

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Саратовский государственный технический университет имени Гагарина Ю.А.»

Филиал федерального государственного бюджетного образовательного
учреждения высшего образования
«Саратовский государственный технический университет имени Гагарина Ю.А.»
в г. Петровске

УТВЕРЖДАЮ
Директор филиала СГТУ
имени Гагарина Ю.А. в г. Петровске
 Е.А. Бессилашникова
«06» 16.06.2024 г.



МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

по междисциплинарному курсу
МДК.01.04 «Системное программирование»

специальности
09.02.07 «Информационные системы и программирование»

Методические указания рассмотрены
на заседании предметной (цикловой) комиссии
общепрофессиональных дисциплин,
профессиональных модулей специальностей
технического профиля
«14» июня 2024 года, протокол №12

Председатель ПЦК  /Ю.А. Табарова /

Пояснительная записка

Методические указания по выполнению практических работ подготовлены на основе рабочей программы ПМ.01 «Разработка модулей программного обеспечения для компьютерных систем» междисциплинарного курса МДК.01.04 «Системное программирование», разработанной на основе ФГОС СПО по специальности 09.02.07 «Информационные системы и программирование» и соответствующих общих (ОК) и профессиональных (ПК) компетенций:

ОК 01. Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам.

ОК 02. Использовать современные средства поиска, анализа и интерпретации информации и информационные технологии для выполнения задач профессиональной деятельности

ОК 03. Планировать и реализовывать собственное профессиональное и личностное развитие, предпринимательскую деятельность в профессиональной сфере, использовать знания по финансовой грамотности в различных жизненных ситуациях.

ОК 04. Эффективно взаимодействовать и работать в коллективе и команде.

ОК 05. Осуществлять устную и письменную коммуникацию на государственном языке Российской Федерации с учетом особенностей социального и культурного контекста

ОК 06. Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных общечеловеческих ценностей, в том числе с учетом гармонизации межнациональных и межрелигиозных отношений, применять стандарты антикоррупционного поведения

ОК 07. Содействовать сохранению окружающей среды, ресурсосбережению, применять знания об изменении климата, принципы бережливого производства, эффективно действовать в чрезвычайных ситуациях

ОК 08. Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности

ОК 09. Пользоваться профессиональной документацией на государственном и иностранном языках.

ОК 10. Использовать знания по финансовой грамотности, планировать предпринимательскую деятельность в профессиональной сфере

ПК 1.2 Разрабатывать программные модули в соответствии с техническим заданием.

ПК 1.3 Выполнять отладку программных модулей с использованием специализированных программных средств.

Целью освоения междисциплинарного курса МДК.01.04 «Системное программирование» является:

При выполнении практических работ студент должен **знать:**

- основные этапы разработки программного обеспечения;
- основные принципы технологии структурного и объектно-ориентированного программирования;
- способы оптимизации и приемы рефакторинга;
- основные принципы отладки и тестирования программных продуктов.

При выполнении практических работ студент должен **уметь:**

- осуществлять разработку кода программного модуля на языках низкого и высокого уровней;
 - создавать программу по разработанному алгоритму как отдельный модуль;
 - выполнять отладку и тестирование программы на уровне модуля;
- осуществлять разработку кода программного модуля на современных языках программирования;
- уметь выполнять оптимизацию и рефакторинг программного кода;
 - оформлять документацию на программные средства.

Содержание практических занятий определено рабочей программой и тематическим планированием, соответствует теоретическому материалу изучаемых разделов учебной дисциплины.

Объём практических занятий по дисциплине определяется учебным планом по данной специальности.

Продолжительность практического занятия - 2 академических часа. Перед проведением практического занятия преподавателем организуется инструктаж, а по ее окончании – обсуждение итогов.

Комплект методических указаний по выполнению практических работ междисциплинарного курса МДК.01.04 «Системное программирование» содержит 30 практических занятий.

**Перечень практических работ
междисциплинарному курсу
МДК.01.04 «Системное программирование»**

ПРАКТИЧЕСКАЯ РАБОТА №1-8

Тема: Использование потоков

ПРАКТИЧЕСКАЯ РАБОТА №9-16

Тема: Обмен данными

ПРАКТИЧЕСКАЯ РАБОТА №17-24

Тема: Сетевое программирование сокетов.

ПРАКТИЧЕСКАЯ РАБОТА №25-30

Тема: Работы с буфером экрана.

ИНСТРУКЦИИ ДЛЯ ОБУЧАЮЩИХСЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Прежде чем приступить к выполнению заданий, внимательно прочитайте данные рекомендации. Практические работы включают в себя задания следующих видов:

НАПРИМЕР:

1. Ответ на поставленные вопросы (с аргументацией)

Прочитайте вопрос и вникните в него.

Для удобства подчеркните ту, фразу, которая, по вашему мнению, является главной. Это поможет вам быстрее сориентироваться при ответе на вопрос.

Если вы считаете, что можете ответить на вопрос без помощи лекции и дополнительной литературы – приступайте. Если же вопрос заставляет вас сомневаться, откройте лекционную тетрадь (учебник или дополнительную литературу), прочитайте необходимый пункт, вникните в содержание и после этого приступайте за работу.

ГЛАВНОЕ! Не переписывайте отрывки лекции в рабочую тетрадь! Четко отвечайте на ПОСТАВЛЕННЫЙ вопрос!

Не забудьте привести аргументацию (обоснование) вашей позиции, если вопрос предполагает личностное отношение к проблеме.

2. Заполнение таблиц и схем

Прочитайте название таблицы или схемы.

Исходя из названия, вы поймете цель предстоящей работы.

Воспользуйтесь материалами лекций или другими источниками, чтобы заполнить таблицу (схему).

Используйте цветные графические материалы для выделения строк, столбцов или элементов схем.

Особое внимание обращайтесь на четкость при отборе материала: делайте записи кратко и четко!

ПРАКТИЧЕСКАЯ РАБОТА №1-8

Тема: Использование потоков

Цель: Изучить использование потоков

Оборудование: В соответствии с рабочей программой ПМ.01 «Разработка модулей программного обеспечения для компьютерных систем»:

- Автоматизированные рабочие места на 12-15 обучающихся (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Автоматизированное рабочее место преподавателя (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Проектор и экран;
- Маркерная доска;
- Лицензионное программное обеспечение общего и профессионального назначения.

Справочный материал

Для применения многопоточности существует несколько причин. Предположим, в приложении предпринимается обращение к какому-то серверу в сети, которое может занять определенное время. Вряд ли захочется, чтобы пользовательский интерфейс из-за этого блокировался, и пользователю пришлось просто дожидаться момента, когда от сервера вернется ответ. Пользователь может выполнять в это время какие-то другие действия или вообще отменить отправленный серверу запрос. В таких ситуациях применение многопоточности приносит пользу.

Для всех видов активности, требующих ожидания, например, из-за получения доступа к файлу, базе данных или сети, может запускаться новый поток, позволяющий выполнять в это же время другие задачи. Многопоточность может помочь, даже если есть одни только насыщенные в плане обработки задачи. Многочисленные потоки одного и того же процесса могут одновременно выполняться разными ЦП или, что чаще встречается в наши дни, разными ядрами одного многоядерного ЦП.

Разумеется, необходимо знать особенности одновременного выполнения множества потоков. Из-за того, что они выполняются в одно и то же время, при получении ими доступа к одним и тем же данным могут возникать проблемы. Чтобы этого не происходило, должны быть реализованы механизмы синхронизации.

Поток (thread) представляет собой независимую последовательность инструкций в программе. Потоки играют важную роль как для клиентских, так и для серверных приложений. К примеру, во время ввода какого-то кода C# в окне редактора Visual Studio проводится анализ на предмет различных синтаксических ошибок. Этот анализ осуществляется отдельным фоновым потоком. То же самое происходит и в средстве проверки орфографии в Microsoft Word. Один поток ожидает ввода данных пользователем, а другой в это время выполняет в фоновом режиме некоторый анализ. Третий поток может сохранять записываемые данные во временный файл, а четвертый – загружать дополнительные данные из Интернета.

В приложении, которое функционирует на сервере, один поток всегда ожидает поступления запроса от клиента и потому называется потоком-слушателем (listener thread). При получении запроса он сразу же пересылает его отдельному рабочему

потоку (worker thread), который дальше сам продолжает взаимодействовать с клиентом. Поток-слушатель после этого незамедлительно возвращается к своим обязанностям по ожиданию поступления следующего запроса от очередного клиента.

Каждый процесс состоит из ресурсов, таких как оконные дескрипторы, файловые дескрипторы и другие объекты ядра, имеет выделенную область в виртуальной памяти и содержит как минимум один поток. Потоки планируются к выполнению операционной системой. У любого потока имеется приоритет, счетчик команд, указывающий на место в программе, где происходит обработка, и стек, в котором сохраняются локальные переменные потока. Стек у каждого потока выглядит по-своему, но память для программного кода и куча разделяются среди всех потоков, которые функционируют внутри одного процесса.

Это позволяет потокам внутри одного процесса быстро взаимодействовать между собой, поскольку все потоки процесса обращаются к одной и той же виртуальной памяти. Однако это также и усложняет дело, поскольку дает возможность множеству потоков изменять одну и ту же область памяти.

Основы многопоточной обработки

Различают две разновидности многозадачности: на основе процессов и на основе потоков. В связи с этим важно понимать отличия между ними. Процесс отвечает за управление ресурсами, к числу которых относится виртуальная память и дескрипторы Windows, и содержит как минимум один поток. Наличие хотя бы одного потока является обязательным для выполнения любой программы. Поэтому многозадачность на основе процессов - это средство, благодаря которому на компьютере могут параллельно выполняться две программы и более.

Так, многозадачность на основе процессов позволяет одновременно выполнять программы текстового редактора, электронных таблиц и просмотра содержимого в Интернете. При организации многозадачности на основе процессов программа является наименьшей единицей кода, выполнение которой может координировать планировщик задач.

Поток представляет собой координируемую единицу исполняемого кода. Своим происхождением этот термин обязан понятию "поток исполнения". При организации многозадачности на основе потоков у каждого процесса должен быть по крайней мере один поток, хотя их может быть и больше. Это означает, что в одной программе одновременно могут решаться две задачи и больше. Например, текст может форматироваться в редакторе текста одновременно с его выводом на печать, при условии, что оба эти действия выполняются в двух отдельных потоках.

Отличия в многозадачности на основе процессов и потоков могут быть сведены к следующему: многозадачность на основе процессов организуется для параллельного выполнения программ, а многозадачность на основе потоков — для параллельного выполнения отдельных частей одной программы.

Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря возможности выгодно использовать время простоя, неизбежно возникающее в ходе выполнения большинства программ. Как известно, большинство устройств ввода-вывода, будь то устройства, подключенные к сетевым портам, накопители на дисках или клавиатура, работают намного медленнее, чем центральный процессор (ЦП). Поэтому большую часть своего времени программе приходится ожидать отправки данных на устройство ввода-вывода или приема информации из него. А благодаря

многопоточной обработке программа может решать какую-нибудь другую задачу во время вынужденного простоя.

Например, в то время как одна часть программы отправляет файл через соединение с Интернетом, другая ее часть может выполнять чтение текстовой информации, вводимой с клавиатуры, а третья — осуществлять буферизацию очередного блока отправляемых данных.

Поток может находиться в одном из нескольких состояний. В целом, поток может быть выполняющимся; готовым к выполнению, как только он получит время и ресурсы ЦП; приостановленным, т.е. временно не выполняющимся; возобновленным в дальнейшем; заблокированным в ожидании ресурсов для своего выполнения; а также завершенным, когда его выполнение окончено и не может быть возобновлено.

В среде .NET Framework определены две разновидности потоков: приоритетный и фоновый. По умолчанию создаваемый поток автоматически становится приоритетным, но его можно сделать фоновым. Единственное отличие приоритетных потоков от фоновых заключается в том, что фоновый поток автоматически завершается, если в его процессе остановлены все приоритетные потоки.

В связи с организацией многозадачности на основе потоков возникает потребность в особом режиме, который называется синхронизацией и позволяет координировать выполнение потоков вполне определенным образом. Для такой синхронизации в С# предусмотрена отдельная подсистема.

Все процессы состоят хотя бы из одного потока, который обычно называют основным, поскольку именно с него начинается выполнение программы. Из основного потока можно создать другие потоки.

В языке С# и среде .NET Framework поддерживаются обе разновидности многозадачности: на основе процессов и на основе потоков. Поэтому средствами С# можно создавать как процессы, так и потоки, а также управлять и теми и другими. Для того чтобы начать новый процесс, от программирующего требуется совсем немного усилий, поскольку каждый предыдущий процесс совершенно обособлен от последующего.

Намного более важной оказывается поддержка в С# многопоточной обработки, благодаря которой упрощается написание высокопроизводительных, многопоточных программ на С# по сравнению с некоторыми другими языками программирования.

Классы, поддерживающие многопоточное программирование, определены в пространстве имен System.Threading. Поэтому любая многопоточная программа на С# включает в себя следующую строку кода:

using System.Threading;

Пространство имен System.Threading содержит различные типы, позволяющие создавать многопоточные приложения. Пожалуй, главным среди них является класс Thread, поскольку он представляет отдельный поток. Чтобы программно получить ссылку на поток, выполняемый конкретным его экземпляром, просто вызовите статическое свойство Thread.CurrentThread:


```

static void ExtractExecutingThread()
{
    // Получить поток, выполняющий данный метод.
    Thread currThread = Thread.CurrentThread;
}

```

На платформе .NET не существует прямого соответствия "один к одному" между доменами приложений (AppDomain) и потоками. Фактически определенный AppDomain может иметь несколько потоков, выполняющихся в каждый конкретный момент времени. Более того, конкретный поток не привязан к одному домену приложений на протяжении своего времени существования. Потоки могут пересекать границы доменов приложений, когда это вздумается планировщику Windows и CLR.

Хотя активные потоки могут пересекать границы AppDomain, каждый поток в каждый конкретный момент времени может выполняться только внутри одного домена приложений (другими словами, невозможно, чтобы один поток работал в более чем одном домене приложений сразу). Чтобы программно получить доступ к AppDomain, в котором работает текущий поток, вызовите статический метод Thread.GetDomain():

```

static void ExtractAppDomainHostingThread()
{
    // Получить AppDomain, в котором работает текущий поток.
    AppDomain ad = Thread.GetDomain();
}

```

Единственный поток также в любой момент может быть перемещен в определенный контекст, и он может перемещаться в пределах нового контекста по прихоти CLR. Для получения текущего контекста, в котором выполняется поток, используйте статическое свойство Thread.CurrentContext (которое возвращает объект System.Runtime.Remoting.Contexts.Context):

```

static void ExtractCurrentThreadContext()
{
    // Получить контекст, в котором работает текущий поток.
    Context ctx = Thread.CurrentContext;
}

```

За перемещение потоков между доменами приложений и контекстами отвечает CLR. Как разработчик .NET, вы всегда остаетесь в счастливом неведении относительно того, когда завершается каждый конкретный поток (или куда именно он будет помещен после перемещения). Тем не менее, полезно знать о различных способах получения лежащих в основе примитивов.

Наиболее простым способом для создания потока является определение делегата и его вызов асинхронным образом. Делегаты могут исполнять роль безопасных для типов ссылок на методы. Помимо этого класс Delegate поддерживает и возможность асинхронного вызова этих методов. Для решения поставленной

задачи он создает "за кулисами" отдельный поток.

На объявление делегата .NET компилятор C# отвечает построением запечатанного класса, который наследуется от System.MulticastDelegate (который, в свою очередь, унаследован от System.Delegate). Эти базовые классы предоставляют каждому делегату возможность поддерживать список адресов методов, которые могут быть вызваны позднее.

```
// Тип делегата C#.  
public delegate int BinaryOp(int x, int y);
```

Исходя из определения, BinaryOp может указывать на любой метод, принимающий два целых числа (по значению) в качестве аргументов и возвращающий целое число. После компиляции сборка с определением делегата будет содержать полноценное определение класса, сгенерированного динамически при построении проекта, на основе объявления делегата. В случае BinaryOp этот класс более или менее похож на следующий (записан в псевдокоде):

```
public sealed class BinaryOp : System.MulticastDelegate  
{  
    public BinaryOp(object target, uint functionAddress);  
    public void Invoke(int x, int y);  
    public IAsyncResult BeginInvoke(int x, int y, AsyncCallback cb, object state);  
    public int EndInvoke(IAsyncResult result);  
}
```

Сгенерированный метод Invoke() используется для вызова метода, поддерживаемого объектом делегата в синхронном режиме. Поэтому вызывающий поток (такой как первичный поток приложения) должен будет ждать, пока не завершится вызов делегата. Также вспомните, что в C# метод Invoke() не нужно вызывать в коде напрямую — он может быть инициирован неявно, "за кулисами", при применении "нормального" синтаксиса вызова метода.

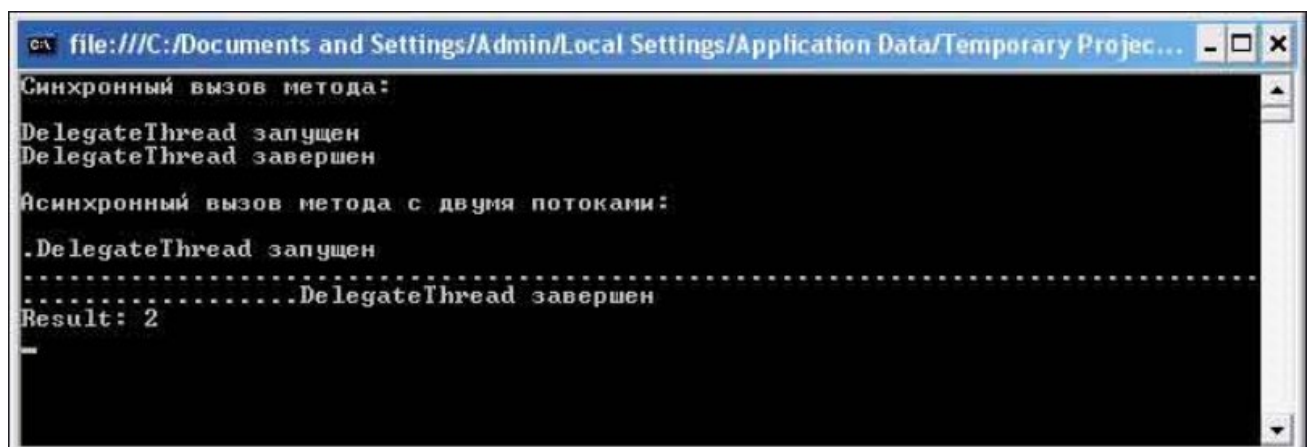
Теперь можно применять различные приемы для асинхронного вызова данного делегата и возврата результатов.

Одним из таких приемов является опрос и проверка, завершил ли делегат свою работу. Созданный класс delegate предоставляет метод BeginInvoke(), в котором могут передаваться входные параметры, определенные вместе с типом делегата. Метод BeginInvoke() всегда имеет два дополнительных параметра типа AsyncCallback и object, которые будут рассматриваться позже. Сейчас главный интерес представляет возвращаемый тип BeginInvoke() — IAsyncResult. С помощью IAsyncResult можно извлекать информацию о делегате и проверять, завершил ли он свою работу, что здесь и делается с применением свойства IsCompleted. Цикл while продолжает выполняться в главном потоке программы до тех пор, пока делегат не завершит работу:

Задание 1

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading;
6
7  namespace ConsoleApplication1
8  {
9      public delegate int BinaryOp(int data, int time);
10
11     class Program
12     {
13         static void Main()
14         {
15             Console.WriteLine("Синхронный вызов метода: \n");
16             // Синхронный вызов метода
17             DelegateThread(1, 5000);
18
19             Console.WriteLine("\nАсинхронный вызов метода с двумя потоками: \n");
20             // Асинхронный вызов метода с применением делегата
21             BinaryOp bo = DelegateThread;
22
23             IAsyncResult ar = bo.BeginInvoke(1, 5000, null, null);
24             while (!ar.IsCompleted)
25             {
26                 // Выполнение операций в главном потоке
27                 Console.Write(".");
28                 Thread.Sleep(50);
29             }
30             int result = bo.EndInvoke(ar);
31             Console.WriteLine("Result: " + result);
32
33             Console.ReadLine();
34         }
35
36         static int DelegateThread(int data, int time)
37         {
38             Console.WriteLine("DelegateThread запущен");
39             // Делаем задержку, для эмуляции длительной операции
40             Thread.Sleep(time);
41             Console.WriteLine("DelegateThread завершен");
42             return ++data;
43         }
44     }
45 }
```

После запуска этого приложения можно увидеть, что главный поток и поток делегата выполняются параллельно, а после завершения работы потока делегата главный поток прекращает проход по циклу:



```
file:///C:/Documents and Settings/Admin/Local Settings/Application Data/Temporary Projec...
Синхронный вызов метода:
DelegateThread запущен
DelegateThread завершен

Асинхронный вызов метода с двумя потоками:
.DelegateThread запущен
.....DelegateThread завершен
Result: 2
```

Справочный материал

Вместо выполнения проверки на предмет того, завершил ли делегат работу, после завершения работы главным потоком можно вызвать метод `EndInvoke()` типа делегата. Метод `EndInvoke()` сам ожидает, когда делегат завершит свою работу. Если главный поток завершает выполнение, не дожидаясь завершения работы делегата, поток делегата останавливается.

Метод `BeginInvoke()` всегда возвращает объект, реализующий интерфейс `AsyncResult`, в то время как `EndInvoke()` ожидает единственный параметр совместимого с `AsyncResult` типа. Совместимый с `AsyncResult` объект, возвращаемый из `BeginInvoke()` — это в основном связывающий механизм, который позволяет вызывающему потоку получить позже результат вызова асинхронного метода через `EndInvoke()`.

Интерфейс `AsyncResult` (находящийся в пространстве имен `System`) определен следующим образом:

