {  public int SleepTime { get; set; }  public string Message { get; set; }  } |

Внесём изменения в код:

|  |
| --- |
| ThreadClass threadClass = new ThreadClass  {  SleepTime = 2000,  Message = "Поток завершен."  };  Thread thread = new Thread(new ParameterizedThreadStart(ThreadMethod))  {  Priority = ThreadPriority.Highest,  Name = "Вторичный поток"  };  thread.Start(threadClass);  Console.WriteLine("Ждем окончания работы потока.");  static void ThreadMethod(object obj)  {  ThreadClass threadClass = (ThreadClass)obj;  Thread.Sleep(threadClass.SleepTime);  Console.WriteLine(threadClass.Message);  } |

Определяем и инициализирует начальными значениями переменную threadClass типа ThreadClass

Передаем её поток, но тут также есть ограничение - метод Thread.Start() небезопасен к типам, и мы вынуждены приводить полученный объект к необходимому типу:

|  |
| --- |
| ThreadClass threadClass = (ThreadClass)obj; |

Хорошим решением описываемой проблемы будет объявление всех методов и переменных в одном классе, а сам поток запускать через делегат **ThreadStart**:

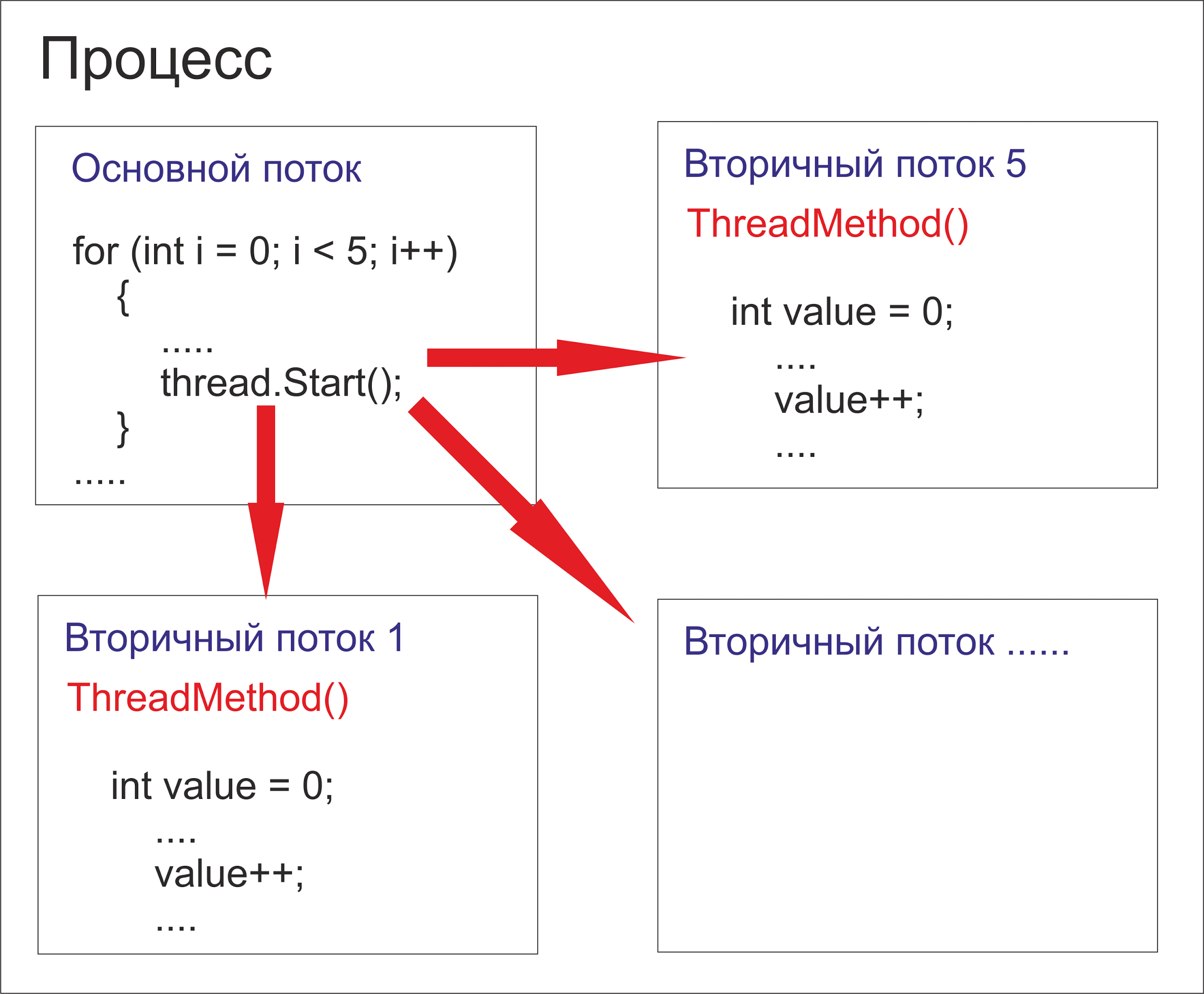
|  |
| --- |
| ThreadClass threadClass = new ThreadClass(2000, "Поток завершен.");  Thread thread = new Thread(new ThreadStart(threadClass.ThreadClassMethod))  {  Priority = ThreadPriority.Highest,  Name = "Вторичный поток"  };  thread.Start();  Console.WriteLine("Ждем окончания работы потока.");  public class ThreadClass  {  private int \_sleepTime;  private string \_message;  public ThreadClass(int sleepTime, string message)  {  \_sleepTime = sleepTime;  \_message = message;  }  public void ThreadClassMethod()  {  Thread.Sleep(\_sleepTime);  Console.WriteLine(\_message);  }  } |