```
public interface IAsyncResult  
{  
    object AsyncState { get; }  
    WaitHandle AsyncWaitHandle { get; }  
    bool CompletedSynchronously { get; }  
    bool IsCompleted { get; }  
}
```

Другой способ ожидания результата от асинхронного делегата предусматривает применение дескриптора ожидания (wait handle), который ассоциируется с `AsyncResult`. Получить доступ к этому дескриптору ожидания можно через свойство `AsyncWaitHandle`. Это свойство возвращает объект типа `WaitHandle`, с помощью которого можно организовать ожидание завершения работы потоком делегата.

Метод `WaitOne()` принимает в качестве первого необязательного параметра значение тайм-аута, в котором можно указать максимальный период времени ожидания. В рассматриваемом здесь примере этот период составляет 50 миллисекунд. В случае возникновения тайм-аута метод `WaitOne()` возвращает значение `false`, и проход по циклу `while` продолжается. Если во время ожидания до тайм-аута дело не доходит, осуществляется выход из цикла `while` с помощью `break` и получение результата методом `EndInvoke()`. В пользовательском интерфейсе результат выглядит примерно так же, как в предыдущем примере, отличается лишь способ реализации ожидания.

Еще один способ для ожидания результатов от делегата заключается в применении так называемого асинхронного обратного вызова (`asynchronous callback`). С помощью третьего параметра в методе `BeginInvoke` можно передать метод, удовлетворяющий требованиям делегата `AsyncCallback`. Делегат `AsyncCallback` требует определять параметр `AsyncResult` и возвращаемый тип `void`.

Вместо опроса делегата о том, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задания. Чтобы включить такое поведение, необходимо передать экземпляр делегата `System.AsyncCallback` в качестве параметра

методу BeginInvoke(); до сих пор этот параметр был равен null. Если передается объект AsyncCallback, делегат автоматически вызовет указанный метод по завершении асинхронного вызова.

Метод обратного вызова будет вызван во вторичном потоке, а не в первичном. Это имеет важное последствие для потоков с графическим интерфейсом пользователя (WPF или Windows Forms), поскольку элементы управления привязаны к потоку, который их создал, и могут управляться только им. Далее, при рассмотрении библиотеки TPL, будут показаны некоторые примеры работы потоков из графического интерфейса.

Как и любой делегат, AsyncCallback может вызывать только методы, соответствующие определенному шаблону, который в данном случае требует единственного параметра IAsyncResult и ничего не возвращает:

***// Целевые методы AsyncCallback должны иметь следующую сигнатуру
void MyAsyncCallbackMethod(IAsyncResult res)***

В рассматриваемом примере третьему параметру назначается адрес метода TakesAWhileCompleted, который удовлетворяет требованиям делегата AsyncCallback. В последнем параметре методу BeginInvoke можно передать объект, к которому будет производиться доступ из метода обратного вызова. Здесь удобно передать экземпляр самого делегата, так что метод обратного вызова сможет использовать его для получения результата асинхронного метода.

По завершении работы делегата DelegateThread сразу же вызывается метод TakesAWhileCompleted(). Ожидать результатов внутри главного потока нет никакой необходимости. Завершать главный поток перед завершением работы потоков делегатов может быть необязательно, если только остановка работы потоков делегатов при завершении выполнения главного потока не представляет проблемы:

```
using System.Diagnostics;  
...  
Console.WriteLine("\nАсинхронный вызов метода с двумя потоками: \n");  
// Асинхронный вызов метода с применением делегата  
BinaryOp bo = DelegateThread;  
  
bo.BeginInvoke(1, 5000, TakesAWhileCompleted, bo);  
for (int i = 0; i < 100; i++)  
{  
    // Выполнение операций в главном потоке  
    Console.Write(".");  
    Thread.Sleep(50);  
}
```

Метод TakesAWhileCompleted() определяется с типами параметров и возврата, которые указаны в делегате AsyncCallback. Последний параметр, передаваемый в BeginInvoke(), может быть прочитан с помощью ar.AsyncState, а результат получен вызовом в TakesAWhileDelegate метода EndInvoke:

```

static void TakesAWhileCompleted(IAsyncResult ar)
{
    if (ar == null) throw new ArgumentNullException("ar");

    BinaryOp bo = ar.AsyncState as BinaryOp;
    Trace.Assert(bo != null, "Неверный тип объекта");
    int result = bo.EndInvoke(ar);
    Console.WriteLine("Результат: " + result);
}

```

Вместо того чтобы определять отдельный метод и передавать его методу BeginInvoke(), можно воспользоваться лямбда-выражением. Параметр ar имеет тип IAsyncResult. В такой реализации нет необходимости присваивать значение последнему параметру метода BeginInvoke(), потому что лямбда-выражение может напрямую получать доступ к переменной bo, находящейся за пределами контекста данного метода. Однако блок реализации лямбда-выражения все равно должен вызываться из потока делегата, что может и не быть очевидным при определении метода подобным образом.

Модель программирования и приемы, описанные для асинхронных делегатов — опрос, дескрипторы ожидания и асинхронные вызовы — доступны не только для делегатов. Эта же модель программирования, которая называется шаблоном асинхронного вызова (asynchronous pattern), встречается в разнообразных местах .NET Framework. Например, с помощью метода BeginGetResponse() класса HttpWebRequest можно асинхронно отправлять HTTP-запросы, а с помощью метода BeginExecuteReader() класса SqlCommand — запросы к базе данных. Параметры похожи на те, что можно передавать в методе BeginInvoke() делегата, а механизмы, применяемые для получения результатов, выглядят точно так же.

Класс Thread является самым элементарным из всех типов пространства имен System.Threading. Этот класс представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри определенного AppDomain. Этот тип также определяет набор методов (как статических, так и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего AppDomain, а также приостанавливать, останавливать и уничтожать определенный поток. Список основных статических членов приведен ниже:

CurrentContext

Это свойство только для чтения возвращает контекст, в котором в данный момент выполняется поток

CurrentThread

Это свойство только для чтения возвращает ссылку на текущий выполняемый поток.

GetDomain(), GetDomainID()

Этот метод возвращает ссылку на текущий AppDomain или идентификатор этого домена, в котором выполняется текущий поток

Sleep()

Этот метод приостанавливает текущий поток на заданное время

Класс Thread также поддерживает несколько методов уровня экземпляра, часть из которых описана в таблице ниже. Отмена или приостановка активного потока обычно считается плохой идеей. Когда вы делаете это, есть шанс (хотя и небольшой), что поток может допустить "утечку" своей рабочей нагрузки, когда его беспокоят или прерывают.

Член уровня экземпляра	Назначение
IsAlive	Возвращает булевское значение, указывающее на то, запущен ли поток (и еще не прерван и не отменен)
IsBackground	Получает или устанавливает значение, указывающее, является ли данный поток "фоновым" (подробнее объясняется далее)
Name	Позволяет вам установить дружественное текстовое имя потока
Priority	Получает или устанавливает приоритет потока, который может принимать значение из перечисления ThreadPriority
ThreadState	Получает состояние данного потока, которому может быть присвоено значение из перечисления ThreadState
Abort()	Инструктирует CLR прервать поток, как только это будет возможно
Interrupt()	Прерывает (т.е. приостанавливает) текущий поток на заданный период ожидания
Join()	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, в котором вызван Join()) не завершится
Resume()	Возобновляет ранее приостановленный поток
Start()	Инструктирует CLR запустить поток как можно скорее
Suspend()	Приостанавливает поток. Если поток уже приостановлен, вызов Suspend() не дает эффекта

Задание 2

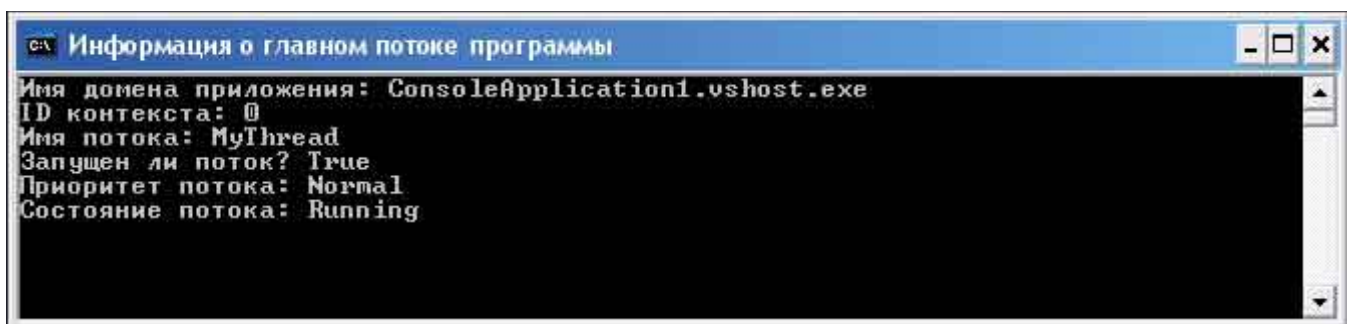
Получение статистики о текущем потоке

Вспомните, что точка входа исполняемой сборки (т.е. метод Main()) запускается в первичном потоке выполнения. Чтобы проиллюстрировать базовое применение типа Thread, предположим, что имеется новое консольное приложение. Как известно, статическое свойство Thread.CurrentThread извлекает объект Thread, представляющий текущий выполняющийся поток. После получения текущего потока можно вывести разнообразную статистику о нем:

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading;
6
7 namespace ConsoleApplication1
8 {
9     class Program
10     {
11         static void Main()
12         {
13             Console.Title = "Информация о главном потоке программы";
14
15             Thread thread = Thread.CurrentThread;
16             thread.Name = "MyThread";
17             Console.WriteLine(@"Имя домена приложения: {0}
18 ID контекста: {1}
19 Имя потока: {2}
20 Запущен ли поток? {3}
21 Приоритет потока: {4}
22 Состояние потока: {5}",
23 Thread.GetDomain().FriendlyName, Thread.CurrentContext.ContextID, thread.Name, thread.IsAlive, thread.Priority, thread.ThreadState);
24             Console.ReadLine();
25         }
26     }
27 }

```



Хотя этот код более-менее очевиден, обратите внимание, что класс `Thread` поддерживает свойство по имени `Name`. Если не установить его значение явно, то `Name` вернет пустую строку. Присваивание дружественного имени конкретному объекту `Thread` может значительно упростить отладку. Во время сеанса отладки в Visual Studio 2010 можно открыть окно `Threads` (Потоки), выбрав пункт меню `Debug --> Windows --> Threads` (Отладка --> Окна --> Потоки).

Обратите внимание, что в типе `Thread` определено свойство по имени `Priority`. По умолчанию все потоки имеют уровень приоритета `Normal`. Однако это можно изменить в любой момент жизненного цикла потока, используя свойство `Thread.Priority` и связанное с ним перечисление `System.Threading.ThreadPriority`:

```

public enum ThreadPriority
{
    Lowest,
    BelowNormal,
    Normal, // Значение по умолчанию.
    AboveNormal,
    Highest
}

```


Справочный материал

При установке уровня приоритета потока равным значению, отличному от принятого по умолчанию (`ThreadPriority.Normal`), следует иметь в виду, что это не предоставляет прямого контроля над тем, как планировщик потоков будет переключать потоки между собой. На самом деле уровень приоритета потока предоставляет CLR подсказку относительно важности действий потока. Таким образом, поток с уровнем приоритета `ThreadPriority.Highest` не обязательно гарантированно получит наивысший приоритет.

Опять-таки, если планировщик потоков занят решением определенной задачи (например, синхронизацией объекта, переключением потоков или их перемещением), то уровень приоритета, скорее всего, будет соответствующим образом изменен. Однако, как бы то ни было, среда CLR прочитает эти значения и проинструктирует планировщик потоков о том, как наилучшим образом выделять порции времени. Потоки с идентичным уровнем приоритета должны получать одинаковый объем времени на выполнение своей работы.

В большинстве случаев редко требуется (если вообще требуется) напрямую менять уровень приоритета потока. Теоретически можно повысить уровень приоритета для множества потоков, тем самым предотвращая выполнение низкоприоритетных потоков на их запрошенных уровнях (поэтому будьте осторожны).

При программном создании дополнительных потоков для выполнения некоторой единицы работы необходимо следовать строго регламентированному процессу:

- Создать метод, который будет точкой входа для нового потока.
- Создать новый делегат `ParametrizedThreadStart` (или `ThreadStart`), передав конструктору адрес метода, определенного на предыдущем шаге.
- Создать объект `Thread`, передав в качестве аргумента конструктора `ParametrizedThreadStart/ThreadStart`.
- Установить начальные характеристики потока (имя, приоритет и т.п.).
- Вызвать метод `Thread.Start()`. Это запустит поток на методе, который указан делегатом, созданным на втором шаге, как только это будет возможно.

Согласно второму шагу, можно использовать два разных типа делегатов для "указания" метода, который выполнит вторичный поток. Делегат `ThreadStart` относится к пространству имен `System.Threading`, начиная с .NET 1.0, и он может указывать на любой метод, не принимающий аргументов и ничего не возвращающий. Этот делегат пригодится, когда метод предназначен просто для запуска в фоновом режиме, без какого-либо дальнейшего взаимодействия.

Очевидное ограничение `ThreadStart` связано с невозможность передавать ему параметры для обработки. Тем не менее, тип делегата `ParametrizedThreadStart` позволяет передать единственный параметр типа `System.Object`. Учитывая, что с помощью `System.Object` представляется все, что угодно, можно передать любое количество параметров через специальный класс или структуру. Однако имейте в виду, что делегат `ParametrizedThreadStart` может указывать только на методы, возвращающие `void`.

Делегат ThreadStart

Предположим, что есть консольное приложение, которое позволяет конечному пользователю выбирать, будет ли приложение выполнять свою работу в единственном первичном потоке либо распределит рабочую нагрузку на два отдельных потока выполнения.

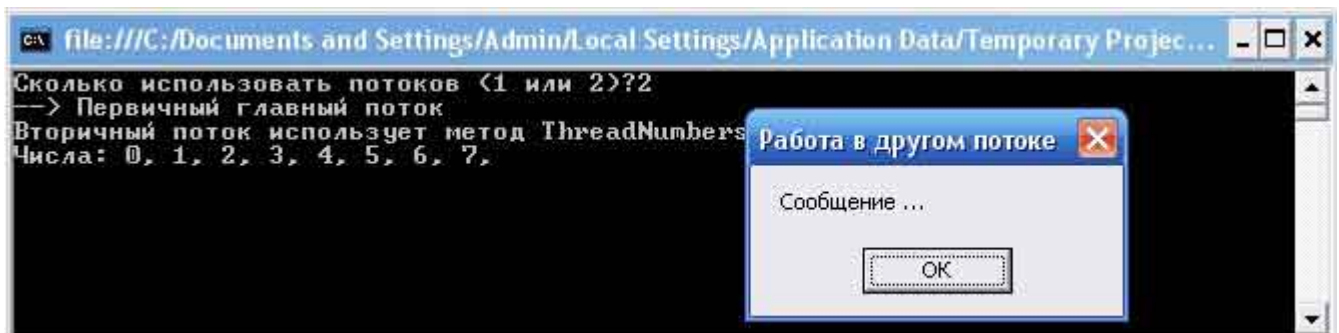
После импортирования пространства имен System.Threading следующий шаг заключается в определении метода для выполнения работы (возможного) вторичного потока. Чтобы сосредоточиться на механизме построения многопоточных программ, этот метод будет просто выводить на консоль последовательность чисел, приостанавливаясь примерно на 2 секунды на каждом шаге.

Задание 3

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading;
6  using System.Windows;
7
8  namespace ConsoleApplication1
9  {
10     public class MyThread
11     {
12         public void ThreadNumbers()
13         {
14             // Информация о потоке
15             Console.WriteLine("{0} поток использует метод ThreadNumbers", Thread.CurrentThread.Name);
16             // Выводим числа
17             Console.Write("Числа: ");
18             for (int i = 0; i < 10; i++)
19             {
20                 Console.Write(i + ", ");
21                 // Используем задержку
22                 Thread.Sleep(3000);
23             }
24             Console.WriteLine();
25         }
26     }
27
28     class Program
29     {
30         static void Main()
31         {
32             Console.Write("Сколько использовать потоков (1 или 2)?");
33             string number = Console.ReadLine();
34
35             Thread mythread = Thread.CurrentThread;
36             mythread.Name = "Первичный";
37
38             // Выводим информацию о потоке
39             Console.WriteLine("--> {0} главный поток", Thread.CurrentThread.Name);
40             MyThread mt = new MyThread();
41
42             switch (number)
43             {
44                 case "1":
45                     mt.ThreadNumbers();
46                     break;
47                 case "2":
48                     // Создаем поток
49                     Thread backgroundThread = new Thread(new ThreadStart(mt.ThreadNumbers));
50                     backgroundThread.Name = "Вторичный";
51                     backgroundThread.Start();
52                     break;
53                 default:
54                     Console.WriteLine("использую 1 поток");
55                     goto case "1";
56             }
57             MessageBox.Show("Сообщение ...", "Работа в другом потоке");
58             Console.ReadLine();
59         }
60     }
61 }
```

Внутри Main() сначала пользователю предлагается решить, сколько потоков применять для выполнения работы приложения: один или два. Если пользователь запрашивает один поток, нужно просто вызвать метод ThreadNumbers() внутри первичного потока. Если же пользователь отдает предпочтение двум потокам, необходимо создать делегат ThreadStart, указывающий на ThreadNumbers(), передать объект делегата конструктору нового объекта Thread и вызвать метод Start(), информируя среду CLR, что этот поток готов к обработке.

Если теперь запустить эту программу в одном потоке, обнаружится, что финальное окно сообщения не отображает сообщения, пока вся последовательность чисел не будет выведена на консоль. Поскольку после вывода каждого числа установлена пауза примерно в 2 секунды, это создаст не слишком приятное впечатление у пользователя. Однако в случае выбора двух потоков окно сообщения отображается немедленно, поскольку для вывода чисел на консоль выделен отдельный уникальный объект Thread:



Зачастую в многопоточной программе требуется, чтобы основной поток был последним потоком, завершающим ее выполнение. Формально программа продолжает выполняться до тех пор, пока не завершатся все ее приоритетные потоки. Поэтому требовать, чтобы основной поток завершал выполнение программы, совсем не обязательно. Тем не менее этого правила принято придерживаться в многопоточном программировании, поскольку оно явно определяет конечную точку программы.

Справочный материал

Делегат ParametrizedThreadStart

Вспомните, что делегат ThreadStart может указывать только на методы, возвращающие void и не имеющие аргументов. Во многих случаях это подходит, но если нужно передать данные методу, выполняющемуся во вторичном потоке, то придется использовать тип делегата ParametrizedThreadStart.

Пример:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading;
6
7  namespace ConsoleApplication1
8  {
9      public class Params
10     {
11         public int a, b;
12         public Params(int a, int b)
13         {
14             this.a = a;
15             this.b = b;
16         }
17     }
18
19     class Program
20     {
21         static void Add(object obj)
22         {
23             if (obj is Params)
24             {
25                 Console.WriteLine("ID потока метода Add(): " + Thread.CurrentThread.ManagedThreadId);
26                 Params pr = (Params)obj;
27                 Console.WriteLine("{0} + {1} = {2}", pr.a, pr.b, pr.a+pr.b);
28             }
29         }
30
31         static void Main()
32         {
33             Console.WriteLine("Главный поток. ID: " + Thread.CurrentThread.ManagedThreadId);
34
35             Params pm = new Params(10, 10);
36             Thread t = new Thread(new ParameterizedThreadStart(Add));
37             t.Start(pm);
38
39             // Задержка
40             Thread.Sleep(5);
41             Console.ReadLine();
42         }
43     }
44 }
```

Класс AutoResetEvent

В этих первых примерах для того, чтобы заставить первичный поток подождать, пока вторичный поток завершится, применялось несколько грубых способов. Во время рассмотрения асинхронных делегатов в качестве переключателя использовалась простая переменная булевского типа. Однако это решение не может быть рекомендуемым, поскольку оба потока обращаются к одному и тому же элементу данных, что может привести к его повреждению.

Более безопасной, хотя также нежелательной альтернативой может быть вызов `Thread.Sleep()` на определенный период времени. Проблема в том, что нет желания ждать больше, чем необходимо.

Простой и безопасный к потокам способ заставить один поток ожидать завершения другого потока, предусматривает использование класса `AutoResetEvent`. В потоке, который должен ждать (таком как поток метода `Main()`), создадим экземпляр этого класса и передадим конструктору `false`, указав, что уведомления пока не было. В точке, где требуется ожидать, вызовем метод `WaitOne()`. Ниже приведен измененный класс `Program`, который делает все это, используя статическую

переменную-член `AutoResetEvent`:

```
1  class Program
2  {
3      private static AutoResetEvent waitHandle = new AutoResetEvent(false);
4
5      static void Main()
6      {
7          Console.WriteLine("Главный поток. ID: " + Thread.CurrentThread.ManagedThreadId);
8
9          Params pm = new Params(10, 10);
10         Thread t = new Thread(new ParameterizedThreadStart(Add));
11         t.Start(pm);
12
13         // Задержка
14         Thread.Sleep(5);
15
16         // Ждем уведомления
17         waitHandle.WaitOne();
18         Console.WriteLine("Все потоки завершились");
19
20         Console.ReadLine();
21     }
22
23     static void Add(object obj)
24     {
25         if (obj is Params)
26         {
27             Console.WriteLine("ID потока метода Add(): " + Thread.CurrentThread.ManagedThreadId);
28             Params pr = (Params)obj;
29             Console.WriteLine("{0} + {1} = {2}", pr.a, pr.b, pr.a + pr.b);
30
31             // Сообщить другому потоку о завершении работы
32             waitHandle.Set();
33         }
34     }
35 }
```

Потоки переднего плана и фоновые потоки

Теперь, когда известно, как создавать новые потоки выполнения программно с помощью типов из пространства имен `System.Threading`, давайте проясним разницу между потоками переднего плана и фоновыми потоками.

- Потоки переднего плана (foreground threads) обеспечивают предохранение текущего приложения от завершения. Среда CLR не остановит приложение (что означает выгрузку текущего домена приложения) до тех пор, пока не будут завершены все потоки переднего плана.

- Фоновые потоки (background threads) воспринимаются средой CLR как расширяемые пути выполнения, которые в любой момент времени могут игнорироваться (даже если они в текущее время заняты выполнением некоторой части работы). Таким образом, если все потоки переднего плана прекращаются, то все фоновые потоки автоматически уничтожаются при выгрузке домена приложения.

Важно отметить, что потоки переднего плана и фоновые потоки — это не синонимы первичных и рабочих потоков. По умолчанию каждый поток, создаваемый через метод `Thread.Start()`, автоматически становится потоком переднего плана. Это означает, что домен приложения не выгрузится до тех пор, пока все потоки выполнения не завершат свою часть работы. В большинстве случаев именно такое поведение и нужно.

Чтобы создать фоновый поток необходимо установить свойство `IsBackground` в `true`.

Приоритеты потоков

Как упоминалось ранее, за планирование потоков к запуску отвечает

операционная система. На этот процесс планирования можно влиять, назначая потокам приоритеты. Прежде чем менять приоритет, нужно разобраться в том, как функционирует планировщик потоков. Операционная система планирует выполнение потоков на основе их приоритетов. Поток с наивысшим приоритетом начинает выполняться в ЦП первым. Поток прекращает выполнение и освобождает ЦП, если ему требуется ожидание какого-то ресурса.

Есть несколько причин для перехода потока в режим ожидания. Например, это может происходить из-за получения команды на засыпание, из-за необходимости ожидать завершения дисковых операций ввода-вывода, поступление сетевого пакета и т.п. Если поток не освобождает ЦП самостоятельно, его вытесняет планировщик потоков. Если поток имеет выделенный квант времени, он может использовать ЦП непрерывно.

Если выполняется несколько потоков с одинаковым приоритетом, каждый из которых ожидает получения доступа к ЦП, планировщик потоков применяет алгоритм кругового обслуживания, предоставляя этим потокам доступ к ЦП по очереди. В случае вытеснения поток помещается в конец очереди.

Алгоритм кругового обслуживания и кванты времени применяются только тогда, когда выполняется множество потоков с одинаковым приоритетом. Приоритет является динамическим. Если поток интенсивно использует ЦП (постоянно требует доступа к ЦП без перерывов на ожидание ресурсов), его приоритет понижается до уровня базового приоритета, который был определен с данным потоком. Если поток ожидает какой-то ресурс, поток получает "форсаж" приоритета, и его приоритет повышается. Благодаря "форсажу" вероятность того, что поток получит доступ к ЦП в следующий раз, когда завершится ожидание, значительно увеличивается.

В классе Thread базовый приоритет потока устанавливается в свойстве Priority. Допустимые значения определены в перечислении ThreadPriority. Эти значения представляют различные уровни приоритета и выглядят следующим образом: Highest, AboveNormal, Normal, BelowNormal и Lowest.

При назначении потоку более высокого приоритета следует соблюдать осторожность, поскольку это уменьшает шансы на выполнение для других потоков. Если это настоятельно необходимо, то лучше изменять приоритет только на короткое время.

Управление потоками

Поток создается за счет вызова метода Start() объекта Thread. Однако после вызова метода Start() новый поток все еще пребывает не в состоянии Running, а в состоянии Unstarted. В состояние Running поток переходит сразу после того, как планировщик потоков операционной системы выберет его для выполнения. Информация о текущем состоянии потока доступна через свойство Thread.ThreadState.

С помощью метода Thread.Sleep() поток можно перевести в состояние WaitSleepJoin и при этом указать, через какой промежуток времени поток должен возобновить работу.

Чтобы остановить поток, необходимо вызвать метод Thread.Abort(). При вызове этого метода в соответствующем потоке генерируется исключение типа ThreadAbortException. В случае если для этого исключения предусмотрен обработчик, перед завершением поток сможет выполнить необходимые операции по

очистке. Чтобы продолжить выполнение потока после выдачи исключения `ThreadAbortException`, следует вызвать метод `Thread.ResetAbort()`. Состояние потока, получающего запрос на немедленное прекращение, изменяется с `AbortRequested` на `Aborted`, если поток не производит сброс.

Если необходимо дождаться завершения работы потока, можно вызвать метод `Thread.Join()`. Этот метод блокирует текущий поток и переводит его в состояние `WaitSleepJoin` до тех пор, пока не будет завершен присоединенный к нему поток.

Многопоточное программирование является далеко не простой задачей. При запуске множества потоков, получающих доступ к одним и тем же данным, могут возникать трудно выявляемые проблемы. То же самое верно в случае применения задач, `Parallel LINQ` и класса `Parallel`. Во избежание сложностей нужно изучить особенности обеспечения синхронизации и проблемы, которые могут возникать в случае использования множества потоков. В настоящей статье рассматриваются две такие проблемы — состязания за ресурсы и взаимоблокировки, а так же описываются проблемы параллелизма.

Состязания за ресурсы

Состязание за ресурсы может возникать в случае, если два или более потоков получают доступ к одним и тем же объектам, а доступ к совместно используемому состоянию не синхронизируется.

Чтобы продемонстрировать состязание за ресурсы, ниже приведен пример, в котором определяется класс `StateObject` с полем `int` и методом `ChangeState`. В реализации `ChangeState` значение `state` проверяется на предмет равенства 5. Если это так, выполняется инкремент. Следующий оператор `Trace.Assert` немедленно проверяет, действительно ли `state` теперь имеет значение 6.

Кажется очевидным, что после инкремента переменной, имеющей значение 5, она должна быть равна 6. Однако это необязательно так. Например, если один поток только что выполнил оператор `if (state == 5)`, планировщик может вытеснить его и запустить еще один поток. Второй поток попадет в тело `if` и, поскольку в переменной состояния по-прежнему содержится значение 5, оно будет инкрементировано до 6. После этого снова настанет черед выполнения первого потока, в результате чего в следующем операторе значение переменной состояния будет увеличено до 7. Именно здесь и возникает состязание за ресурсы с выводом соответствующего сообщения:

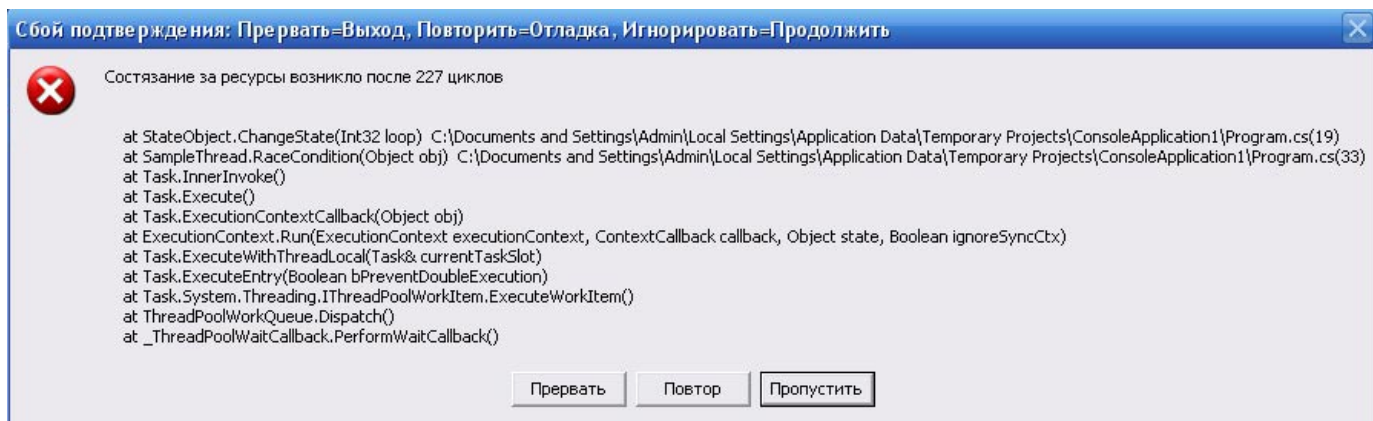
```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading;
6  using System.Diagnostics;
7  using System.Threading.Tasks;
8
9  namespace ConsoleApplication1
10 {
11     public class StateObject
12     {
13         private int state = 5;
14         public void ChangeState(int loop)
15         {
16             if (state == 5)
17             {
18                 state++;
19                 Trace.Assert(state == 6, "Состязание за ресурсы возникло после " + loop + " циклов");
20             }
21             state = 5;
22         }
23     }
24
25     public class SampleThread
26     {
27         public void RaceCondition(object obj)
28         {
29             Trace.Assert(obj is StateObject, "obj должен иметь тип StateObject");
30             StateObject state = obj as StateObject;
31             int i = 0;
32             while (true)
33             {
34                 state.ChangeState(i++);
35             }
36         }
37
38         class Program
39         {
40             static void Main()
41             {
42                 var state = new StateObject();
43                 for (int i = 0; i < 20; i++)
44                     new Task(new SampleThread().RaceCondition, state).Start();
45                 Thread.Sleep(1000);
46             }
47         }
48     }
49 }

```

После запуска этой программы можно будет увидеть, как возникают состязания за ресурсы. То, сколько времени пройдет до возникновения первого состязания за ресурсы, зависит от используемой системы и компоновки программы – окончательной или отладочной. В случае если она компоновалась как окончательная версия, проблема будет возникать чаще, потому что код оптимизирован. Если в системе установлено несколько ЦП либо двух- или четырехядерные ЦП, на которых множество потоков могут выполняться одновременно, проблема тоже будет возникать чаще, чем в системе с одноядерным ЦП.

Из-за вытесняющей многозадачности в системе с одноядерным ЦП состязания также возникают, но не так часто. Ниже показан пример того, как может выглядеть выдаваемое программой сообщение. Здесь это сообщение информирует о том, что состязание за ресурсы возникло после 227 циклов. При каждом запуске приложения результаты будут выглядеть по-разному:



Избежать возникновения данной проблемы можно, заблокировав разделяемый объект. Это делается с помощью оператора `lock`. Внутри блока кода, отвечающего за блокировку объекта состояния, может попадать только один поток. Из-за того, что этот объект разделяется среди всех потоков, в случае, если какой-то один из потоков уже заблокировал его, другой поток при достижении отвечающего за блокировку блока кода должен остановиться и ожидать своей очереди.

При получении блокировки поток вступает во владение ею и снимает ее при достижении конца отвечающего за блокировку блока кода. Когда каждый поток, изменяющий объект, на который ссылается переменная состояния, использует блокировку, проблема с состязанием за ресурсы больше не возникает.

Слишком большое количество блокировок тоже может приводить к проблемам, например, к взаимоблокировке. Взаимоблокировкой (deadlock) называется ситуация, когда как минимум два потока останавливаются и ожидают друг от друга снятия блокировки. Поскольку оба потока ожидают друг от друга выполнения соответствующего действия, получается, что они блокируют друг друга, из-за чего их ожидание может длиться бесконечно.

При построении многопоточного приложения необходимо гарантировать, что любая часть разделяемых данных защищена от возможности изменения их значений множеством потоков. Учитывая, что все потоки в `AppDomain` имеют параллельный доступ к разделяемым данным приложения, представьте, что может случиться, если несколько потоков одновременно обратятся к одному и тому же элементу данных. Поскольку планировщик потоков случайным образом будет приостанавливать их работу, что если поток А будет прерван до того, как завершит свою работу? А вот что: поток В после этого прочтет нестабильные данные.

Задание 4

```
1  using System;
2  using System.Threading;
3
4  namespace ConsoleApplication1
5  {
6      public class MyTheard
7      {
8          public void ThreadNumbers()
9          {
10             // Информация о потоке
11             Console.WriteLine("{0} поток использует метод ThreadNumbers", Thread.CurrentThread.Name);
12             // Выводим числа
13             Console.Write("Числа: ");
14             for (int i = 0; i < 10; i++)
15             {
16                 Random rand = new Random();
17                 Thread.Sleep(1000 * rand.Next(5));
18                 Console.Write(i + ", ");
19             }
20             Console.WriteLine();
21         }
22     }
23
24     class Program
25     {
26         static void Main()
27         {
28             MyTheard mt = new MyTheard();
29
30             // Создаем 10 потоков
31             Thread[] threads = new Thread[10];
32
33             for (int i = 0; i < 10; i++)
34             {
35                 threads[i] = new Thread(new ThreadStart(mt.ThreadNumbers));
36                 threads[i].Name = string.Format("Работает поток: #{0}", i);
37             }
38
39             // Запускаем все потоки
40             foreach (Thread t in threads)
41                 t.Start();
42
43             Console.ReadLine();
44         }
45     }
46 }
```

Прежде чем посмотреть на тестовые запуски, давайте еще раз проясним проблему. Первичный поток внутри этого домена приложений начинает свое существование, порождая десять вторичных рабочих потоков. Каждый рабочий поток должен вызвать метод `ThreadNumbers()` на одном и том же экземпляре `MyTheard`. Учитывая, что никаких мер для блокировки разделяемых ресурсов этого объекта (консоли) не предпринималось, есть хороший шанс, что текущий поток будет отключен, прежде чем метод `ThreadNumbers()` сможет напечатать полные результаты. Поскольку в точности не известно, когда это может случиться (и может ли вообще), будут получаться непредвиденные результаты. Например, может появиться следующий вывод:

Следует, однако, иметь в виду, что блокируемый объект не должен быть общедоступным, так как в противном случае он может быть заблокирован из другого, неконтролируемого в программе фрагмента кода и в дальнейшем вообще не разблокируется.

В прошлом для блокировки объектов очень часто применялась конструкция `lock (this)`. Но она пригодна только в том случае, если `this` является ссылкой на закрытый объект. В связи с возможными программными и концептуальными ошибками, к которым может привести конструкция `lock (this)`, применять ее больше не рекомендуется. Вместо нее лучше создать закрытый объект, чтобы затем заблокировать его.

Задание 5

Модифицировать предыдущий пример добавив в него синхронизацию:

```
public class MyTheard
{
    private object threadLock = new object();

    public void ThreadNumbers()
    {
        // Используем маркер блокировки
        lock (threadLock)
        {
            // Информация о потоке
            Console.WriteLine("{0} поток использует метод ThreadNumbers", Thread.CurrentThread.Name);
            // Выводим числа
            Console.Write("Числа:");
            for (int i = 0; i < 10; i++)
            {
                Random rand = new Random();
                Thread.Sleep(1000 * rand.Next(5));
                Console.Write(i + ", ");
            }
            Console.WriteLine();
        }
    }
}
```

Как только поток войдет в контекст `lock`, маркер блокировки (в данном случае – текущий объект) станет недоступным другим потокам до тех пор, пока блокировка не будет снята по выходе из контекста `lock`. Таким образом, если поток А захватит маркер блокировки, другие потоки не смогут войти ни в один из контекстов, использующих тот же маркер, до тех пор, пока поток А не освободит его.

Чтобы блокировать код в статическом методе, нужно объявить приватную статическую переменную-член, которая будет служить в качестве маркера блокировки. Если теперь запустить приложение, можно увидеть, что каждый поток получил возможность выполнить свою работу до конца:

```
C:\Users\komputer-pc\Desktop\prob\Zadachi C#\robnik2\robnik2\bin\Debug\r... - [X]
Работает поток: #0 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #1 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #2 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #3 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #4 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #5 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #6 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #7 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #8 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Работает поток: #9 поток использует метод ThreadNumbers
Числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Класс `Interlocked` позволяет создавать простые операторы для атомарных операций с переменными. Например, операция `i++` не является безопасной в отношении потоков. Она подразумевает извлечение значения из памяти, увеличение этого значения на 1 и его обратное сохранение в памяти. Такие операции могут прерываться планировщиком потоков. Класс `Interlocked` предоставляет методы, позволяющие выполнять инкремент, декремент, обмен и считывание значений в безопасной к потокам манере.

Применение класса `Interlocked` является гораздо более быстрым подходом по сравнению с остальными приемами по обеспечению синхронизации. Однако пользоваться им можно для устранения только простых последствий синхронизации.

Например, вместо того чтобы применять оператор `lock` для блокирования доступа к переменной при установке для нее нового значения в случае, если ее текущим значением является `null`, можно воспользоваться классом `Interlocked`, что гораздо быстрее. Ниже представлены основные члены данного класса:

Название	Назначение
<code>CompareExchange()</code>	Безопасно проверяет два значения на эквивалентность. Если они эквивалентны, изменяет одно из значений на третье
<code>Decrement()</code>	Безопасно уменьшает значение на 1
<code>Exchange()</code>	Безопасно меняет два значения местами
<code>Increment()</code>	Безопасно увеличивает значение на 1

Хотя это не видно сразу, процесс атомарного изменения отдельного значения довольно часто применяется в многопоточной среде. Предположим, что имеется метод по имени `AddOne()`, который увеличивает целочисленную переменную-название по имени `intVal`. Вместо написания кода синхронизации вроде следующего:

```

public void AddOne()
{
    lock(myLockToken)
    {
        intVal++;
    }
}

```

можно воспользоваться статическим методом `Interlocked.Increment()` и в результате упростить код. Этому методу нужно передать по ссылке переменную для увеличения. Обратите внимание, что метод `Increment()` не только изменяет значение входного параметра, но также возвращает полученное новое значение:

```

public void AddOne()
{
    int newVal = Interlocked.Increment(ref intVal);
}

```

В дополнение к `Increment` и `Decrement` тип `Interlocked` позволяет автоматически присваивать числовые и объектные данные. Например, чтобы присвоить значение 83 переменной-члену, можно обойтись без явного оператора `lock` (или явной логики `Monitor`) и применить вместо этого метод `Interlock.Exchange()`:

Класс `Monitor`

Компилятор C# преобразует оператор `lock` в код, использующий класс `Monitor`. Например, показанный ниже оператор `lock`:

```

lock (obj)
{
    // синхронизированная область для obj
}

```

будет преобразован в код, который вызывает метод `Enter()` и ожидает, пока поток не получит объектную блокировку. В каждый момент времени только один поток может быть владельцем объектной блокировки. После получения блокировки поток сможет входить в синхронизируемый раздел. Метод `Exit()` класса `Monitor` позволяет снимать блокировку.

Компилятор помещает вызов метода `Exit()` в обработчик `finally` блока `try`, чтобы блокировка снималась даже в случае генерации исключения:

```

Monitor.Enter(obj);
try
{
    // синхронизированная область для obj
}
finally
{
    Monitor.Exit(obj);
}

```

Класс `Monitor` обладает одним важным преимуществом по сравнению с оператором `lock` в C#: он позволяет добавлять значение тайм-аута для ожидания получения блокировки. Таким образом, вместо того, чтобы ожидать блокировку до бесконечности, можно вызвать метод `TryEnter` и передать в нем значение тайм-аута, указывающее, сколько максимум времени должно ожидаться получение блокировки.

Когда блокировка `obj` получена, метод `TryEnter()` устанавливает булевский параметр `ref` в `true` и производит синхронизированный доступ к состоянию, охраняемому объектом `obj`. Если `obj` блокируется другим потоком на протяжении более 500 миллисекунд, то `TryEnter()` устанавливает переменную `lockTaken` в `false` и поток больше не ожидает, а используется для выполнения другой работы. Возможно, позже поток попытается получить блокировку еще раз.

Атрибут `[Synchronization]`

Последний из примитивов синхронизации, которые здесь рассматриваются — это атрибут `[Synchronization]`, который является членом пространства имен `System.Runtime.Remoting.Contexts`. Этот атрибут уровня класса эффективно блокирует весь код членов экземпляра объекта, обеспечивая безопасность в отношении потоков.

Когда среда CLR размещает объекты, снабженные атрибутами `[Synchronization]`, она помещает объект в контекст синхронизации. Объекты, которые не должны выходить за границы контекста, должны наследоваться от `ContextBoundObject`. Поэтому, чтобы сделать класс `MyThread` безопасным к потокам (без явного написания кода внутри членов класса), необходимо модифицировать его следующим образом:

```

using System.Runtime.Remoting.Contexts;
...
// Все методы MyThread теперь безопасны к потокам!
[Synchronization]
public class MyThread : ContextBoundObject
{
    public void ThreadNumbers()
    {...}
}

```


В некоторых отношениях этот подход выглядит как "ленивый" способ написания безопасного к потокам кода, учитывая, что не приходится углубляться в детали относительно того, какие именно аспекты типа действительно манипулируют чувствительными к потокам данными. Однако главный недостаток этого подхода состоит в том, что даже если определенный метод не использует чувствительные к потокам данные, CLR будет по-прежнему блокировать вызовы этого метода. Очевидно, что это приведет к деградации общей функциональности типа, поэтому используйте такую технику с осторожностью.

Рассмотрим следующую ситуацию. Поток T выполняется в кодовом блоке lock, и ему требуется доступ к ресурсу R, который временно недоступен. Что же тогда делать потоку T? Если поток T войдет в организованный в той или иной форме цикл опроса, ожидая освобождения ресурса R, то тем самым он свяжет соответствующий объект, блокируя доступ к нему других потоков. Это далеко не самое оптимальное решение, поскольку оно лишает отчасти преимуществ программирования для многопоточной среды.

Более совершенное решение заключается в том, чтобы временно освободить объект и тем самым дать возможность выполняться другим потокам. Такой подход основывается на некоторой форме сообщения между потоками, благодаря которому один поток может уведомлять другой о том, что он заблокирован и что другой поток может возобновить свое выполнение. Сообщение между потоками организуется в C# с помощью методов Wait(), Pulse() и PulseAll().

Методы Wait(), Pulse() и PulseAll() определены в классе Monitor и могут вызываться только из заблокированного фрагмента блока. Они применяются следующим образом. Когда выполнение потока временно заблокировано, он вызывает метод Wait(). В итоге поток переходит в состояние ожидания, а блокировка с соответствующего объекта снимается, что дает возможность использовать этот объект в другом потоке. В дальнейшем ожидающий поток активизируется, когда другой поток войдет в аналогичное состояние блокировки, и вызывает метод Pulse() или PulseAll().

При вызове метода Pulse() возобновляется выполнение первого потока, ожидающего своей очереди на получение блокировки. А вызов метода PulseAll() сигнализирует о снятии блокировки всем ожидающим потокам.

Ниже приведены две наиболее часто используемые формы метода Wait():

public static bool Wait(object obj)

public static bool Wait(object obj, int миллисекунд_простоя)

В первой форме ожидание длится вплоть до уведомления об освобождении объекта, а во второй форме — как до уведомления об освобождении объекта, так и до истечения периода времени, на который указывает количество миллисекунд_простоя.

В обеих формах obj обозначает объект, освобождение которого ожидается. Ниже приведены общие формы методов Pulse() и PulseAll():

public static void Pulse(object obj)

public static void PulseAll(object obj)

где obj обозначает освобождаемый объект.

Если методы Wait(), Pulse() и PulseAll() вызываются из кода, находящегося за пределами синхронизированного кода, например из блока lock, то генерируется исключение SynchronizationLockException.

Для того чтобы стало понятнее назначение методов `Wait()` и `Pulse()`, рассмотрим пример программы, имитирующей тиканье часов и отображающей этот процесс на экране словами "тик" и "так". Для этой цели в программе создается класс `TickTock`, содержащий два следующих метода: `Tick()` и `Tock()`. Метод `Tick()` выводит на экран слово "тик", а метод `Tock()` – слово "так".

Для запуска часов далее в программе создаются два потока: один из них вызывает метод `Tick()`, а другой – метод `Tock()`. Преследуемая в данном случае цель состоит в том, чтобы оба потока выполнялись, поочередно выводя на экран слова "тик" и "так", из которых образуется повторяющийся ряд "тик-так", имитирующий ход часов:

Задание 6