# Синхронизация потоков

Зачастую в приложении имеются какие-то разделяемые ресурсы, такие как: переменные, файлы, иные ресурсы. При использовании потоков остро становится проблема синхронизации и ограничения доступа к таким ресурсам конкурентами.

Приведём пример:

|  |
| --- |
| static void Main()  {  for (int i = 0; i < 5; i++)  {  Thread thread = new Thread(ThreadMethod) {Name = i.ToString()};  thread.Start();  }  Console.ReadLine();  }  static void ThreadMethod()  {  int value = 0;  for (int i = 0; i < 5; i++)  {  Console.WriteLine("Поток {0}: {1}", Thread.CurrentThread.Name, value);  value++;  Thread.Sleep(200);  }  } |



Здесь создаются и запускаются на выполнение последовательно 5 потоков, каждый вызывает метод

ThreadMethod(). В данном методе происходит вывод в консоль значение переменной value и имени потока, вызвавшего метод, затем инкриминация значения. Основная проблема состоит в том, что переменная value, являясь разделяемым ресурсом, будет выводиться некорректно, т.к. потоки, вызывая метод ThreadMethod() и производя инкремент value, будут менять её значение.

Чтобы избежать подобного поведения, необходима синхронизация потоков. Рассмотрим блокировку доступа с помощью оператора lock. В общем случае данный тип блокировки определяется так:

|  |
| --- |
| lock (lockObject)  {  // синхронизируемый код  } |

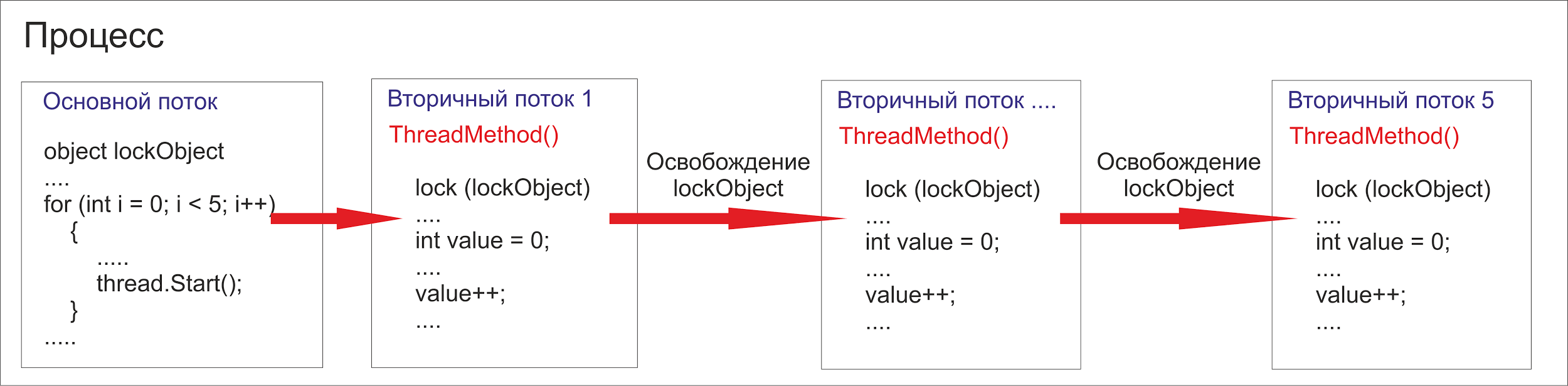
Где в данном случае lockObject является объектом синхронизации, и он гарантирует, что доступ к ресурсу будет иметь поток, который получил эту блокировку, вплоть до её освобождения. Данный объект должен быть типа object.

В качестве наглядной иллюстрации работу lock можно рассматривать как очередь в магазине с одной кассой - все покупатели обслуживаются строго друг за другом, обслуживание нескольких покупателей одновременно невозможно.

Изменим предыдущий пример:

|  |
| --- |
| private static object lockObject = new object();  static void Main()  {  for (int i = 0; i < 5; i++)  {  Thread thread = new Thread(ThreadMethod) {Name = i.ToString()};  thread.Start();  }  Console.ReadLine();  }  static void ThreadMethod()  {  lock (lockObject)  {  int value = 0;  for (int i = 0; i < 5; i++)  {  Console.WriteLine("Поток {0}: {1}", Thread.CurrentThread.Name, value);  value++;  Thread.Sleep(200);  }  }  } |

Таким образом, добавив объект блокировки и использовав оператор lock, потоки будут иметь последовательный доступ с переменной и работа приложения будет корректной.

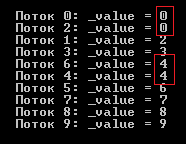


Ещё одним из вариантов синхронизации рассмотрим атрибут [Synchronization]. Он относится к атрибутам уровня класса и является эффективным средством. Объекты, имеющие данный атрибут, размещаются, исполняющей средой, в контексте синхронизации.

Рассмотрим пример:

|  |
| --- |
| static void Main()  {  ClassWithSynchronizationAttribute obj = new ClassWithSynchronizationAttribute();  for (int i = 0; i < 10; i++)  {  Thread thread = new Thread(obj.ThreadMethod) {Name = i.ToString()};  thread.Start();  }  Console.ReadLine();  }  public class ClassWithSynchronizationAttribute  {  private int \_value;  public void ThreadMethod()  {  Console.WriteLine(" Поток {0}: \_value = {1}", Thread.CurrentThread.Name, \_value);  \_value++;  Thread.Sleep(2000);  }  } |

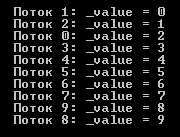
Запустив его на выполнение, мы видим, что переменная \_value инкриминируется неверно из-за конкурентного доступа к ней потоков:



Изменим пример таким образом, чтобы было возможность синхронизации - добавим атрибут [Synchronization] и наследуем класс от ContextBoundObject:

|  |
| --- |
| [Synchronization]  public class ClassWithSynchronizationAttribute : ContextBoundObject  {  // код класса  } |

После таких нехитрых манипуляций код станет полностью безопасным к потокам.