```
1  using System;
2  using System.Threading;
3
4  namespace _Pulse__Wait
5  {
6      class TickTock
7      {
8          private object lockOn = new object();
9
10         public void Tick(bool running)
11         {
12             lock (lockOn)
13             {
14                 if (!running)
15                 {
16                     // Остановить часы
17                     Monitor.Pulse(lockOn);
18                     return;
19                 }
20
21                 Console.Write("Тик ");
22                 // Разрешить выполнение метода Tock()
23                 Monitor.Pulse(lockOn);
24
25                 // Ожидать завершения Tock()
26                 Monitor.Wait(lockOn);
27             }
28         }
29
30         public void Tock(bool running)
31         {
32             lock (lockOn)
33             {
34                 if (!running)
35                 {
36                     Monitor.Pulse(lockOn);
37                     return;
38                 }
39
40                 Console.WriteLine("так");
41                 Monitor.Pulse(lockOn);
42                 Monitor.Wait(lockOn);
43             }
44         }
45     }
```

```

46
47     class MyThread
48     {
49         public Thread thrd;
50         TickTock ttobj;
51
52         // Новый поток
53         public MyThread(string name, TickTock tt)
54         {
55             thrd = new Thread(this.Run);
56             ttobj = tt;
57             thrd.Name = name;
58             thrd.Start();
59         }
60
61         void Run()
62         {
63             if (thrd.Name == "Tick")
64             {
65                 for (int i = 0; i < 5; i++)
66                     ttobj.Tick(true);
67                 ttobj.Tick(false);
68             }
69             else
70             {
71                 for (int i = 0; i < 5; i++)
72                     ttobj.Tock(true);
73                 ttobj.Tock(false);
74             }
75         }
76     }
77
78     class Program
79     {
80         static void Main()
81         {
82             TickTock tt = new TickTock();
83             MyThread mt1 = new MyThread("Tick", tt);
84             MyThread mt2 = new MyThread("Tock", tt);
85             mt1.thrd.Join();
86             mt2.thrd.Join();
87
88             Console.WriteLine("Часы остановлены");
89             Console.ReadLine();
90         }
91     }
92 }

```

```

C:\Users\komputer-pc\Desktop\prob\Zadachi C#\robnik2\robnik2\bin\Debug\r...
Тик так
Тик так
Тик так
Тик так
Тик так
Часы остановлены

```

Справочный материал

Рассмотрим «задание 5» более подробно. В методе `Main()` создается объект `tt` типа `TickTock`, который используется для запуска двух потоков на выполнение. Если в методе `Run()` из класса `MyThread` обнаруживается имя потока `Tick`, соответствующее ходу часов "тик", то вызывается метод `Tick()`. А если это имя потока `Tock`, соответствующее ходу часов "так", то вызывается метод `Tock()`.

Каждый из этих методов вызывается пять раз подряд с передачей логического значения `true` в качестве аргумента. Часы идут до тех пор, пока этим методам передается логическое значение `true`, и останавливаются, как только передается логическое значение `false`. Самая важная часть рассматриваемой здесь программы находится в методах `Tick()` и `Tock()`.

Прежде всего обратите внимание на код метода `Tick()` в блоке `lock`. Напомним, что методы `Wait()` и `Pulse()` могут использоваться только в синхронизированных блоках кода. В начале метода `Tick()` проверяется значение текущего параметра, которое служит явным признаком остановки часов. Если это логическое значение `false`, то часы остановлены. В этом случае вызывается метод `Pulse()`, разрешающий выполнение любого потока, ожидающего своей очереди.

Если же часы идут при выполнении метода `Tick()`, то на экран выводится слово "тик" с пробелом, затем вызывается метод `Pulse()`, а после него — метод `Wait()`. При вызове метода `Pulse()` разрешается выполнение потока для того же самого объекта, а при вызове метода `Wait()` выполнение метода `Tick()` приостанавливается до тех пор, пока метод `Pulse()` не будет вызван из другого потока. Таким образом, когда вызывается метод `Tick()`, отображается одно слово "тик" с пробелом, разрешается выполнение другого потока, а затем выполнение данного метода приостанавливается.

Mutex

Класс `Mutex` (*mutual exclusion* — взаимное исключение или мьютекс) является одним из классов в `.NET Framework`, позволяющих обеспечить синхронизацию среди множества процессов. Он очень похож на класс `Monitor` тем, что тоже допускает наличие только одного владельца. Только один поток может получить блокировку и иметь доступ к защищаемым мьютексом синхронизированным областям кода.

В конструкторе класса `Mutex` указывается, должен ли мьютексом изначально владеть вызывающий поток, и его имя. Кроме того, конструктор позволяет получить информацию о том, существует ли уже такой класс.

Мьютекс представляет собой взаимно исключающий синхронизирующий объект. Это означает, что он может быть получен потоком только по очереди. Мьютекс предназначен для тех ситуаций, в которых общий ресурс может быть одновременно использован только в одном потоке. Допустим, что системный журнал совместно используется в нескольких процессах, но только в одном из них данные могут записываться в файл этого журнала в любой момент времени. Для синхронизации процессов в данной ситуации идеально подходит мьютекс.

У `Mutex` имеется несколько конструкторов. Ниже приведены два наиболее употребительных конструктора:

```
public Mutex()  
public Mutex(bool initiallyOwned)
```

В первой форме конструктора создается мьютекс, которым первоначально никто не владеет. А во второй форме исходным состоянием мьютекса завладевает вызывающий поток, если параметр `initiallyOwned` имеет логическое значение `true`. В противном случае мьютексом никто не владеет.

Для того чтобы получить мьютекс, в коде программы следует вызвать метод `WaitOne()` для этого мьютекса. Метод `WaitOne()` наследуется классом `Mutex` от класса `Thread.WaitHandle`. Метод `WaitOne()` ожидает до тех пор, пока не будет получен мьютекс, для которого он был вызван. Следовательно, этот метод блокирует выполнение вызывающего потока до тех пор, пока не станет доступным указанный мьютекс. Он всегда возвращает логическое значение `true`.

Когда же в коде больше не требуется владеть мьютексом, он освобождается посредством вызова метода `ReleaseMutex()`.

В приведенном ниже примере программы описанный выше механизм синхронизации демонстрируется на практике. В этой программе создаются два потока в виде классов `IncThread` и `DecThread`, которым требуется доступ к общему ресурсу: переменной `SharedRes.Count`. В потоке `IncThread` переменная `SharedRes.Count` инкрементируется, а в потоке `DecThread` — декрементируется. Во избежание одновременного доступа обоих потоков к общему ресурсу `SharedRes.Count` этот доступ синхронизируется мьютексом `Mtx`, также являющимся членом класса `SharedRes`:

Задание 7

```
1  using System;
2  using System.Threading;
3
4  namespace ConsoleApplication1
5  {
6      // В этом классе содержится общий ресурс в виде переменной Count,
7      // а так же мьютекс mtx
8      class SharedRes
9      {
10         public static int Count;
11         public static Mutex mtx = new Mutex();
12     }
13
14     // В этом классе Count инкрементируется
15     class IncThread
16     {
17         int num;
18         public Thread Thrd;
19
20         public IncThread(string name, int n)
21         {
22             Thrd = new Thread(this.Run);
23             num = n;
24             Thrd.Name = name;
25             Thrd.Start();
26         }
27
28         // Точка входа в поток
29         void Run()
30         {
31             Console.WriteLine(Thrd.Name + " ожидает мьютекс");
32
33             // Получить мьютекс
34             SharedRes.mtx.WaitOne();
35
36             Console.WriteLine(Thrd.Name + " получает мьютекс");
37
38             do
39             {
40                 Thread.Sleep(500);
41                 SharedRes.Count++;
42                 Console.WriteLine("В потоке {0}, Count={1}", Thrd.Name, SharedRes.Count);
43                 num--;
44             } while (num > 0);
45         }
46     }
47 }
```

```

45
46     Console.WriteLine(Thrd.Name + " освобождает мьютекс");
47
48     SharedRes.mtx.ReleaseMutex();
49 }
50
51 }
52
53 class DecThread
54 {
55     int num;
56     public Thread Thrd;
57
58     public DecThread(string name, int n)
59     {
60         Thrd = new Thread(new ThreadStart(this.Run));
61         num = n;
62         Thrd.Name = name;
63         Thrd.Start();
64     }
65
66     // Точка входа в поток
67     void Run()
68     {
69         Console.WriteLine(Thrd.Name + " ожидает мьютекс");
70
71         // Получить мьютекс
72         SharedRes.mtx.WaitOne();
73
74         Console.WriteLine(Thrd.Name + " получает мьютекс");
75
76         do
77         {
78             Thread.Sleep(500);
79             SharedRes.Count--;
80             Console.WriteLine("В потоке {0}, Count={1}", Thrd.Name, SharedRes.Count);
81             num--;
82         } while (num > 0);
83
84         Console.WriteLine(Thrd.Name + " освобождает мьютекс");
85
86         SharedRes.mtx.ReleaseMutex();
87     }
88 }
89
90 class Program
91 {
92     static void Main()
93     {
94         IncThread mt1 = new IncThread("Inc thread", 5);
95
96         // разрешить инкрементирующему потоку начаться
97         Thread.Sleep(1);
98
99         DecThread mt2 = new DecThread("Dec thread", 5);
100
101         mt1.Thrd.Join();
102         mt2.Thrd.Join();
103
104         Console.ReadLine();
105     }
106 }

```

```
C:\Users\komputer-pc\Desktop\prob\Zadachi C#\robnik2\robnik2\bin\Debug\r... - □ ×
Dec thread ожидает мьютекс
Inc thread ожидает мьютекс
Inc thread получает мьютекс
в потоке Inc thread, Count=1
в потоке Inc thread, Count=2
в потоке Inc thread, Count=3
в потоке Inc thread, Count=4
в потоке Inc thread, Count=5
Inc thread освобождает мьютекс
Dec thread получает мьютекс
в потоке Dec thread, Count=4
в потоке Dec thread, Count=3
в потоке Dec thread, Count=2
в потоке Dec thread, Count=1
в потоке Dec thread, Count=0
Dec thread освобождает мьютекс
-
```

Справочный материал

Как следует из приведенного выше результата, доступ к общему ресурсу (переменной SharedRes.Count) синхронизирован, и поэтому значение данной переменной может быть одновременно изменено только в одном потоке.

Semaphore

Семафор подобен мьютексу, за исключением того, что он предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам. Поэтому семафор пригоден для синхронизации целого ряда ресурсов. Семафор управляет доступом к общему ресурсу, используя для этой цели счетчик. Если значение счетчика больше нуля, то доступ к ресурсу разрешен. А если это значение равно нулю, то доступ к ресурсу запрещен. С помощью счетчика ведется подсчет количества разрешений. Следовательно, для доступа к ресурсу поток должен получить разрешение от семафора.

Обычно поток, которому требуется доступ к общему ресурсу, пытается получить разрешение от семафора. Если значение счетчика семафора больше нуля, то поток получает разрешение, а счетчик семафора декрементируется. В противном случае поток блокируется до тех пор, пока не получит разрешение. Когда же потоку больше не требуется доступ к общему ресурсу, он высвобождает разрешение, а счетчик семафора инкрементируется. Если разрешения ожидает другой поток, то он получает его в этот момент. Количество одновременно разрешаемых доступов указывается при создании семафора. Так, если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс.

Семафоры особенно полезны в тех случаях, когда общий ресурс состоит из группы или пула ресурсов. Например, пул ресурсов может состоять из целого ряда сетевых соединений, каждое из которых служит для передачи данных. Поэтому потоку, которому требуется сетевое соединение, все равно, какое именно соединение он получит. В данном случае семафор обеспечивает удобный механизм управления доступом к сетевым соединениям.

Семафор реализуется в классе System.Threading.Semaphore, у которого имеется

несколько конструкторов. Ниже приведена простейшая форма конструктора данного класса:

public Semaphore(int initialCount, int maximumCount)

где `initialCount` — это первоначальное значение для счетчика разрешений семафора, т.е. количество первоначально доступных разрешений; `maximumCount` — максимальное значение данного счетчика, т.е. максимальное количество разрешений, которые может дать семафор.

Семафор применяется таким же образом, как и описанный ранее мьютекс. В целях получения доступа к ресурсу в коде программы вызывается метод `WaitOne()` для семафора. Этот метод наследуется классом `Semaphore` от класса `WaitHandle`. Метод `WaitOne()` ожидает до тех пор, пока не будет получен семафор, для которого он вызывается. Таким образом, он блокирует выполнение вызывающего потока до тех пор, пока указанный семафор не предоставит разрешение на доступ к ресурсу.

Если коду больше не требуется владеть семафором, он освобождает его, вызывая метод `Release()`. Ниже приведены две формы этого метода:

public int Release()
public int Release(int releaseCount)

В первой форме метод `Release()` высвобождает только одно разрешение, а во второй форме — количество разрешений, определяемых параметром `releaseCount`. В обеих формах данный метод возвращает подсчитанное количество разрешений, существовавших до высвобождения.

В .NET 4 предлагается два класса с функциональностью семафора: `Semaphore` и `SemaphoreSlim`. Класс `Semaphore` может быть именован, использовать ресурсы в масштабе всей системы и обеспечивать синхронизацию между различными процессами. Класс `SemaphoreSlim` представляет собой облегченную версию класса `Semaphore`, которая оптимизирована для обеспечения более короткого времени ожидания.

События представляют собой еще один ресурс для обеспечения синхронизации в масштабе всей системы.

Для использования системных событий из управляемого кода .NET Framework предлагает классы `ManualResetEvent`, `AutoResetEvent`, `ManualResetEventSlim` и `CountdownEvent`, которые находятся в пространстве имен `System.Threading`. Классы `ManualResetEventSlim` и `CountdownEvent` появились в версии .NET 4.

Эти классы являются производными от класса `EventWaitHandle`, находящегося на верхнем уровне иерархии классов, и применяются в тех случаях, когда один поток ожидает появления некоторого события в другом потоке. Как только такое событие появляется, второй поток уведомляет о нем первый поток, позволяя тем самым возобновить его выполнение.

Ниже приведены конструкторы классов `ManualResetEvent` и `AutoResetEvent`:

public ManualResetEvent(bool initialState)
public AutoResetEvent(bool initialState)

Если в обеих формах параметр `initialState` имеет логическое значение `true`, то о событии первоначально уведомляется. А если он имеет логическое значение `false`, то о событии первоначально не уведомляется.

Применяются события очень просто. Так, для события типа `ManualResetEvent` порядок применения следующий. Поток, ожидающий некоторое событие, вызывает метод `WaitOne()` для событийного объекта, представляющего данное событие. Если событийный объект находится в сигнальном состоянии, то происходит немедленный возврат из метода `WaitOne()`. В противном случае выполнение вызывающего потока приостанавливается до тех пор, пока не будет получено уведомление о событии. Как только событие произойдет в другом потоке, этот поток установит событийный объект в сигнальное состояние, вызвав метод `Set()`. Поэтому метод `Set()` следует рассматривать как уведомляющий о том, что событие произошло.

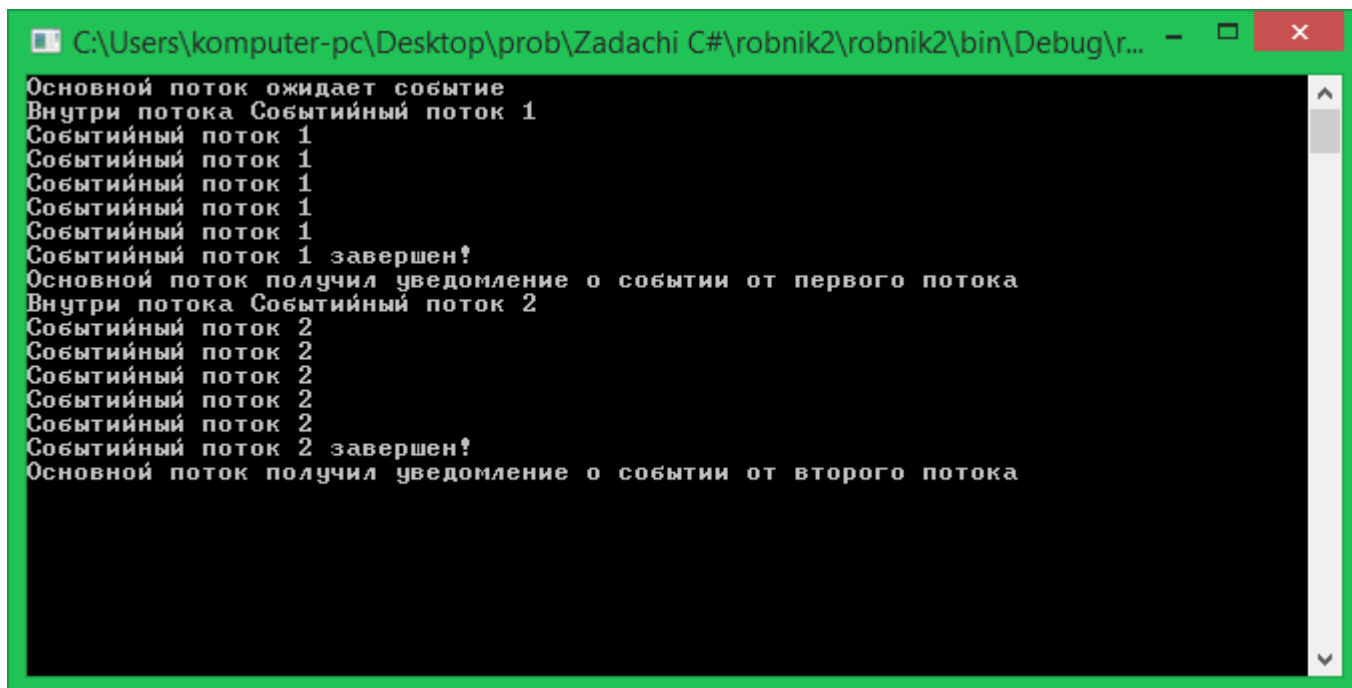
После установки событийного объекта в сигнальное состояние произойдет немедленный возврат из метода `WaitOne()`, и первый поток возобновит свое выполнение. А в результате вызова метода `Reset()` событийный объект возвращается в несигнальное состояние.

Событие `AutoResetEvent` отличается от события типа `ManualResetEvent` лишь способом установки в исходное состояние. Если для события типа `ManualResetEvent` событийный объект остается в сигнальном состоянии до тех пор, пока не будет вызван метод `Reset()`, то для события типа `AutoResetEvent` событийный объект автоматически переходит в несигнальное состояние, как только поток, ожидающий это событие, получит уведомление о нем и возобновит свое выполнение. Поэтому если применяется событие типа `AutoResetEvent`, то вызывать метод `Reset()` необязательно.

Событие `ManualResetEventSlim` переводится в сигнальное состояние вызовом метода `Set()`, а с помощью `Reset()` возвращается обратно в несигнальное состояние. В случае вызова метода `Set()` при наличии множества потоков, ждущих перехода события в сигнальное состояние, ожидание всех этих потоков немедленно прекращается. В случае, если поток просто вызывает метод `WaitOne()`, а событие уже находится в сигнальном состоянии, ожидавший поток может сразу же продолжить работу.

Задание 8

```
1  using System;
2  using System.Threading;
3
4  namespace ConsoleApplication1
5  {
6      class MyThread
7      {
8          public Thread Thrd;
9          ManualResetEvent mre;
10
11         public MyThread(string name, ManualResetEvent evt)
12         {
13             Thrd = new Thread(this.Run);
14             Thrd.Name = name;
15             mre = evt;
16             Thrd.Start();
17         }
18
19         void Run()
20         {
21             Console.WriteLine("Внутри потока " + Thrd.Name);
22
23             for (int i = 0; i < 5; i++)
24             {
25                 Console.WriteLine(Thrd.Name);
26                 Thread.Sleep(500);
27             }
28
29             Console.WriteLine(Thrd.Name + " завершен!");
30
31             // Уведомление о событии
32             mre.Set();
33         }
34     }
35
36     class Program
37     {
38         static void Main()
39         {
40             ManualResetEvent evtObj = new ManualResetEvent(false);
41
42             MyThread mt1 = new MyThread("Событийный поток 1", evtObj);
43
44             Console.WriteLine("Основной поток ожидает событие");
45
46             evtObj.WaitOne();
47
48             Console.WriteLine("Основной поток получил уведомление о событии от первого потока");
49
50             evtObj.Reset();
51
52             mt1 = new MyThread("Событийный поток 2", evtObj);
53
54             evtObj.WaitOne();
55
56             Console.WriteLine("Основной поток получил уведомление о событии от второго потока");
57             Console.ReadLine();
58         }
59     }
60 }
```