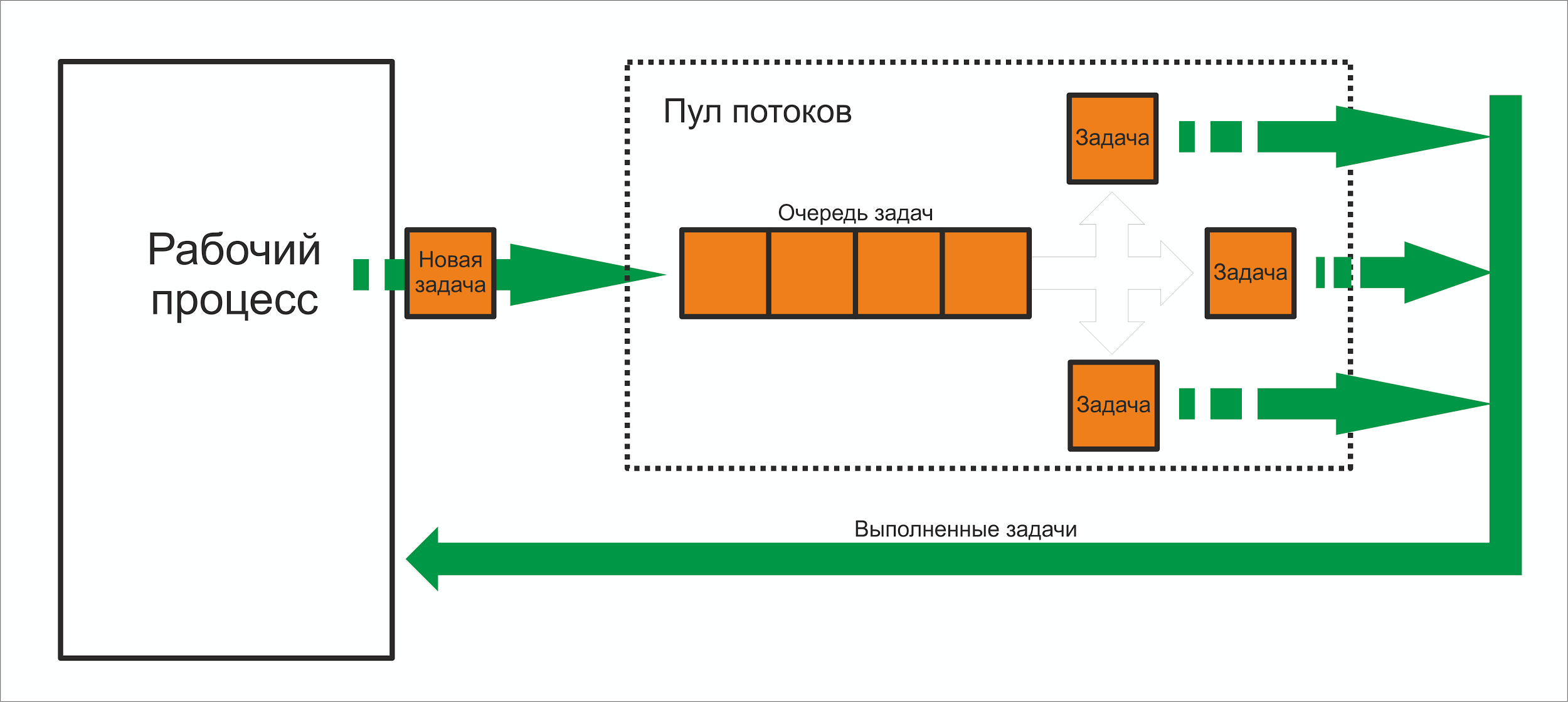
Использовать этот подход, несмотря на легкость его применения, необходимо с осторожностью, так как члены класса будут блокироваться в любом случае, необходимо это или нет, что может негативно отразится на производительности.

# Пул потоков. ThreadPool

На создание, удаление и переключение между потоками необходимы затраты времени. Чтобы минимизировать расходы, связанные с такими операциями, используются пулы потоков. Основной их смысл - меньше число используемых потоков и больше объём выполненной работы на каждый, также пул берёт на себя автоматическое регулирование количества потоков.