```
C:\Users\komputer-pc\Desktop\prob\Zadachi C#\robnik2\robnik2\bin\Debug\r...
Основной поток ожидает событие
Внутри потока Событийный поток 1
Событийный поток 1
Событийный поток 1
Событийный поток 1
Событийный поток 1
Событийный поток 1
Событийный поток 1 завершен!
Основной поток получил уведомление о событии от первого потока
Внутри потока Событийный поток 2
Событийный поток 2
Событийный поток 2
Событийный поток 2
Событийный поток 2
Событийный поток 2
Событийный поток 2 завершен!
Основной поток получил уведомление о событии от второго потока
```

Прежде всего обратите внимание на то, что событие типа `ManualResetEvent` передается непосредственно конструктору класса `MyThread`. Когда завершается метод `Run()` из класса `MyThread`, он вызывает для событийного объекта метод `Set()`, устанавливающий этот объект в сигнальное состояние. В методе `Main()` формируется событийный объект `evtObj` типа `ManualResetEvent`, первоначально устанавливаемый в исходное, несигнальное состояние. Затем создается экземпляр объекта типа `MyThread`, которому передается событийный объект `evtObj`.

После этого основной поток ожидает уведомления о событии. А поскольку событийный объект `evtObj` первоначально находится в несигнальном состоянии, то основной поток вынужден ожидать до тех пор, пока для экземпляра объекта типа `MyThread` не будет вызван метод `Set()`, устанавливающий событийный объект `evtObj` в сигнальное состояние. Это дает возможность основному потоку возобновить свое выполнение.

Затем событийный объект устанавливается в исходное состояние, и весь процесс повторяется, но на этот раз для второго потока. Если бы не событийный объект, то все потоки выполнялись бы одновременно, а результаты их выполнения оказались бы окончательно запутанными. Для того чтобы убедиться в этом, попробуйте закомментировать вызов метода `WaitOne()` в методе `Main()`.

Если бы в рассматриваемой здесь программе событийный объект типа `AutoResetEvent` использовался вместо событийного объекта типа `ManualResetEvent`, то вызывать метод `Reset()` в методе `Main()` не пришлось бы. Ведь в этом случае событийный объект автоматически устанавливается в несигнальное состояние, когда поток, ожидающий данное событие, возобновляет свое выполнение. Для опробования этой разновидности события замените в данной программе все ссылки на объект типа `ManualResetEvent` ссылками на объект типа `AutoResetEvent` и удалите все вызовы метода `Reset()`. Видоизмененная версия программы будет работать так же, как и прежде.

Содержание работы

1. Откройте новый документ Word, выполните настройку документа и заполните необходимую информацию согласно методических указаний. Сохраните документ в папке «Мои документы» с именем «Фамилия, группа, Пр.р.№», не забывайте периодически сохранять документ в процессе выполнения работы.

2. Внимательно изучите краткие теоретические сведения.

3. Оформите предложенный текст в соответствии с требованиями к проекту.

4. Сделайте выводы, подготовьтесь к защите.

Контрольные вопросы:

1. Что такое потоки?

2. Создание потоков?

3. Запуск потоков?

4. Приостановление потоков?

5. Приоритеты потоков?

6. Изменение типов потоков?

7. Свойства потоков?

8. Локальное хранилище потоков.

ПРАКТИЧЕСКАЯ РАБОТА №9-16

Тема: Обмен данными

Цель: Изучить обмен данными.

Оборудование: В соответствии с рабочей программой ПМ.01 «Разработка модулей программного обеспечения для компьютерных систем»:

- Автоматизированные рабочие места на 12-15 обучающихся (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Автоматизированное рабочее место преподавателя (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Проектор и экран;
- Маркерная доска;
- Лицензионное программное обеспечение общего и профессионального назначения.

Справочный материал

Программистам на C# никогда не приходится непосредственно удалять управляемый объект из памяти (в языке C# нет даже ключевого слова вроде delete). Вместо этого объекты .NET размещаются в области памяти, которая называется управляемой кучей (managed heap), откуда они автоматически удаляются сборщиком мусора, когда наступает "определенный момент в будущем".

Рассматривая различие между классами, объектами и ссылками. Вспомните, что класс представляет собой ни что иное, как схему, которая описывает то, каким образом экземпляр данного типа должен выглядеть и вести себя в памяти. Определяются классы в файлах кода (которым по соглашению назначается расширение *.cs). Для примера создадим новый проект типа Console Application (Консольное приложение) на C# и определим в нем следующий простой класс UserInfo:

Задание 1

```
1  using System;
2
3  namespace ConsoleApplication1
4  {
5      class UserInfo
6      {
7          public string Name { get; set; }
8          public byte Age { get; set; }
9
10         public UserInfo() { }
11         public UserInfo(string Name, byte Age)
12         {
13             this.Name = Name;
14             this.Age = Age;
15         }
16
17         public override string ToString()
18         {
19             return String.Format(@"Имя пользователя: {0}
20 Возраст: {1}", Name, Age);
21         }
22     }
23
24     class Program
25     {
26         static void Main()
27         {
28             // Создание нового объекта UserInfo в управляемой куче
29             // Возвращаемая ссылка на этот объект user1:
30             UserInfo User1 = new UserInfo("Alex", 26);
31             Console.WriteLine(User1.ToString());
32             Console.ReadLine();
33         }
34
35         public static void MyUser()
36         {
37             UserInfo ui = new UserInfo();
38         }
39     }
40 }
```

При создании приложений на C# можно смело полагать, что исполняющая среда .NET будет сама заботиться об управляемой куче без непосредственного вмешательства со стороны программиста. На самом деле "золотое правило" по управлению памятью в .NET звучит просто:

Размещайте объект в управляемой куче с использованием ключевого слова `new` и забывайте об этом.

После создания объект будет автоматически удален сборщиком мусора тогда, когда в нем отпадет необходимость. Разумеется, возникает вопрос о том, каким образом сборщик мусора определяет момент, когда в объекте отпадает необходимость? В двух словах на этот вопрос можно ответить так: сборщик мусора удаляет объект из кучи тогда, когда тот становится недостижимым ни в одной части программного кода.

Как станет ясно со временем, программирование в среде с автоматической сборкой мусора значительно облегчает разработку приложений. Программистам на C++ хорошо известно, что если они специально не позаботятся об удалении размещаемых в куче объектов, вскоре обязательно начнут возникать "утечки памяти". На самом деле отслеживание проблем, связанных с утечкой памяти,

является одним из самых длительных (и утомительных) аспектов программирования в неуправляемых средах. Благодаря назначению ответственным за уничтожение объектов сборщика мусора, обязанности по управлению памятью, по сути, сняты с плеч программиста и возложены на CLR-среду.

Справочный материал

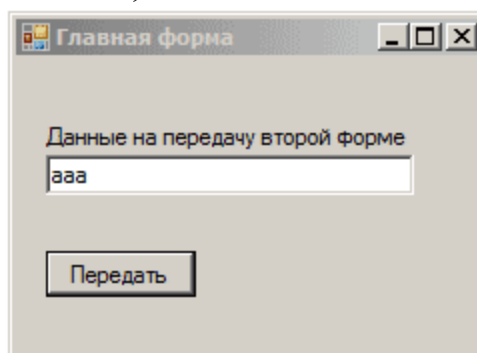
Вопрос, рассматриваемый в данной статье, скорее относится к теме построения архитектуры приложения, а не конкретно рассматриваемой проблемы. Передавать данные от одной формы в другую вовсе не составляет труда. Для этого достаточно контрол, данные которого мы хотим получить, сделать открытым, то есть пометить модификатором `public`. Также, возможен и другой вариант. Например, в первой форме мы создаем объект второй формы, передав в конструктор ссылку на себя, то есть, передав из первой формы во вторую ссылку на первую

`SecondForm secondForm = new SecondForm(this);`

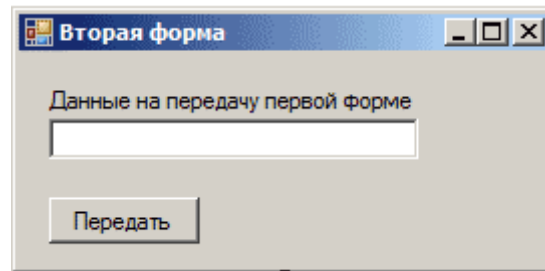
Естественно, перед этим следует позаботиться о создании перегрузки конструктора второй формы.

И такой способ достаточно распространен. Однако, со своей простотой, он несет много потенциальных проблем, главная из которых – нарушение принципа инкапсуляции. Одним словом, вторая форма ничего не должна знать о существовании первой и, уж тем более, не должна иметь возможность влиять на неё.

Решение данной проблемы достаточно простое. Обратимся непосредственно к коду. В дизайнере создаем главную форму (она будет запускаться при запуске приложения). Поместим один `TextBox`, `Label` и `Button`.



По нажатию на кнопку будет открываться вторая форма и текст из текстового поля главной формы передастся в текстовое поле второй формы. Изначально, вторая форма выглядит так:



Аналогично первой, она имеет те же контролы. Больше нам и не надо. Точка входа в приложение запускает главную форму:

Задание 2

```
01. using System;
02. using System.Collections.Generic;
03. using System.Linq;
04. using System.Windows.Forms;
05.
06. namespace From1FormTo2
07. {
08.     static class Program
09.     {
10.         // The main entry point for the application.
11.         [STAThread]
12.         static void Main()
13.         {
14.             Application.EnableVisualStyles();
15.             Application.SetCompatibleTextRenderingDefault(false);
16.             Application.Run(new MainForm());
17.         }
18.     }
19. }
```

Код главной формы выглядит так:

```
01.     using System;
02.     using System.Collections.Generic;
03.     using System.ComponentModel;
04.     using System.Data;
05.     using System.Drawing;
06.     using System.Linq;
07.     using System.Text;
08.     using System.Windows.Forms;
09.
10.     namespace From1FormTo2
11.     {
12.         public partial class MainForm : Form
13.         {
14.             //вторая форма
15.             SecondForm secondForm;
16.
17.             //конструктор
18.             public MainForm()
19.             {
20.                 InitializeComponent();
21.             }
22.
23.             //обработчик события передачи данных
24.             //от главной формы ко второй
25.             private void btn_mainForm_Click(object sender, EventArgs e)
26.             {
27.                 secondForm = new SecondForm(tb_mainForm.Text.Trim());
28.                 secondForm.ShowDialog();
29.
30.                 if (secondForm.DialogResult == DialogResult.OK)
31.                     tb_mainForm.Text = secondForm.ReturnData();
32.             }
33.         }
34.     }
```

Соответственно, не забудьте подключить кнопку на событие **Click**. Здесь, в классе главной формы, есть поле **SecondForm secondForm**, представляющее объект-вторую форму. При нажатии на кнопку «Передать», создается вторая форма (вызывается перегруженный конструктор, его мы еще создадим) и запускается методом **ShowDialog()**. В данном случае нам подходит именно этот метод. При чем, после этого мы обязательно проверяем, не закрыли ли вторую форму, а выполнили клик по её кнопке. Если, на второй форме был выполнен клик по кнопке, значит первая форма должна принять данные от второй. Это происходит путем вызова метода **ReturnData()** у второй формы.

Теперь самое интересное – код второй формы:


```

01. using System;
02. using System.Collections.Generic;
03. using System.ComponentModel;
04. using System.Data;
05. using System.Drawing;
06. using System.Linq;
07. using System.Text;
08. using System.Windows.Forms;
09.
10. namespace From1FormTo2
11. {
12.     public partial class SecondForm : Form
13.     {
14.         //перегруженный конструктор
15.         public SecondForm(string data)
16.         {
17.             InitializeComponent();
18.             tb_secondForm.Text = data;
19.         }
20.
21.         //обработчик события передачи данных
22.         //от второй формы к главной
23.         private void btn_secondForm_Click
24.             (object sender, EventArgs e)
25.         {
26.             this.DialogResult = DialogResult.OK;
27.         }
28.
29.         //открытый метод для доступа к
30.         //текстовому полю данной формы
31.         public string ReturnData()
32.         {
33.             return (tb_secondForm.Text.Trim());
34.         }
35.     }
36. }

```

Как видим, имеется единственная перегрузка конструктора, который принимает тип **string**. Помним, что мы пытаемся передавать текст из **TextBox**. В конструкторе происходит плановая инициализация компонент и установка текста текстового поля в переданное значение от первой формы. Далее, подписавшись на событие **Click** для кнопки второй формы, мы создали обработчик **btn_secondForm_Click**, который и имитирует работу кнопки «Ok» любого диалогового окна. Таким образом, нажимая на кнопке «Отправить» (второй формы), мы приводим в исполнение условие

(secondForm.DialogResult == DialogResult.OK)

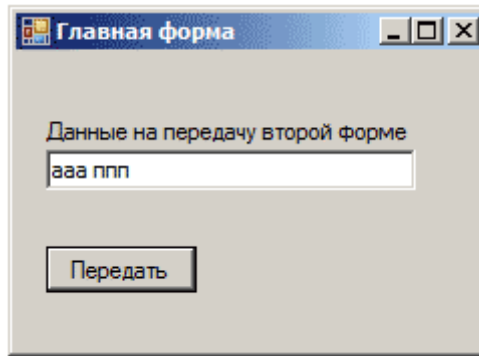
первой формы, ввиду чего, вызывая метод **secondForm.ReturnData()**, мы устанавливаем текстовое поле первой формы в значение текстового поля второй формы.

Работа данного метода, я думаю, уже не требует пояснений. Он просто возвращает текст из единственного текстового поля, при этом, оставляя его приватным.

В итоге, мы передали данные во вторую форму из первой и со второй в первую не нарушая принципы инкапсуляции.

Попробуйте внести текст «aaa» в текстовое поле первой формы и выполнить нажатие на кнопке. Вы увидите в открывшейся второй форме этот текст в её текстовом поле. Попробуйте изменить текст на «aaa ппп» и нажать на кнопку. Вы

увидите как после закрытия второй формы данный текст отобразится в текстовом поле главной формы.



Задание 3

Взаимодействие между различными приложениями может осуществляться по-разному, например, через сокеты. Но в .NET 4.0 была введена новая функциональность, которая представляет собой создание участка общей разделяемой памяти для приложений.

Основной функционал новой технологии заключается в пространстве имен System.IO.MemoryMappedFiles.

Создадим два консольных приложения, одно из которых будет посылать сообщение в общую память, а другое - считывать это сообщение.

Для этого нам потребуются два класса: MemoryMappedFile - он будет создавать участок разделяемой памяти и MemoryMappedViewAccessor - с его помощью мы будем проводить взаимодействие (чтение/запись) с общей памятью. Код приложения, записывающего сообщение в память.

Программа записи:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.IO.MemoryMappedFiles;
6
7  namespace WriteMemoryAp
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             Console.WriteLine("Введите сообщение");
14             //Ввод выражения для записи в общую память
15             char[] message = Console.ReadLine().ToCharArray();
16             //Размер введенного сообщения
17             int size = message.Length;
18
19             //Создание участка разделяемой памяти
20             //Первый параметр - название участка,
21             //второй - длина участка памяти в байтах: тип char занимает 2 байта
22             //плюс четыре байта для одного объекта типа Integer
23             MemoryMappedFile sharedMemory = MemoryMappedFile.CreateOrOpen("MemoryFile", size * 2 + 4);
24             //Создаем объект для записи в разделяемый участок памяти
25             using (MemoryMappedViewAccessor writer = sharedMemory.CreateViewAccessor(0, size * 2 + 4))
26             {
27                 //запись в разделяемую память
28                 //запись размера с нулевого байта в разделяемой памяти
29                 writer.Write(0, size);
30                 //запись сообщения с четвертого байта в разделяемой памяти
31                 writer.WriteArray<char>(4, message, 0, message.Length);
32             }
33
34             Console.WriteLine("Сообщение записано в разделяемую память");
35             Console.WriteLine("Для выхода из программы нажмите любую клавишу");
36             Console.ReadLine();
37         }
38     }
39 }
```

Программа чтения:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.IO.MemoryMappedFiles;
6
7 namespace ReadMemoryAp
8 {
9     class Program
10     {
11         static void Main(string[] args)
12         {
13             //Массив для сообщения из общей памяти
14             char[] message;
15             //Размер введенного сообщения
16             int size;
17
18             //Получение существующего участка разделяемой памяти
19             //Параметр - название участка
20             MemoryMappedFile sharedMemory = MemoryMappedFile.OpenExisting("MemoryFile");
21             //Сначала считываем размер сообщения, чтобы создать массив данного размера
22             //Integer занимает 4 байта, начинается с первого байта, поэтому передаем цифры 0 и 4
23             using (MemoryMappedViewAccessor reader = sharedMemory.CreateViewAccessor(0, 4, MemoryMappedFileAccess.Read))
24             {
25                 size = reader.ReadInt32(0);
26             }
27
28             //Считываем сообщение, используя полученный выше размер
29             //Сообщение - это строка или массив объектов char, каждый из которых занимает два байта
30             //Поэтому вторым параметром передаем число символов умножив на из размер в байтах плюс
31             //А первый параметр - смещение - 4 байта, которое занимает размер сообщения
32             using (MemoryMappedViewAccessor reader = sharedMemory.CreateViewAccessor(4, size * 2, MemoryMappedFileAccess.Read))
33             {
34                 //Массив символов сообщения
35                 message = new char[size];
36                 reader.ReadArray<char>(0, message, 0, size);
37             }
38             Console.WriteLine("Получено сообщение :");
39             Console.WriteLine(message);
40             Console.WriteLine("Для выхода из программы нажмите любую клавишу");
41             Console.ReadLine();
42         }
43     }
44 }
45
```

Справочный материал

В разделе Чтение текстового файла этой статьи описывается, как использовать класс `StreamReader` для чтения текстового файла. В разделах Запись в текстовый файл и Запись в текстовый файл описывается, как использовать класс `StreamWriter` для записи текста в файл.

В следующем коде используется класс `StreamReader` для открытия, чтения и закрытия текстового файла. Можно передать путь к текстовому файлу в конструктор `StreamReader` для автоматического открытия файла. Метод `ReadLine` считывает каждую строку текста и перемещает указатель файла на следующую строку по мере чтения. Если метод `ReadLine` достигает конца файла, он возвращает пустую ссылку. Дополнительные сведения см. в разделе Класс `StreamReader`.

Задание 4

1. Создайте пример текстового файла в Блокноте. Выполните приведенные ниже действия.

- Вставьте текст `hello world` в Блокнот.
- Сохраните файл как `Sample.txt`.

2. Запустите Microsoft Visual Studio.

3. В меню Файл выберите пункт Создать, а затем выберите Проект.

4. Выберите Проекты Visual C# в разделе Типы проектов, а затем выберите

Консольное приложение в разделе Шаблоны.

5. Добавьте следующий код в начало файла Class1.cs:

using System.IO;

6. Добавьте указанный ниже код в метод Main:

```
String line;
try
{
    //Pass the file path and file name to the StreamReader constructor
    StreamReader sr = new StreamReader("C:\\Sample.txt");
    //Read the first line of text
    line = sr.ReadLine();
    //Continue to read until you reach end of file
    while (line != null)
    {
        //write the line to console window
        Console.WriteLine(line);
        //Read the next line
        line = sr.ReadLine();
    }
    //close the file
    sr.Close();
    Console.ReadLine();
}
catch(Exception e)
{
    Console.WriteLine("Exception: " + e.Message);
}
finally
{
    Console.WriteLine("Executing finally block.");
}
```

7. В меню Отладка выберите Пуск для компиляции и запуска приложения. Нажмите клавишу ВВОД, чтобы закрыть окно консоли. В окне консоли отображается содержимое файла Sample.txt:

Вывод консоли: Hello world.

```

1  //Read a Text File
2  using System;
3  using System.IO;
4  namespace readwriteapp
5  {
6      class Class1
7      {
8          [STAThread]
9          static void Main(string[] args)
10         {
11             String line;
12             try
13             {
14                 //Pass the file path and file name to the StreamReader constructor
15                 StreamReader sr = new StreamReader("C:\\Sample.txt");
16                 //Read the first line of text
17                 line = sr.ReadLine();
18                 //Continue to read until you reach end of file
19                 while (line != null)
20                 {
21                     //write the lie to console window
22                     Console.WriteLine(line);
23                     //Read the next line
24                     line = sr.ReadLine();
25                 }
26                 //close the file
27                 sr.Close();
28                 Console.ReadLine();
29             }
30             catch(Exception e)
31             {
32                 Console.WriteLine("Exception: " + e.Message);
33             }
34             finally
35             {
36                 Console.WriteLine("Executing finally block.");
37             }
38         }
39     }
40 }

```

Задание 5

В следующем коде используется класс StreamWriter для открытия, записи и закрытия текстового файла. Аналогично тому, как используется класс StreamReader, можно передать путь к текстовому файлу в конструктор StreamWriter для автоматического открытия файла. Метод WriteLine записывает всю текстовую строку в текстовый файл.

1. Запустите Visual Studio.
2. В меню Файл выберите пункт Создать, а затем выберите Проект.
3. Выберите Проекты Visual C# в разделе Типы проектов, а затем выберите Консольное приложение в разделе Шаблоны.
4. Добавьте следующий код в начало файла Class1.cs:
using System.IO;
5. Добавьте указанный ниже код в метод Main:

```

try
{
    //Pass the filepath and filename to the StreamWriter Constructor
    StreamWriter sw = new StreamWriter("C:\\Test.txt");
    //Write a line of text
    sw.WriteLine("Hello World!!");
    //Write a second line of text
    sw.WriteLine("From the StreamWriter class");
    //Close the file
    sw.Close();
}
catch(Exception e)
{
    Console.WriteLine("Exception: " + e.Message);
}
finally
{
    Console.WriteLine("Executing finally block.");
}
}

```

6. В меню Отладка выберите Пуск для компиляции и запуска приложения. Этот код создает файл с именем Test.txt на диске C. Откройте Test.txt в текстовом редакторе, например в Блокноте. Test.txt содержит две текстовые строки:

Hello World!!
From the StreamWriter class

```

1  //Write a text file - Version-1
2  using System;
3  using System.IO;
4  namespace readwriteapp
5  {
6      class Class1
7      {
8          [STAThread]
9          static void Main(string[] args)
10         {
11             try
12             {
13                 //Pass the filepath and filename to the StreamWriter Constructor
14                 StreamWriter sw = new StreamWriter("C:\\Test.txt");
15                 //Write a line of text
16                 sw.WriteLine("Hello World!!");
17                 //Write a second line of text
18                 sw.WriteLine("From the StreamWriter class");
19                 //Close the file
20                 sw.Close();
21             }
22             catch(Exception e)
23             {
24                 Console.WriteLine("Exception: " + e.Message);
25             }
26             finally
27             {
28                 Console.WriteLine("Executing finally block.");
29             }
30         }
31     }
32 }

```


Задание 6

В следующем коде используется класс `StreamWriter` для открытия, записи и закрытия текстового файла. В отличие от предыдущего примера, этот код передает в конструктор два дополнительных параметра. Первый параметр — путь к файлу и имя файла. Второй параметр, `true`, указывает, что файл открыт в режиме добавления. Если вы зададите `false` для второго параметра, содержимое файла перезаписывается при каждом запуске кода. Третий параметр задает `Unicode`, чтобы кодирование файла в `StreamWriter` выполнялось в формате Юникода. Можно также указать следующие методы кодирования для третьего параметра:

- ASCII
- Юникод
- UTF7
- UTF8

Метод `Write` аналогичен методу `WriteLine`, за исключением того, что метод `Write` не вставляет автоматически сочетание символов возврата каретки или перевода строки (CR/LF). Это полезно, когда нужно одновременно записывать по одному символу.

1. Запустите Visual Studio.
2. В меню Файл выберите пункт Создать и затем пункт Проект.
3. Нажмите Проекты Visual C# в разделе Типы проектов, а затем нажмите Консольное приложение в разделе Шаблоны.

4. Добавьте следующий код в начало файла `Class1.cs`:

```
using System.IO;  
using System.Text;
```

5. Добавьте указанный ниже код в метод `Main`:

```
Int64 x;  
try  
{  
    //Open the File  
    StreamWriter sw = new StreamWriter("C:\\Test1.txt", true, Encoding.ASCII);  
  
    //Write out the numbers 1 to 10 on the same line.  
    for(x=0; x < 10; x++)  
    {  
        sw.Write(x);  
    }  
  
    //close the file  
    sw.Close();  
}  
catch(Exception e)  
{  
    Console.WriteLine("Exception: " + e.Message);  
}  
finally  
{  
    Console.WriteLine("Executing finally block.");  
}
```

6. В меню Отладка выберите Пуск для компиляции и запуска приложения. Этот код создает файл с именем `Test1.txt` на диске C. Откройте `Test1.txt` в текстовом редакторе, например в Блокноте. `Test1.txt` содержит одну текстовую строку: 0123456789.

```

1  //Write a text file - Version 2
2  using System;
3  using System.IO;
4  using System.Text;
5  namespace readwriteapp
6  {
7      class Class1
8      {
9          [STAThread]
10         static void Main(string[] args)
11         {
12             Int64 x;
13             try
14             {
15                 //Open the File
16                 StreamWriter sw = new StreamWriter("C:\\Test1.txt", true, Encoding.ASCII);
17                 //Writeout the numbers 1 to 10 on the same line.
18                 for(x=0; x < 10; x++)
19                 {
20                     sw.Write(x);
21                 }
22                 //close the file
23                 sw.Close();
24             }
25             catch(Exception e)
26             {
27                 Console.WriteLine("Exception: " + e.Message);
28             }
29             finally
30             {
31                 Console.WriteLine("Executing finally block.");
32             }
33         }
34     }
35 }

```

Устранение неполадок

Перенос кода в блок **try-catch-finally** для обработки ошибок и исключений является хорошей практикой программирования, когда речь заходит о выполнении любых операций с файлом. В частности, может потребоваться освободить дескрипторы файла в окончательном блоке, чтобы файл не был заблокирован на неопределенный срок. Некоторые возможные ошибки включают файл, который не существует, или файл, который уже используется.

Задание 7

Создать новое консольное приложение в котором нужно отобразить содержимое текстового файла. Текстовый файл помещается в папку нового проекта с названием **Text.txt**.


```

1  using System;
2  using System.IO;
3
4  class Example
5  {
6      static void Main()
7      {
8          FileStream fin;
9          string s;
10         try
11         {
12             fin = new FileStream("C:/Temp/test.txt", FileMode.Open);
13         }
14         catch (IOException exc)
15         {
16             Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
17             return;
18         }
19         StreamReader fstr_in = new StreamReader(fin);
20         try
21         {
22             while ((s = fstr_in.ReadLine()) != null)
23             {
24                 Console.WriteLine(s);
25             }
26         }
27         catch (IOException exc)
28         {
29             Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
30         }
31
32         finally
33         {
34             fstr_in.Close();
35         }
36         Console.WriteLine("Нажмите любую кнопку!");
37         Console.ReadKey();
38     }
39 }
40
41 }

```

Пример полученного вывода:

```

(Lennon/McCartney)

Oh! Darling, please believe me
I'll never do you no harm
Believe me when I tell you
I'll never do you no harm

Oh! Darling, if you leave me
I'll never make it alone
Believe me when I beg you
Don't ever leave me alone

When you told me you didn't need me anymore
Well you know I nearly broke down and cried
When you told me you didn't need me anymore
Well you know I nearly broke down and died

Oh! Darling, if you leave me
I'll never make it alone
Believe me when I tell you
I'll never do you no harm

When you told me you didn't need me anymore
Well you know I nearly broke down and cried
When you told me you didn't need me anymore
Well you know I nearly broke down and died

Oh! Darling, please believe me
I'll never let you down
Believe me when I tell you
I'll never do you no harm
Нажмите любую кнопку!

```

Задание 8

Создать новое консольное приложение: Сортировка целочисленного массива. Бинарный поиск элементов. Полученные значения, вывод консоли записать в текстовый файл.

```
1  using System;
2
3  class arraySort
4  {
5
6      static void print(int[] alpha)
7      {
8          foreach (int i in alpha)
9              Console.WriteLine(i + " ");
10         Console.WriteLine();
11     }
12
13     static void Main()
14     {
15
16         int[] alpha = { -4, 8, 28, -6, -98, -47, 12 };
17         //Отобразить исходный массив
18
19         Console.WriteLine("Исходный массив ...");
20         print(alpha);
21
22         //Найти значение 8
23         int pattern = 8;
24         int indx = Array.BinarySearch(alpha, pattern);
25         Console.WriteLine("Индекс элемента массива со значением 8: " + indx);
26
27         //Сортируем массив
28         Array.Sort(alpha);
29         Console.WriteLine("Сортировка по возрастанию ...");
30         //Отображаем массив после сортировки
31         print(alpha);
32
33         //Сортируем массив по убыванию
34         Array.Reverse(alpha);
35         //Отображаем массив после сортировки по убыванию
36         Console.WriteLine("Сортировка по убыванию ...");
37         print(alpha);
38         Console.WriteLine("Нажмите любую кнопку!");
39         Console.ReadKey();
40     }
41 }
```

Пример полученного вывода:

```
Исходный массив ...
-4
8
28
-6
-98
-47
12

Индекс элемента массива со значением 8: -7
Сортировка по возрастанию ...
-98
-47
-6
-4
8
12
28

Сортировка по убыванию ...
28
12
8
-4
-6
-47
-98

Нажмите любую кнопку!
```

Содержание работы

1. Откройте новый документ Word, выполните настройку документа и заполните необходимую информацию согласно методических указаний. Сохраните документ в папке «Мои документы» с именем «Фамилия, группа, Пр.р.№», не забывайте периодически сохранять документ в процессе выполнения работы.
2. Внимательно изучите краткие теоретические сведения.
3. Оформите предложенный текст в соответствии с требованиями к проекту.
4. Сделайте выводы, подготовьтесь к защите.

Контрольные вопросы

- 1) Обмен данными?
- 2) Типы обмена данными?
- 3) Различия обмена данных?
- 4) Обмен данными между процессами?
- 5) Использование буфера обмена для IPC?
- 6) Использование COM для IPC?
- 7) Использование копирования данных для IPC?
- 8) Использование DDE для IPC?

ПРАКТИЧЕСКАЯ РАБОТА №17-24

Тема: Сетевое программирование сокетов

Цель: Изучить сетевое программирование сокетов.

Оборудование: В соответствии с рабочей программой ПМ.01 «Разработка модулей программного обеспечения для компьютерных систем»:

- Автоматизированные рабочие места на 12-15 обучающихся (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Автоматизированное рабочее место преподавателя (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Проектор и экран;
- Маркерная доска;
- Лицензионное программное обеспечение общего и профессионального назначения.

Справочный материал

Сокет — это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами (локальными или удаленными). Реализация сокетов обеспечивает инкапсуляцию протоколов сетевого и транспортного уровней.

Первоначально сокеты были разработаны для UNIX в Калифорнийском университете в Беркли. В UNIX обеспечивающий связь метод ввода-вывода следует алгоритму open/read/write/close. Прежде чем ресурс использовать, его нужно открыть, задав соответствующие разрешения и другие параметры. Как только ресурс открыт, из него можно считывать или в него записывать данные. После использования ресурса пользователь должен вызывать метод Close(), чтобы подать сигнал операционной системе о завершении его работы с этим ресурсом.

Когда в операционную систему UNIX были добавлены средства *межпроцессного взаимодействия (Inter-Process Communication, IPC)* и сетевого обмена, был заимствован привычный шаблон ввода-вывода. Все ресурсы, открытые для связи, в UNIX и Windows идентифицируются дескрипторами. Эти дескрипторы, или *описатели (handles)*, могут указывать на файл, память или какой-либо другой канал связи, а фактически указывают на внутреннюю структуру данных, используемую операционной системой. Сокет, будучи таким же ресурсом, тоже представляется дескриптором. Следовательно, для сокетов жизнь дескриптора можно разделить на три фазы: открыть (создать) сокет, получить из сокета или отправить сокету и в конце концов закрыть сокет.

Интерфейс IPC для взаимодействия между разными процессами построен поверх методов ввода-вывода. Они облегчают для сокетов отправку и получение данных. Каждый целевой объект задается адресом сокета, следовательно, этот адрес можно указать в клиенте, чтобы установить соединение с целью.

Типы сокетов

Существуют два основных типа сокетов — потоковые сокеты и дейтаграммные.

Потоковые сокеты (stream socket)

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т. е. через эту конечную точку приложение может и передавать, и получать данные.

Потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, сдублированных данных. Потоковый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Потоковые сокеты достигают этого уровня качества за счет использования протокола ***Transmission Control Protocol (TCP)***. TCP обеспечивает поступление данных на другую сторону в нужной последовательности и без ошибок.

Для этого типа сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают. Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности.

Однако, отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании TCP этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на уровень TCP определенное число байтов и другая сторона получает эти байты. В свою очередь TCP эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе.

Потоки базируются на явных соединениях: сокет А запрашивает соединение с сокетом В, а сокет В либо соглашается с запросом на установление соединения, либо отвергает его.

Если данные должны гарантированно доставляться другой стороне или размер их велик, потоковые сокеты предпочтительнее дейтаграммных. Следовательно, если надежность связи между двумя приложениями имеет первостепенное значение, выбирайте потоковые сокеты.

Сервер электронной почты представляет пример приложения, которое должно доставлять содержание в правильном порядке, без дублирования и пропусков. Потоковый сокет рассчитывает, что TCP обеспечит доставку сообщений по их назначениям.

Дейтаграммные сокеты (datagram socket)

Дейтаграммные сокеты иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается — сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Потоковые сокеты по сравнению с дейтаграммными действительно дают более надежный метод, но для некоторых приложений накладные расходы, связанные с установкой явного соединения, неприемлемы (например, сервер времени суток, обеспечивающий синхронизацию времени для своих клиентов). В конце концов на установление надежного соединения с сервером требуется время, которое просто

вносит задержки в обслуживание, и задача серверного приложения не выполняется. Для сокращения накладных расходов нужно использовать дейтаграммные сокеты.

Использование дейтаграммных сокетов требует, чтобы передачей данных от клиента к серверу занимался *User Datagram Protocol (UDP)*. В этом протоколе на размер сообщений налагаются некоторые ограничения, и в отличие от потоковых сокетов, умеющих надежно отправлять сообщения серверу-адресату, дейтаграммные сокеты надежность не обеспечивают. Если данные затерялись где-то в сети, сервер не сообщит об ошибках.

Кроме двух рассмотренных типов существует также обобщенная форма сокетов, которую называют необрабатываемыми или сырыми.

Сырые сокеты (raw socket)

Главная цель использования сырых сокетов состоит в обходе механизма, с помощью которого компьютер обрабатывает TCP/IP. Это достигается обеспечением специальной реализации стека TCP/IP, замещающей механизм, предоставленный стеком TCP/IP в ядре — пакет непосредственно передается приложению и, следовательно, обрабатывается гораздо эффективнее, чем при проходе через главный стек протоколов клиента.

По определению, сырой сокет — это сокет, который принимает пакеты, обходит уровни TCP и UDP в стеке TCP/IP и отправляет их непосредственно приложению.

При использовании таких сокетов пакет не проходит через фильтр TCP/IP, т.е. никак не обрабатывается, и предстает в своей сырой форме. В таком случае обязанность правильно обработать все данные и выполнить такие действия, как удаление заголовков и разбор полей, ложится на получающее приложение — все равно, что включить в приложение небольшой стек TCP/IP.

Однако нечасто может потребоваться программа, работающая с сырыми сокетами. Если вы не пишете системное программное обеспечение или программу, аналогичную анализатору пакетов, вникать в такие детали не придется. Сырые сокеты главным образом используются при разработке специализированных низкоуровневых протокольных приложений. Например, такие разнообразные утилиты TCP/IP, как *trace route*, *ping* или *arp*, используют сырые сокеты.

Работа с сырыми сокетами требует солидного знания базовых протоколов TCP/UDP/IP.

Порты

Порт определен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. По существу с его помощью расширяется понятие IP-адреса. Компьютер, на котором в одно время выполняется несколько приложений, получая пакет из сети, может идентифицировать целевой процесс, пользуясь уникальным номером порта, определенным при установлении соединения.

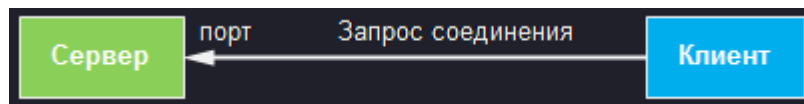
Сокет состоит из IP-адреса машины и номера порта, используемого приложением TCP. Поскольку IP-адрес уникален в Интернете, а номера портов уникальны на отдельной машине, номера сокетов также уникальны во всем Интернете. Эта характеристика позволяет процессу общаться через сеть с другим процессом исключительно на основании номера сокета.

За определенными службами номера портов зарезервированы — это широко известные номера портов, например порт 21, использующийся в FTP. Ваше

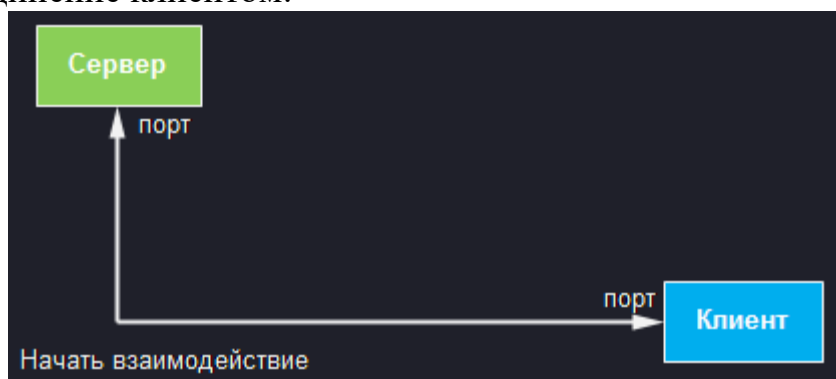
приложение может пользоваться любым номером порта, который не был зарезервирован и пока не занят. Агентство Internet Assigned Numbers Authority (IANA) ведет перечень широко известных номеров портов.

Обычно приложение клиент-сервер, использующее сокет, состоит из двух разных приложений - клиента, иницилирующего соединение с целью (сервером), и сервера, ожидающего соединения от клиента.

Например, на стороне клиента, приложение должно знать адрес цели и номер порта. Отправляя запрос на соединение, клиент пытается установить соединение с сервером:



Если события развиваются удачно, при условии что сервер запущен прежде, чем клиент попытался с ним соединиться, сервер соглашается на соединение. Дав согласие, серверное приложение создает новый сокет для взаимодействия именно с установившим соединение клиентом:



Теперь клиент и сервер могут взаимодействовать между собой, считывая сообщения каждый из своего сокета и, соответственно, записывая сообщения.

Работа с сокетами в .NET

Поддержку сокетов в .NET обеспечивают классы в пространстве имен System.Net.Sockets — начнем с их краткого описания.

Таблица 1- Классы для работы с сокетами

Класс	Описание
<i>MulticastOption</i>	Класс MulticastOption устанавливает значение IP-адреса для присоединения к IP-группе или для выхода из нее.
<i>NetworkStream</i>	Класс NetworkStream реализует базовый класс потока, из которого данные отправляются и в котором они получаются. Это абстракция высокого уровня, представляющая соединение с каналом связи TCP/IP.
<i>TcpClient</i>	Класс TcpClient строится на классе Socket, чтобы обеспечить TCP-обслуживание на более высоком уровне. TcpClient предоставляет несколько методов для отправки и получения данных через сеть.
<i>TcpListener</i>	Этот класс также построен на низкоуровневом классе Socket. Его основное назначение — серверные приложения. Он ожидает входящие запросы на соединения от клиентов и уведомляет приложение о любых соединениях.
<i>UdpClient</i>	UDP — это протокол, не организующий соединение, следовательно, для реализации UDP-обслуживания в .NET

	требуется другая функциональность.
<i>SocketException</i>	Это исключение порождается, когда в сокете возникает ошибка.
<i>Socket</i>	Последний класс в пространстве имен System.Net.Sockets — это сам класс Socket. Он обеспечивает базовую функциональность приложения сокета.

Класс Socket

Класс Socket играет важную роль в сетевом программировании, обеспечивая функционирование как клиента, так и сервера. Главным образом, вызовы методов этого класса выполняют необходимые проверки, связанные с безопасностью, в том числе проверяют разрешения системы безопасности, после чего они переправляются к аналогам этих методов в Windows Sockets API.

Прежде чем обращаться к примеру использования класса Socket, рассмотрим некоторые важные свойства и методы этого класса:

Свойства и методы класса Socket

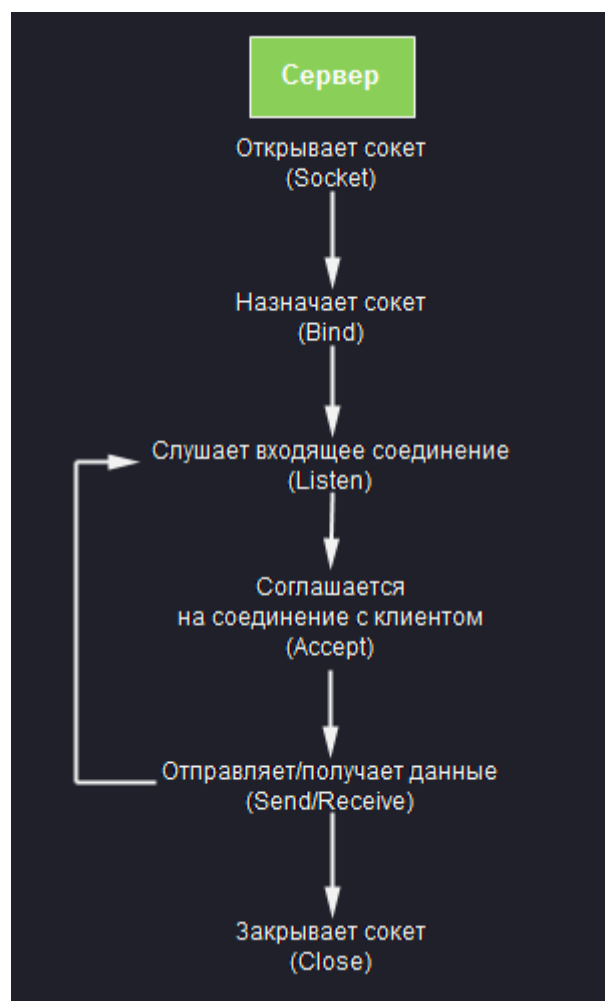
Свойство или метод	Описание
<i>AddressFamily</i>	Дает семейство адресов сокета — значение из перечисления Socket.AddressFamily.
<i>Available</i>	Возвращает объем доступных для чтения данных.
<i>Blocking</i>	Дает или устанавливает значение, показывающее, находится ли сокет в блокирующем режиме.
<i>Connected</i>	Возвращает значение, информирующее, соединен ли сокет с удаленным хостом.
<i>LocalEndPoint</i>	Дает локальную конечную точку.
<i>ProtocolType</i>	Дает тип протокола сокета.
<i>RemoteEndPoint</i>	Дает удаленную конечную точку сокета.
<i>SocketType</i>	Дает тип сокета.
<i>Accept()</i>	Создает новый сокет для обработки входящего запроса на соединение.
<i>Bind()</i>	Связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение.
<i>Close()</i>	Заставляет сокет закрыться.
<i>Connect()</i>	Устанавливает соединение с удаленным хостом.
<i>GetSocketOption()</i>	Возвращает значение SocketOption.
<i>IOControl()</i>	Устанавливает для сокета низкоуровневые режимы работы. Этот метод обеспечивает низкоуровневый доступ к лежащему в основе классу Socket.
<i>Listen()</i>	Помещает сокет в режим прослушивания (ожидания). Этот метод предназначен только для серверных приложений.
<i>Receive()</i>	Получает данные от соединенного сокета.
<i>Poll()</i>	Определяет статус сокета.
<i>Select()</i>	Проверяет статус одного или нескольких сокетов.
<i>Send()</i>	Отправляет данные соединенному сокету.
<i>SetSocketOption()</i>	Устанавливает опцию сокета.
<i>Shutdown()</i>	Запрещает операции отправки и получения данных на сокете.

В следующем примере используем TCP, чтобы обеспечить упорядоченные, надежные двусторонние потоки байтов. Построим завершенное приложение, включающее клиент и сервер. Сначала продемонстрируем, как сконструировать на потоковых сокетах TCP сервер, а затем клиентское приложение для тестирования нашего сервера.

Следующая программа создает сервер, получающий запросы на соединение от клиентов. Сервер построен синхронно, следовательно, выполнение потока блокируется, пока сервер не даст согласия на соединение с клиентом. Это приложение демонстрирует простой сервер, отвечающий клиенту. Клиент завершает соединение, отправляя серверу сообщение <TheEnd>.

Задание 1 Сервер TCP

Создание структуры сервера показано на следующей функциональной диаграмме:



Вот полный код программы SocketServer.cs:

```

1  // SocketServer.cs
2  using System;
3  using System.Text;
4  using System.Net;
5  using System.Net.Sockets;
6
7  namespace SocketServer
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             // Устанавливаем для сокета локальную конечную точку
14             IPHostEntry ipHost = Dns.GetHostEntry("localhost");
15             IPAddress ipAddr = ipHost.AddressList[0];
16             IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
17
18             // Создаем сокет Tcp/Ip
19             Socket sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
20
21             // Назначаем сокет локальной конечной точке и слушаем входящие сокеты
22             try
23             {
24                 sListener.Bind(ipEndPoint);
25                 sListener.Listen(10);
26
27                 // Начинаем слушать соединения
28                 while (true)
29                 {
30                     Console.WriteLine("Ожидаем соединение через порт {0}", ipEndPoint);
31
32                     // Программа приостанавливается, ожидая входящее соединение
33                     Socket handler = sListener.Accept();
34                     string data = null;
35
36                     // Мы дождались клиента, пытающегося с нами соединиться
37
38                     byte[] bytes = new byte[1024];
39                     int bytesRec = handler.Receive(bytes);
40
41                     data += Encoding.UTF8.GetString(bytes, 0, bytesRec);
42
43                     // Показываем данные на консоли
44                     Console.WriteLine("Полученный текст: " + data + "\n\n");
45
46                     // Отправляем ответ клиенту\
47                     string reply = "Спасибо за запрос в " + data.Length.ToString()
48                                     + " символов";
49                     byte[] msg = Encoding.UTF8.GetBytes(reply);
50                     handler.Send(msg);
51
52                     if (data.IndexOf("<TheEnd>") > -1)
53                     {
54                         Console.WriteLine("Сервер завершил соединение с клиентом.");
55                         break;
56                     }
57
58                     handler.Shutdown(SocketShutdown.Both);
59                     handler.Close();
60                 }
61             }
62             catch (Exception ex)
63             {
64                 Console.WriteLine(ex.ToString());
65             }
66             finally
67             {
68                 Console.ReadLine();
69             }
70         }
71     }
72 }

```

Справочный материал

Первый шаг заключается в установлении для сокета локальной конечной точки. Прежде чем открывать сокет для ожидания соединений, нужно подготовить для него адрес локальной конечной точки. Уникальный адрес для обслуживания TCP/IP определяется комбинацией IP-адреса хоста с номером порта обслуживания, которая создает конечную точку для обслуживания.

Класс **Dns** предоставляет методы, возвращающие информацию о сетевых адресах, поддерживаемых устройством в локальной сети. Если у устройства локальной сети имеется более одного сетевого адреса, класс **Dns** возвращает информацию обо всех сетевых адресах, и приложение должно выбрать из массива подходящий адрес для обслуживания.

Создадим **IPEndPoint** для сервера, комбинируя первый IP-адрес хост-компьютера, полученный от метода **Dns.Resolve()**, с номером порта:

```
IPHostEntry ipHost = Dns.GetHostEntry("localhost");  
IPAddress ipAddr = ipHost.AddressList[0];  
IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
```

Здесь класс **IPEndPoint** представляет **localhost** на порте **11000**. Далее новым экземпляром класса **Socket** создаем потоковый сокет. Установив локальную конечную точку для ожидания соединений, можно создать сокет:

```
Socket sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream,  
ProtocolType.Tcp);
```

Перечисление **AddressFamily** указывает схемы адресации, которые экземпляр класса **Socket** может использовать для разрешения адреса.

В параметре **SocketType** различаются сокеты **TCP** и **UDP**. В нем можно определить в том числе следующие значения:

Dgram

Поддерживает дейтаграммы. Значение **Dgram** требует указать **Udp** для типа протокола и **InterNetwork** в параметре семейства адресов.

Raw

Поддерживает доступ к базовому транспортному протоколу.

Stream

Поддерживает потоковые сокеты. Значение **Stream** требует указать **Tcp** для типа протокола.

Третий и последний параметр определяет тип протокола, требуемый для сокета. В параметре **ProtocolType** можно указать следующие наиболее важные значения - **Tcp**, **Udp**, **Ip**, **Raw**.

Следующим шагом должно быть назначение сокета с помощью метода **Bind()**. Когда сокет открывается конструктором, ему не назначается имя, а только резервируется дескриптор. Для назначения имени сокету сервера вызывается метод **Bind()**. Чтобы сокет клиента мог идентифицировать потоковый сокет **TCP**, серверная программа должна дать имя своему сокету:

sListener.Bind(ipEndPoint);

Метод **Bind()** связывает сокет с локальной конечной точкой. Вызывать метод **Bind()** надо до любых попыток обращения к методам **Listen()** и **Accept()**.

Теперь, создав сокет и связав с ним имя, можно слушать входящие сообщения, воспользовавшись методом **Listen()**. В состоянии прослушивания сокет будет ожидать входящие попытки соединения:

sListener.Listen(10);

В параметре определяется задел (backlog), указывающий максимальное число соединений, ожидающих обработки в очереди. В приведенном коде значение параметра допускает накопление в очереди до десяти соединений.

В состоянии прослушивания надо быть готовым дать согласие на соединение с клиентом, для чего используется метод **Accept()**. С помощью этого метода получается соединение клиента и завершается установление связи имен клиента и сервера. Метод **Accept()** блокирует поток вызывающей программы до поступления соединения.

Метод **Accept()** извлекает из очереди ожидающих запросов первый запрос на соединение и создает для его обработки новый сокет. Хотя новый сокет создан, первоначальный сокет продолжает слушать и может использоваться с многопоточной обработкой для приема нескольких запросов на соединение от клиентов. Никакое серверное приложение не должно закрывать слушающий сокет. Он должен продолжать работать наряду с сокетами, созданными методом **Accept** для обработки входящих запросов клиентов.

while (true)

```
{  
    Console.WriteLine("Ожидаем соединение через порт {0}", ipEndPoint);  
  
    // Программа приостанавливается, ожидая входящее соединение  
    Socket handler = sListener.Accept();
```

Как только клиент и сервер установили между собой соединение, можно отправлять и получать сообщения, используя методы **Send()** и **Receive()** класса **Socket**.

Метод **Send()** записывает исходящие данные сокету, с которым установлено соединение. Метод **Receive()** считывает входящие данные в потоковый сокет. При использовании системы, основанной на TCP, перед выполнением методов **Send()** и **Receive ()** между сокетами должно быть установлено соединение. Точный протокол между двумя взаимодействующими сущностями должен быть определен заблаговременно, чтобы клиентское и серверное приложения не блокировали друг друга, не зная, кто должен отправить свои данные первым.

Когда обмен данными между сервером и клиентом завершается, нужно закрыть соединение используя методы **Shutdown()** и **Close()**:

```
handler.Shutdown(SocketShutdown.Both);  
handler.Close();
```

SocketShutdown — это перечисление, содержащее три значения для остановки: **Both** - останавливает отправку и получение данных сокетом, **Receive** - останавливает получение данных сокетом и **Send** - останавливает отправку данных сокетом.

Сокет закрывается при вызове метода **Close()**, который также устанавливает в свойстве **Connected** сокета значение **false**.

Задание 2

Клиент на TCP

Функции, которые используются для создания приложения-клиента, более или менее напоминают серверное приложение. Как и для сервера, используются те же методы для определения конечной точки, создания экземпляра сокета, отправки и получения данных и закрытия сокета:

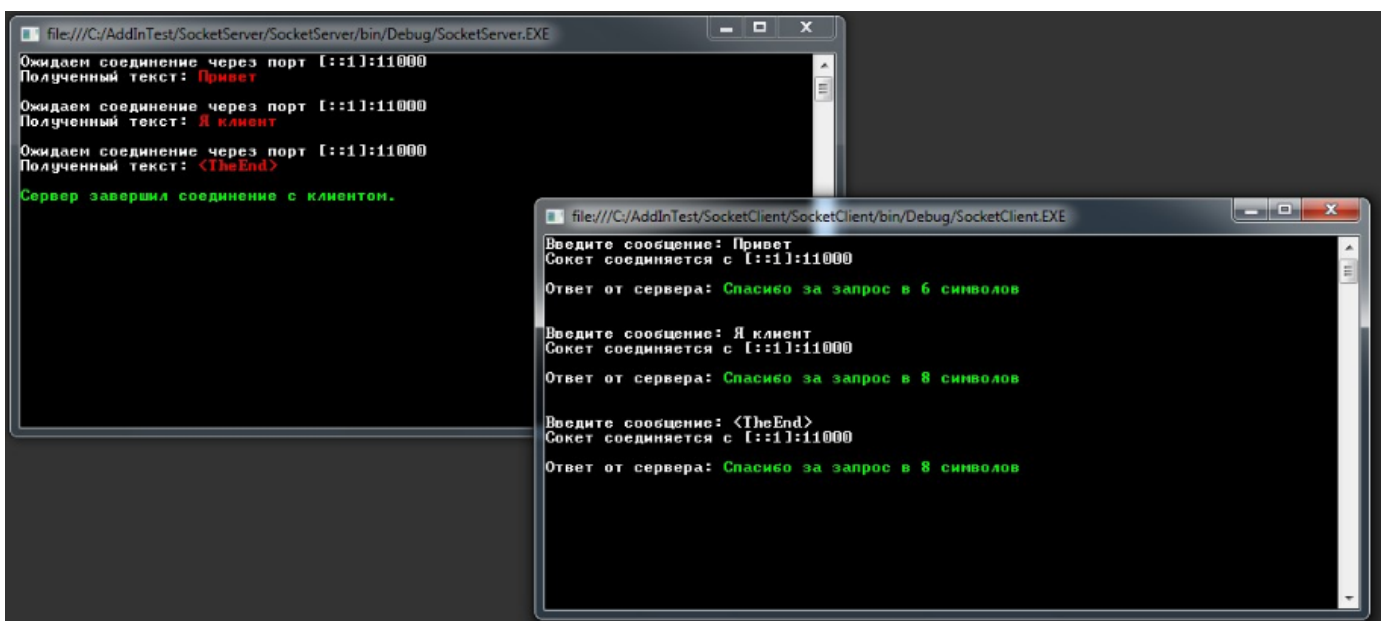
```
1  // SocketClient.cs  
2  using System;  
3  using System.Text;  
4  using System.Net;  
5  using System.Net.Sockets;  
6  
7  namespace SocketClient  
8  {  
9      class Program  
10     {  
11         static void Main(string[] args)  
12         {  
13             try  
14             {  
15                 SendMessageFromSocket(11000);  
16             }  
17             catch (Exception ex)  
18             {  
19                 Console.WriteLine(ex.ToString());  
20             }  
21             finally  
22             {  
23                 Console.ReadLine();  
24             }  
25         }  
26  
27         static void SendMessageFromSocket(int port)  
28         {  
29             // Буфер для входящих данных  
30             byte[] bytes = new byte[1024];  
31  
32             // Соединяемся с удаленным устройством  
33  
34             // Устанавливаем удаленную точку для сокета  
35             IPHostEntry ipHost = Dns.GetHostEntry("localhost");  
36             IPAddress ipAddr = ipHost.AddressList[0];  
37             IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);  
38  
39             Socket sender = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);  
40  
41             // Соединяем сокет с удаленной точкой  
42             sender.Connect(ipEndPoint);
```

```

43
44     Console.WriteLine("Введите сообщение: ");
45     string message = Console.ReadLine();
46
47     Console.WriteLine("Сокет соединяется с {0} ", sender.RemoteEndPoint.ToString());
48     byte[] msg = Encoding.UTF8.GetBytes(message);
49
50     // Отправляем данные через сокет
51     int bytesSent = sender.Send(msg);
52
53     // Получаем ответ от сервера
54     int bytesRec = sender.Receive(bytes);
55
56     Console.WriteLine("\nОтвет от сервера: {0}\n\n", Encoding.UTF8.GetString(bytes, 0, bytesRec));
57
58     // Используем рекурсию для неоднократного вызова SendMessageFromSocket()
59     if (message.IndexOf("<TheEnd>") == -1)
60         SendMessageFromSocket(port);
61
62     // Освобождаем сокет
63     sender.Shutdown(SocketShutdown.Both);
64     sender.Close();
65 }
66
67 }

```

Единственный новый метод - метод **Connect()**, используется для соединения с удаленным сервером. На рисунке ниже показаны клиент и сервер в действии:



Задание 3

В предыдущем задании было рассмотрено самое простое приложение клиент-сервер, теперь с помощью исключения **SocketException** создадим нечто более интересное.

В следующем примере создадим собственную программу сканирования портов, которая пытается соединиться с **localhost** по каждому порту, указанному в цикле. Сообщаем об успешных соединениях, а если установить соединение не удастся, перехватываем порождаемое в этом случае исключение **SocketException**. В качестве графической среды используется **WPF**.

Сканер портов может использоваться для получения списка открытых портов на вашем компьютере. В открытых портах проявляется потенциальная слабость

системы, которой могут воспользоваться приложения-нарушители. Вот полный код программы, включая исходную XAML-разметку:

```
1 <Window x:Class="SocketPortScanner.MainWindow"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       Title="Сканер портов" WindowState="Maximized">
5     <ListView Name="listview_scanner" Margin="5">
6       <ListView.Resources>
7         <Style TargetType="{x:Type ListView}">
8           <Setter Property="ItemContainerStyle">
9             <Setter.Value>
10              <Style TargetType="ListViewItem">
11                <Setter Property="HorizontalContentAlignment" Value="Center"/>
12              </Style>
13            </Setter.Value>
14          </Setter>
15        </Style>
16      </ListView.Resources>
17      <ListView.View>
18        <GridView>
19          <GridView.Columns>
20            <GridViewColumn Header="Port ID" DisplayMemberBinding="{Binding Path=PortNumber}"
21                          Width="150"/>
22            <GridViewColumn Header="Local Address" DisplayMemberBinding="{Binding Path=Local}"
23                          Width="250"/>
24            <GridViewColumn Header="Remote Address" DisplayMemberBinding="{Binding Path=Remote}" Width="250"/>
25            <GridViewColumn Header="State" DisplayMemberBinding="{Binding Path=State}" Width="250"/>
26          </GridView.Columns>
27        </GridView>
28      </ListView.View>
29    </ListView>
30 </Window>
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Windows;
5 using System.Windows.Data;
6 using System.Net;
7 using System.Net.Sockets;
8 using System.Net.NetworkInformation;
9
10 namespace SocketPortScanner
11 {
12     public partial class MainWindow : Window
13     {
14         public MainWindow()
15         {
16             InitializeComponent();
17
18             List<PortInfo> pi = MainWindow.GetOpenPort();
19             listview_scanner.ItemsSource = pi;
20         }
21
22         private static List<PortInfo> GetOpenPort()
23         {
24             IPGlobalProperties properties = IPGlobalProperties.GetIPGlobalProperties();
25             IPEndPoint[] tcpEndPoints = properties.GetActiveTcpListeners();
26             TcpConnectionInformation[] tcpConnections = properties.GetActiveTcpConnections();
27
28             return tcpConnections.Select(p =>
29             {
30                 return new PortInfo(
31                     i: p.LocalEndPoint.Port,
32                     local: String.Format("{0}:{1}", p.LocalEndPoint.Address, p.LocalEndPoint.Port),
33                     remote: String.Format("{0}:{1}", p.RemoteEndPoint.Address, p.RemoteEndPoint.Port),
34                     state: p.State.ToString());
35             }).ToList();
36         }
37     }
38 }
```



```

38
39     class PortInfo
40     {
41         public int PortNumber { get; set; }
42         public string Local { get; set; }
43         public string Remote { get; set; }
44         public string State { get; set; }
45
46         public PortInfo(int i, string local, string remote, string state)
47         {
48             PortNumber = i;
49             Local = local;
50             Remote = remote;
51             State = state;
52         }
53     }
54 }

```

The screenshot shows a window titled "Сканер портов" (Port Scanner) with a table of network connections. The table has four columns: Port ID, Local Address, Remote Address, and State. The data is as follows:

Port ID	Local Address	Remote Address	State
16201	72.27.89.88:16201	80.89.133.162:80	Established
23964	72.27.89.88:23964	93.88.162.77:443	Established
23965	72.27.89.88:23965	93.88.162.48:443	Established
23981	72.27.89.88:23981	93.88.162.78:80	CloseWait
24053	72.27.89.88:24053	109.229.11.157:63073	SynSent
24056	72.27.89.88:24056	78.132.154.114:60874	SynSent
24057	72.27.89.88:24057	78.106.172.228:26290	SynSent
24059	72.27.89.88:24059	78.26.220.69:63803	TimeWait
24060	72.27.89.88:24060	78.26.144.147:50210	SynSent
24061	72.27.89.88:24061	77.223.93.24:52029	SynSent
24063	72.27.89.88:24063	77.35.187.9:35691	SynSent
24065	72.27.89.88:24065	91.146.60.162:48966	SynSent
24066	72.27.89.88:24066	77.35.249.72:62902	SynSent
24067	72.27.89.88:24067	128.0.129.69:11890	SynSent
24068	72.27.89.88:24068	77.35.25.143:62813	SynSent
24069	72.27.89.88:24069	62.182.65.28:10457	SynSent
24070	72.27.89.88:24070	62.32.68.19:63259	SynSent
24071	72.27.89.88:24071	88.215.137.104:32022	Established
24072	72.27.89.88:24072	128.70.107.239:35691	SynSent
24073	72.27.89.88:24073	128.72.122.33:61384	SynSent
24074	72.27.89.88:24074	134.255.154.42:35691	SynSent
24075	72.27.89.88:24075	141.101.235.9:42575	SynSent
24076	72.27.89.88:24076	46.229.143.101:43276	SynSent
62676	72.27.89.88:62676	23.60.69.151:80	CloseWait
65296	72.27.89.88:65296	75.126.203.247:80	CloseWait

Справочный материал

Среда .NET Framework, предоставляет многочисленные классы, помогающие разрабатывать в приложениях безопасный код. Многие из этих классов предлагают основанную на ролях безопасность и криптографию. Среда .NET Framework также дает объекты разрешений доступа к коду, являющиеся компоновочными блоками для защиты от несанкционированного доступа к коду. Они являются фундаментом, гарантирующим управляемому коду безопасность — может выполняться только тот код, который имеет разрешение выполняться в текущем контексте.

Каждое разрешение доступа к коду демонстрирует одно из следующих прав:

- Право доступа к защищенным ресурсам, например к файлам.
- Право выполнения защищенной операции, например обращения к управляемому коду.

Для мира Интернета и особенно для сетевых приложений классы **System.Net**

предоставляют встроенную поддержку аутентификации и разрешений доступа к коду. Среда **.NET Framework** дает класс **SocketPermission**, обеспечивающий соблюдение разрешений доступа к коду.

Класс **SocketPermission** используется для управления правами на установление и принятие соединений, контролирующими доступ к сети через сокет. Этот класс состоит из спецификации хоста и набора "действий", определяющих способы установления соединений с этим хостом. Он обеспечивает безопасность кода, контролируя значения имени хоста, IP-адреса и транспортного протокола.

Для сокетов C# имеется два способа обеспечить разрешение, поддерживающее безопасность:

- Императивно, используя класс **SocketPermission**
- Декларативно, используя класс **SocketPermissionAttribute**

Синтаксис императивной безопасности реализует разрешения, создавая новый экземпляр класса **SocketPermission**, чтобы при выполнении кода запросить конкретное разрешение, например право установить TCP-соединение. Этот способ обычно применяется, когда настройки безопасности изменяются при выполнении приложения. В декларативном синтаксисе используются атрибуты, позволяющие поместить информацию о безопасности в метаданные нашего кода, чтобы клиент, вызывающий код, мог воспользоваться рефлексией и узнать, какие разрешения требуются для кода.

Задание 4

Императивная безопасность

Этот синтаксис для обеспечения безопасности создает новый экземпляр класса **SocketPermission**. Синтаксис императивной безопасности можно использовать для выполнения требований и переопределений, но не запросов. Прежде чем вызвать соответствующий критерий безопасности, необходимо через конструктор инициализировать состояние класса **SocketPermission**, чтобы он представлял конкретную форму разрешения, которую вы ищете.

В следующем приложении демонстрируется основное использование класса **SocketPermission**. Поскольку этот код ведет себя как клиент, до выполнения этой программы нужно запустить приложение **SocketServer.cs**, созданное ранее:

```

1  using System;
2  using System.Text;
3  using System.Net;
4  using System.Net.Sockets;
5  using System.Security;
6  using System.Security.Permissions;
7
8  namespace SocketPermissionSample
9  {
10     class Program
11     {
12         static void Main(string[] args)
13         {
14             // Устанавливаем удаленную точку для сокета
15             IPHostEntry ipHost = Dns.GetHostEntry("localhost");
16             IPAddress ipAddr = ipHost.AddressList[0];
17             IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
18
19             Socket sender = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
20
21             // Настройка разрешений
22             SocketPermission permisSocket = new SocketPermission(
23                 NetworkAccess.Connect, TransportType.Tcp, "localhost",
24                 SocketPermission.AllPorts);
25
26             permisSocket.Assert();
27
28             try
29             {
30                 // Соединяем сокет с удаленной endPoint, перехватываем все ошибки
31                 sender.Connect(ipEndPoint);
32                 Console.WriteLine("Сокет подключен к {0}", sender.RemoteEndPoint.ToString());
33
34                 byte[] bytes = new byte[1024];
35                 byte[] msg = Encoding.UTF8.GetBytes("Простой тест");
36
37                 // Отправляем данные через сокет
38                 int bytesSend = sender.Send(msg);
39
40                 // Получаем ответ от удаленного устройства
41                 int bytesRec = sender.Receive(bytes);
42
43                 Console.WriteLine("Текст ответа: {0}",
44                     Encoding.UTF8.GetString(bytes, 0, bytesRec));
45             }
46             catch (Exception ex)
47             {
48                 Console.WriteLine(ex.ToString());
49             }
50             finally
51             {
52                 if (sender.Connected)
53                 {
54                     // Освобождаем сокет
55                     sender.Shutdown(SocketShutdown.Both);
56                     sender.Close();
57                 }
58             }
59             Console.ReadLine();
60         }
61     }
62 }

```

Приведенный код показывает, как, используя императивный синтаксис, реализовать безопасность доступа к коду. Важность этого кода в данном случае заключается в вызове одного метода класса **SocketPermission** - **Assert()**, который указывает, что приложению разрешается давать согласие на запросы соединения от Интернета и местного ресурса.

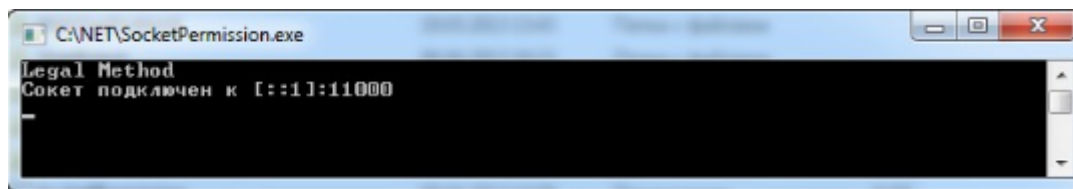
Задание 5

Декларативная безопасность

Декларативная безопасность использует атрибуты .NET, чтобы поместить информацию о безопасности внутрь метаданных кода. Атрибуты можно поместить на уровне сборки, класса или члена и указать необходимый тип запроса, требования или переопределения. Для использования этого синтаксиса безопасности сначала через декларативный синтаксис нужно инициализировать данные состояния объекта **SocketPermissionAttribute**, чтобы он представлял форму разрешения, соблюдение которого обеспечивается в коде.

В следующем примере демонстрируется, как обеспечить выполнение разрешения с использованием **SocketPermissionAttribute**:

```
1  using System;
2  using System.Text;
3  using System.Net;
4  using System.Net.Sockets;
5  using System.Security;
6  using System.Security.Permissions;
7
8  namespace SocketPermissionSample
9  {
10     class Program
11     {
12         static void Main(string[] args)
13         {
14             LegalMethod();
15             Console.ReadLine();
16         }
17
18         [SocketPermission(SecurityAction.Assert, Access="Connect",
19             Host="localhost", Port="All", Transport="Tcp")]
20         public static void LegalMethod()
21         {
22             Console.WriteLine("Legal Method");
23
24             // Устанавливаем удаленную точку для сокета
25             IPHostEntry ipHost = Dns.GetHostEntry("localhost");
26             IPAddress ipAddr = ipHost.AddressList[0];
27             IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);
28
29             Socket sender = new Socket(ipAddr.AddressFamily,
30                 SocketType.Stream, ProtocolType.Tcp);
31
32             try
33             {
34                 // Соединяем сокет с удаленной endPoint
35                 sender.Connect(ipEndPoint);
36                 Console.WriteLine("Сокет подключен к {0}",
37                     sender.RemoteEndPoint.ToString());
38             }
39             catch (Exception ex)
40             {
41                 Console.WriteLine(ex.ToString());
42             }
43             finally
44             {
45                 if (sender.Connected)
46                 {
47                     // Освобождаем сокет
48                     sender.Shutdown(SocketShutdown.Both);
49                     sender.Close();
50                 }
51             }
52         }
53     }
54 }
55 }
```



По функциональности приведенная выше программа похожа на предыдущую, однако, в последней программе вместо императивной безопасности используется синтаксис декларативной безопасности, т.е. разрешения для сокета задаются в атрибуте `SocketPermission`.

Содержание работы

1. Откройте новый документ Word, выполните настройку документа и заполните необходимую информацию согласно методических указаний. Сохраните документ в папке «Мои документы» с именем «Фамилия, группа, Пр.р.№», не забывайте периодически сохранять документ в процессе выполнения работы.

2. Внимательно изучите краткие теоретические сведения.

3. Оформите предложенный текст в соответствии с требованиями к проекту.

4. Сделайте выводы, подготовьтесь к защите.

Контрольные вопросы

1. Понятие технологии Windows Sockets.
2. Основные этапы работы с сокетами.
3. Понятие технологии «клиент-сервер».
4. Что такое полнодуплексное взаимодействие?
5. Что такое динамически подключаемые библиотеки (DLL)?
6. Что такое прикладной программный интерфейс (WinAPI)?
7. Блокирующие и неблокирующие сокеты.
8. Модель OSI, модель TCP/IP.

ПРАКТИЧЕСКАЯ РАБОТА №25-30

Тема: Работы с буфером экрана

Цель: Изучить работу с буфером экрана.

Оборудование: В соответствии с рабочей программой ПМ.01 «Разработка модулей программного обеспечения для компьютерных систем»:

- Автоматизированные рабочие места на 12-15 обучающихся (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Автоматизированное рабочее место преподавателя (процессор не ниже Core i3, оперативная память объемом не менее 4 Гб) или аналоги;
- Проектор и экран;
- Маркерная доска;
- Лицензионное программное обеспечение общего и профессионального назначения.

Справочный материал

Пространство имен System.Drawing обеспечивает доступ к функциональным возможностям графического интерфейса GDI+.

Пространства имен:

**System.Drawing.Drawing2D,
System.Drawing.Imaging,
System.Drawing.Text**

обеспечивают дополнительные функциональные возможности.

Класс **Graphics** предоставляет методы рисования на устройстве отображения. Такие классы, как Rectangle и Point, инкапсулируют элементы GDI+.

Класс Pen используется для рисования линий и кривых, а классы, производные от абстрактного класса Brush, используются для заливки фигур.

Класс - **Graphics**

Инкапсулирует поверхность рисования **GDI+**. Этот класс не наследуется.

Методов в этом классе огромное количество, поэтому рассмотрим некоторые из них:

Имя	Описание
AddMetafileComment	Добавляет комментарий к текущему объекту Metafile.
BeginContainer()	Сохраняет графический контейнер, содержащий текущее состояние данного объекта Graphics, а затем открывает и использует новый графический контейнер.

Имя	Описание
BeginContainer(Rectangle, Rectangle, GraphicsUnit)	Сохраняет графический контейнер, содержащий текущее состояние данного объекта Graphics, а также открывает и использует новый графический контейнер с указанным преобразованием масштаба.
BeginContainer(RectangleF, RectangleF, GraphicsUnit)	Сохраняет графический контейнер, содержащий текущее состояние данного объекта Graphics, а также открывает и использует новый графический контейнер с указанным преобразованием масштаба.
Clear	Очищает всю поверхность рисования и выполняет заливку поверхности указанным цветом фона.
CopyFromScreen(Point, Point, Size)	Выполняет передачу данных о цвете, соответствующих прямоугольной области пикселей, блоками битов с экрана на поверхность рисования объекта Graphics.
CopyFromScreen(Point, Point, Size, CopyPixelOperation)	Выполняет передачу данных о цвете, соответствующих прямоугольной области пикселей, блоками битов с экрана на поверхность рисования объекта Graphics.
CopyFromScreen(Int32, Int32, Int32, Int32, Size)	Выполняет передачу данных о цвете, соответствующих прямоугольной области пикселей, блоками битов с экрана на поверхность рисования объекта Graphics.
Dispose	Освобождает все ресурсы, используемые данным объектом Graphics.
DrawArc(Pen, Rectangle, Single, Single)	Рисует дугу, которая является частью эллипса, заданного структурой Rectangle.
DrawArc(Pen, Int32, Int32, Int32, Int32, Int32, Int32)	Рисует дугу, которая является частью эллипса, заданного парой координат, шириной и высотой.
DrawArc(Pen, Single, Single, Single, Single, Single, Single)	Рисует дугу, которая является частью эллипса, заданного парой координат, шириной и высотой.
DrawBezier(Pen, Point, Point, Point, Point)	Рисует кривую Безье, определяемую четырьмя структурами Point.
DrawLine(Pen, Point, Point)	Проводит линию, соединяющую две структуры Point.
DrawLine(Pen, Int32, Int32, Int32, Int32)	Проводит линию, соединяющую две точки, задаваемые парами координат.
DrawLine(Pen, Single, Single, Single, Single)	Проводит линию, соединяющую две точки, задаваемые парами координат.

Имя	Описание
DrawLines(Pen, Point[])	Рисует набор сегментов линий, которые соединяют массив структур Point.
DrawPath	Рисует объект GraphicsPath.
DrawPie(Pen, Rectangle, Single, Single)	Рисует сектор, который определяется эллипсом, заданным структурой Rectangle и двумя радиальными линиями.
DrawPie(Pen, Int32, Int32, Int32, Int32, Int32, Int32)	Рисует сектор, определяемый эллипсом, который задан парой координат, шириной, высотой и двумя радиальными линиями.
DrawPie(Pen, Single, Single, Single, Single, Single, Single)	Рисует сектор, определяемый эллипсом, который задан парой координат, шириной, высотой и двумя радиальными линиями.
DrawPolygon(Pen, Point[])	Рисует многоугольник, определяемый массивом структур Point.
DrawRectangle(Pen, Rectangle)	Рисует прямоугольник, определяемый структурой Rectangle.
DrawRectangle(Pen, Int32, Int32, Int32, Int32)	Рисует прямоугольник, который определен парой координат, шириной и высотой.
DrawRectangle(Pen, Single, Single, Single, Single)	Рисует прямоугольник, который определен парой координат, шириной и высотой.
DrawRectangles(Pen, Rectangle[])	Рисует набор прямоугольников, определяемых структурами Rectangle.
DrawString(String, Font, Brush, PointF)	Создает указываемую текстовую строку в заданном месте с помощью определяемых объектов Brush и Font.
DrawString(String, Font, Brush, PointF, StringFormat)	Рисует заданную текстовую строку в заданном месте с помощью определяемых объектов Brush и Font, используя атрибуты форматирования заданного формата StringFormat.
DrawString(String, Font, Brush, Single, Single)	Создает указываемую текстовую строку в заданном месте с помощью определяемых объектов Brush и Font.
Equals(Object)	Определяет, равен ли заданный объект текущему объекту. (Унаследовано от Object.)
ExcludeClip(Rectangle)	Обновляет вырезанную область данного объекта Graphics, чтобы исключить из нее часть, определяемую структурой Rectangle.
ExcludeClip(Region)	Обновляет вырезанную область данного объекта Graphics, чтобы исключить из нее часть, определяемую структурой Region.

Имя	Описание
FillClosedCurve(Brush, Point[])	Заполняет внутреннюю часть замкнутой фундаментальной кривой, определяемой массивом структур Point.
FillClosedCurve(Brush, Point[], FillMode)	Заполняет внутреннюю часть замкнутой фундаментальной сплайновой кривой, определяемой массивом структур Point, используя указанный режим заливки.
FillClosedCurve(Brush, Point[], FillMode, Single)	Заполняет внутреннюю часть замкнутой фундаментальной кривой, определяемой массивом структур Point, используя указанные режим заливки и натяжение.
FillEllipse(Brush, Rectangle)	Заполняет внутреннюю часть эллипса, определяемого ограничивающим прямоугольником, который задан структурой Rectangle.
FillEllipse(Brush, Int32, Int32, Int32, Int32)	Заполняет внутреннюю часть эллипса, определяемого ограничивающим прямоугольником, заданным с помощью пары координат, ширины и высоты.
FillEllipse(Brush, Single, Single, Single, Single)	Заполняет внутреннюю часть эллипса, определяемого ограничивающим прямоугольником, заданным с помощью пары координат, ширины и высоты.
FillPath	Заполняет внутреннюю часть объекта GraphicsPath.
FillPie(Brush, Rectangle, Single, Single)	Заполняет внутреннюю часть сектора, определяемого эллипсом, который задан структурой RectangleF, и двумя радиальными линиями.
FillPie(Brush, Int32, Int32, Int32, Int32, Int32, Int32)	Заполняет внутреннюю часть сектора, определяемого эллипсом, который задан парой координат, шириной, высотой и двумя радиальными линиями.
FillPie(Brush, Single, Single, Single, Single, Single, Single)	Заполняет внутреннюю часть сектора, определяемого эллипсом, который задан парой координат, шириной, высотой и двумя радиальными линиями.
FillPolygon(Brush, Point[])	Заполняет внутреннюю часть многоугольника, определяемого массивом точек, заданных структурами Point.
FillPolygon(Brush, Point[], FillMode)	Заполняет внутреннюю часть многоугольника, определенного массивом точек, заданных структурами Point, используя указанный режим заливки.

Имя	Описание
FillRectangle(Brush, Rectangle)	Заполняет внутреннюю часть прямоугольника, определяемого структурой Rectangle.
FillRectangle(Brush, Int32, Int32, Int32, Int32)	Заполняет внутреннюю часть прямоугольника, который определен парой координат, шириной и высотой.
FillRectangle(Brush, Single, Single, Single, Single)	Заполняет внутреннюю часть прямоугольника, который определен парой координат, шириной и высотой.
FillRectangles(Brush, Rectangle[])	Заполняет внутреннюю часть набора прямоугольников, определяемых структурами Rectangle.
FillRegion	Заполняет внутреннюю часть объекта Region.
Flush()	Вызывает принудительное выполнение всех отложенных графических операций и немедленно возвращается, не дожидаясь их окончания.
Flush(FlushIntention)	Вызывает принудительное выполнение всех отложенных графических операций. При этом в соответствии с настройкой метод дожидается или не дожидается окончания операций для возврата.

Класс - **Pen** определяет объект, используемый для рисования прямых линий и кривых. Этот класс не наследуется.

Pen(Color)	Инициализирует новый экземпляр класса Pen с указанным цветом.
Pen(Color, Single)	Инициализирует новый экземпляр класса Pen с указанными свойствами Color и Width. (Width - устанавливает ширину пера Pen, в единицах объекта Graphics, используемого для рисования)

Класс - **Brush** определяет объекты, которые используются для заливки внутри графических фигур, таких как прямоугольники, эллипсы, круги, многоугольники и дорожки.

Это абстрактный базовый класс, который не может быть реализован. Для создания объекта "кисть" используются классы, производные от Brush, такие как SolidBrush, TextureBrush и LinearGradientBrush.

Содержание работы

1. Откройте новый документ Word, выполните настройку документа и заполните необходимую информацию согласно методических указаний. Сохраните документ в папке «Мои документы» с именем «Фамилия, группа, Пр.р.№», не забывайте периодически сохранять документ в процессе выполнения работы.

2. Внимательно изучите краткие теоретические сведения.

3. Оформите предложенный текст в соответствии с требованиями к проекту.

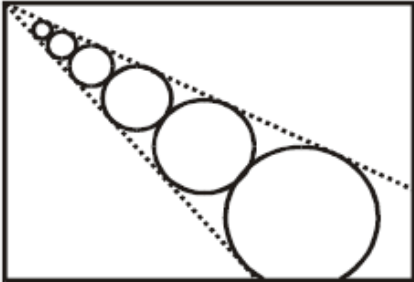
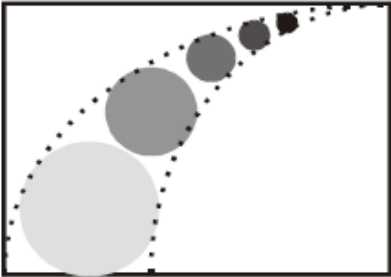
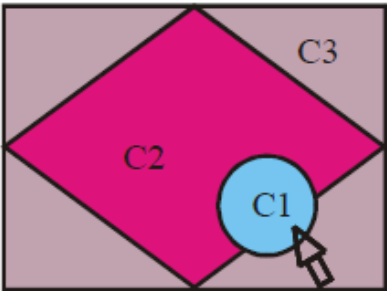
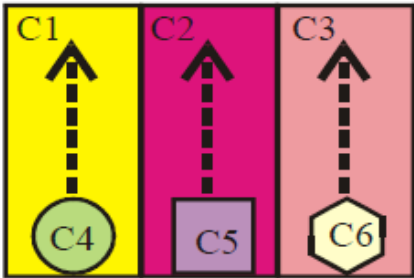
4. Сделайте выводы, подготовьтесь к защите.


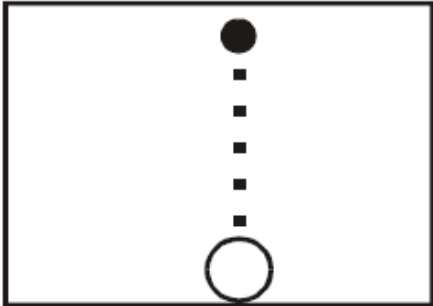

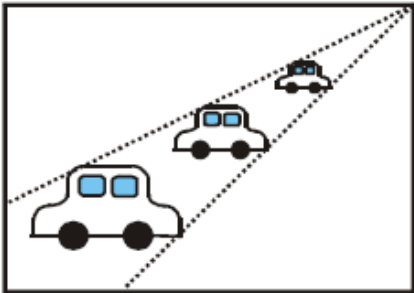
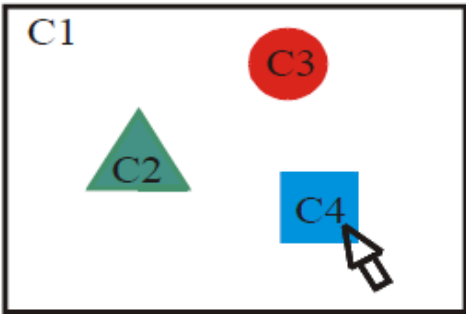
Задание

Фигуры, на которые на рисунке указывает курсор мыши, двигаются по экрану при нажатой левой клавиши мыши по траектории движения мыши.

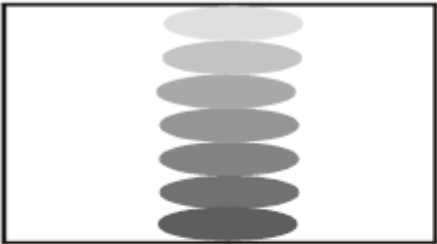
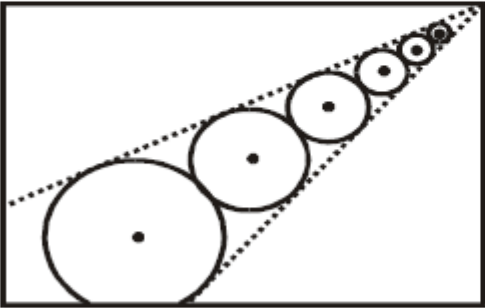
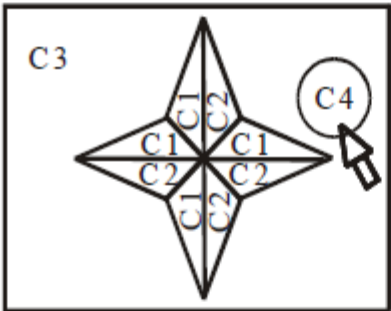

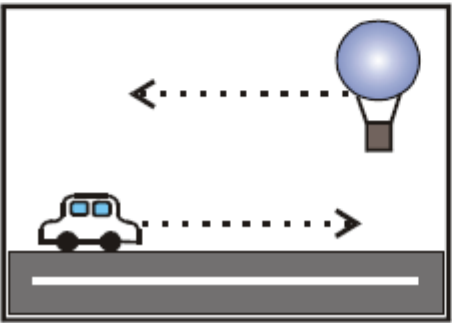
Переменные C1, C2, C3... и т.д. обозначают цвета фигур и участков экрана. Если индексы в переменных разные, то и цвета должны быть разные.

Вывод всех рисунков на экран осуществляется методом двойной буферизации.

№	Рисунок	Пояснения
1		По экрану по показанной траектории движется круг, изменяя свой радиус.
2		По экрану по показанной траектории движется круг, изменяя свой радиус и цвет.
3		На экране цвета C3 через Δt_1 появляется фигура цвета C2, а через Δt_2 фигура цвета C1.
4		Фигуры из каждого сектора экрана через Δt_1 после запуска программы начинают двигаться вверх.

№	Рисунок	Пояснения
5		Через Δt_1 после запуска программы на экране получить данную картинку.
6		По экрану по показанной траектории движется круг, изменяя свой радиус и цвет.
7		В программе имитировать восход солнца.
8		В программе имитировать приближение автомобиля к плоскости экрана.
9		Фигуры появляются на экране друг за другом последовательно через Δt_1 .

№	Рисунок	Пояснения
10		На экране имитировать движение маятника. Левая кнопка мыши колебания начинаются, правая – заканчиваются.
11		На экране имитировать движение маятника. Правая кнопка мыши колебания начинаются, левая – заканчиваются.
12		Фигура появляется на экране через Δt_1 .
13		По траектории показанной на рисунке движется круг, изменяя свой цвет в зависимости от своего местоположения.
14		На экране цвета C1 через Δt_1 появляется круг цвета C2, а через Δt_2 круг цвета C3, который начинает хаотично двигаться внутри другого круга.
15		Фигуры появляются на экране друг за другом последовательно через Δt_1 .

№	Рисунок	Пояснения
16		По экрану по показанной траектории движется эллипс, изменяя свой цвет.
17		По экрану по показанной траектории движется круг, изменяя свой радиус.
18		На экране получить разноцветную фигуру, по которой может двигаться круг.
19		На экране симитировать мерцание звезд на ночном небе
20		На экране проиллюстрировать одновременное движение автомобиля и полет воздушного шара в указанных направлениях. Предметы начинают движение по левому щелчку мыши.

Информационное обеспечение обучения

Основные учебные издания:

1. Аблязов, Р.З. Программирование на ассемблере на платформе x86-64/Р.З. Аблязов. — 2-е изд. — Саратов: Профобразование, 2019. — 301 с. — ISBN 978-5-4488-0117-4. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: <https://www.iprbookshop.ru/88005.html>
2. Введение в разработку приложений для ОС Android: учебное пособие / Ю. В. Березовская, О. А. Юфрякова, В. Г. Вологодина [и др.]. — 3-е изд. — Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. — 427 с. — ISBN 978-5-4497-0890-8. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: <https://www.iprbookshop.ru/102000.html>
3. Зыков, С. В. Введение в теорию программирования. Объектно-ориентированный подход: учебное пособие для СПО / С. В. Зыков. — Саратов: Профобразование, 2021. — 187 с. — ISBN 978-5-4488-0995-8. — Текст: электронный // Электронный ресурс цифровой образовательной среды СПО ПРОФобразование: [сайт]. — URL: <https://profspo.ru/books/102188>
4. Кариев, Ч. А. Разработка Windows-приложений на основе Visual C#: учебное пособие / Ч. А. Кариев. — 3-е изд. — Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. — 978 с. — ISBN 978-5-4497-0909-7. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: <https://www.iprbookshop.ru/102057.html>
5. Котляров, В.П. Основы тестирования программного обеспечения: курс лекций / Котляров В.П. — Москва: Интуит НОУ, 2016. — 348 с. — ISBN 978-5-9556-0027-7. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: <https://book.ru/book/917951>
6. Лебедева, Т. Н. Технология программирования: учебное пособие для СПО/ Т. Н. Лебедева, С. С. Юнусова. — Саратов: Профобразование, 2019. — 140 с. — ISBN 978-5-4488-0351-2. — Текст: электронный // Электронный ресурс цифровой образовательной среды СПО ПРОФобразование: [сайт]. — URL: <https://profspo.ru/books/86081>

Дополнительные учебные издания:

7. Авдеев, В. А. Периферийные устройства: интерфейсы, схемотехника, программирование / В. А. Авдеев. — 2-е изд. — Саратов: Профобразование, 2019. — 848 с. — ISBN 978-5-4488-0053-5. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: <https://www.iprbookshop.ru/88002.html>
8. Биллиг, В. А. Основы программирования на C#: учебное пособие / В. А. Биллиг. — 3-е изд. — Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. — 573 с. — ISBN 978-5-4497-0893-9. — Текст: электронный // Электронный ресурс цифровой образовательной среды СПО ПРОФобразование: [сайт]. — URL: <https://profspo.ru/books/102033>
9. Зубкова, Т. М. Технология разработки программного обеспечения: учебное пособие для СПО / Т. М. Зубкова. — Саратов: Профобразование, 2019. — 468 с. — ISBN 978-

5-4488-0354-3. — Текст: электронный // Электронный ресурс цифровой образовательной среды СПО PROФобразование: [сайт]. — URL: <https://profspo.ru/books/86208>

10. Пирская, Л. В. Разработка мобильных приложений в среде Android Studio: учебное пособие / Л. В. Пирская. — Ростов-на-Дону, Таганрог: Издательство Южного федерального университета, 2019. — 123 с. — ISBN 978-5-9275-3346-6. — Текст: электронный // Электронно-библиотечная система IPR BOOKS: [сайт]. — URL: <https://www.iprbookshop.ru/100196.html>

11. Семакова, А. Введение в разработку приложений для смартфонов на ОС Android: учебное пособие для СПО/ А. Семакова. — Саратов: Профобразование, 2021. — 102 с. — ISBN 978-5-4488-0994-1. — Текст: электронный // Электронный ресурс цифровой образовательной среды СПО PROФобразование: [сайт]. — URL: <https://profspo.ru/books/102187>

Электронные издания (электронные ресурсы)

12. Учебники по программированию <http://programm.ws/index.php>

13. ЭБС - <https://www.iprbookshop.ru>.

14. ЭБС - <https://book.ru>.

15. ЭБС - <https://profspo.ru>.

16. ЭБС - <https://znanium.com/>