Класс ThreadPool как раз и используется в C# для подобных действий. Он позволяет указывать минимальное и максимальное количество потоков, содержащихся в пуле:

|  |
| --- |
| // установить минимальное количество потоков в пуле - 2  ThreadPool.SetMinThreads(2, 2);  // установить максимальное количество потоков в пуле - 5 ThreadPool.SetMaxThreads(5, 5); |



Пример:

|  |
| --- |
| private static object lockObject = new object();  static void Main()  {  ThreadPool.SetMinThreads(2, 2);  ThreadPool.SetMaxThreads(5, 5);  for (int i = 0; i < 5; i++)  {  ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadMethod));  }  Console.ReadLine();  }  static void ThreadMethod(object state)  {  lock (lockObject)  {  int value = 0;  for (int i = 0; i < 5; i++)  {  Console.WriteLine("Поток {0}: {1}", Thread.CurrentThread.Name, value);  value++;  Thread.Sleep(200);  }  }  } |

Здесь пример, показанный с потоками, слегка модифицирован. Теперь в цикле вместо запуска конкретного потока вызов ThreadMethod() назначается потоку из пула потоков, вызовом метода ThreadPool.QueueUserWorkltem() и передачи делегата типа WaitCallback.

# Применение потоков в приложении "Рассыльщик"

Каким образом возможно применение потоков в ранее написанном приложении “Рассыльщик”?

Первое, что приходит на ум - отправка писем.

Изменим класс EmailSendServiceClass, для многопоточной отправки писем:

|  |
| --- |
| class EmailSendServiceClass  {  private string strLogin; // email, c которого будет рассылаться почта  private string strPassword; // пароль к email, с которого будет рассылаться почта  private string strSmtp = "smtp.yandex.ru"; // smtp - server  private int iSmtpPort = 25; // порт для smtp-server  private string strBody; // текст письма для отправки  private string strSubject; // тема письма для отправки  public EmailSendServiceClass(string sLogin, string sPassword)  {  strLogin = sLogin;  strPassword = sPassword;  }  private void SendMail(object objEmails2)  {  Emails2 obj = (Emails2)objEmails2;  string mail = objEmails2.Email;  string name = objEmails2.Name;  using (MailMessage mm = new MailMessage(strLogin, mail))  {  mm.Subject = strSubject;  mm.Body = "Hello world!";  mm.IsBodyHtml = false;  SmtpClient sc = new SmtpClient(strSmtp, iSmtpPort);  sc.EnableSsl = true;  sc.DeliveryMethod = SmtpDeliveryMethod.Network;  sc.UseDefaultCredentials = false;  sc.Credentials = new NetworkCredential(strLogin, strPassword);  try  {  sc.Send(mm);  }  catch (Exception ex)  {  MessageBox.Show("Невозможно отправить письмо " + ex);  }  }  }  public void SendMails(IQueryable<Emails2> emails)  {  foreach (Emails2 email in emails)  {  Thread thread = new Thread(new ParameterizedThreadStart(SendMail));  thread.Start(email);  }  }  } |

В данном случае используется параметризированный делегат для отправки писем. И внеся минимальные изменения в код - получаем многопоточную отправку писем.

# 

# Домашнее задание

1. Написать приложение, считающее в раздельных потоках:
   1. факториал числа N, которое вводится с клавиатуру;
   2. сумму целых чисел до N, которое также вводится с клавиатуры.
2. \*Написать приложение, выполняющее парсинг CSV-файла, произвольной структуры и сохраняющего его в обычный TXT-файл. Все операции проходят в потоках. CSV-файл заведомо имеет большой объём.

# Дополнительные материалы

1. https://msdn.microsoft.com/ru-ru/default.aspx
2. https://metanit.com

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Рихтер Дж. “CLR via C#”
2. Эндрю Троелсен - “ЯЗЫК ПРОГРАММИРОВАНИЯ C#5.0 И ПЛАТФОРМА .NET 4.5”