МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РЕСПУБЛИКИ КАЗАХСТАН

Восточно-Казахстанский технический университет им. Д. Серикбаева

ЖҮЙЕЛІК БАҒДАРЛАМАЛАУ

Лекция конспектісі

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Конспект лекций

Жомартқызы Г., Сулейменова Л.Р.

Специальность: 6В06102 «Вычислительная техника и программное обеспечение»

Өскемен
Усть-Каменогорск
2024

СОДЕРЖАНИЕ

1.1 Основные понятия системного программирования.

1.1.1 Введение

Название «Системное программирование» неразрывно связано с операционными системами компьютеров (ОС). Исторически сложилось так, что первыми большими программами были программы ОС ЭВМ.

Прикладные программы были ограничены объем оставшейся свободной оперативной памяти компьютера и в основном решали вычислительные задачи.

В настоящее время трудно различить, какие программы являются более сложными – прикладные или системные т.к. некоторые прикладные программы фактически являются большими системными проектами и по своей сложности не уступают программам ОС.

В дисциплине «Системное программирование» рассматриваются алгоритмы и решения задач операционной системы.

1.1.2 Понятие и назначение операционных систем

В понятие компьютера включаются как физические или аппаратные, так и логические или информационные ресурсы.

К физическим ресурсам относятся различные устройства компьютера – процессор, устройства ввода-вывода, память и т.д.

К логическим ресурсам относят данные и программы, по которым работает компьютер.

Все ресурсы компьютера обычно называют системными ресурсами.

Тогда операционную систему можно определить как комплекс программ, которые обеспечивают доступ и управление системными ресурсами компьютера.

Программы, работающие на компьютере под управлением ОС, называют пользовательскими программами.

Пользовательские программы, предназначенные для решения одной задачи, называются приложением.

Если ОС позволяет выполнять только одну пользовательскую программу, то ОС называется однопользовательской или однопрограммной.

Если ОС позволяет одновременно выполнять несколько пользовательских программ, то ее называют многопользовательской или мультипрограммной.

Если ОС может работать только на компьютере с одним процессором, то ее называют однопроцессорной.

Если ОС может работать на компьютере, аппаратный ресурс которого может содержать один или несколько процессоров, то ОС называется мультипроцессорной.

Операционные системы, работающие без участия пользователя в режиме реального времени, называют ОС реального времени. Такие операционные системы предназначены для управления некоторых технологических процессов.

1.1.3 Понятие и назначение интерфейса прикладного программирования Win32 API.

Каждая ОС предоставляет пользователю возможность использования практически всех своих системных ресурсов – пользовательский интерфейс, включающий наборы типов, констант, переменных, функций и классов для программирования приложений, работающих под управлением ОС.

Все, работающие в настоящий момент ОС Windows, имеют практически одинаковый интерфейс программирования приложений –Win32 API (Application Programming Interface).

Естественно Win32 API используется не только в прикладном, но и в системном программировании. Условно все функции Win32 API можно подразделить на следующие категории:

– базовые сервисы (Base Services);
– библиотека общих элементов управления (Common Control Library);
– интерфейс графических устройств (Graphics Devices Interface);
– сетевые сервисы (Network Services);
– интерфейс пользователя (Uses Interface);
– управление доступом для Windows (Windows Access Control);
– оболочка Windows (Windows Shell);
– информация о системе Windows (Windows System Information).

В дисциплинах, связанных с разработкой прикладных программ, в основном изучаются функции интерфейса пользователя.

Функции сетевого сервиса изучаются в дисциплинах, связанные с работой локальных вычислительных сетей.

Интерфейс графических устройств интенсивно используется при разработке различных игровых программ.

Назначение и использование остальных функций Win32 API в основном изучается в дисциплине «Системное программирование».

1.1.4 Понятие объектов и дескрипторов в Windows

Использование функций Win32 API базируется на использовании объектов, при этом необходимо помнить, что «классическое» определение объекта как переменной класса стало применяться после разработки функций Win32 API.

Объектом в Windows называется структура данных, которая содержит системный ресурс. Фактически объект Windows это область памяти, выделенную Windows и содержащую информацию об объекте. Объекты

создаются специальными функциями и доступ к объектам возможен только с помощью функций ОС. Win32 API может создавать объекты трех категорий:
– объекты интерфейса пользователя (User);
– объекты интерфейса графических устройств (Graphics Devise Interface);
– объекты ядра операционной системы (Kernel).

Мы будем рассматривать только объекты ядра операционной системы. При создании объекта ему присваивается имя (идентификатор), которое называется дескриптором (handle).В ОС дескриптор объекта представляет собой запись в таблице дескрипторов, которая содержит адрес объекта и средства для идентификации типа объекта.

В Win32 API дескрипторы имеют тип HANDLE. При обращении к объекту необходимо указывать его дескриптор. По окончании работ с объектом его необходимо закрывать.

1.1.5 Понятие динамически подключаемых библиотек

«Динамически подключаемые библиотеки (dynamic-link libraries, DLL) — краеугольный камень операционной системы Windows, начиная с самой первой её версии. В DLL содержатся все функции функций Win32 API. Три самые важные DLL: Kernel32.dll (управление памятью, процессами и потоками), User32.dll (поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений) и GDI32.dll (графика и вывод текста).

В Windows есть и другие DLL, функции которых предназначены для более специализированных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll — стандартные диалоговые окна (вроде FileOpen и FileSave), а ComCtl32.dll поддерживает стандартные элементы управления.

Зачем нужны динамически подключаемые библиотеки? Вот лишь некоторые из причин, по которым нужно применять DLL:
– расширение функциональности приложения. DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код.

Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет DLL от других компаний.
– возможность использования разных языков программирования. У Вас есть выбор, на каком языке писать ту или иную часть приложения;
– более простое управление проектом. Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще.
– экономия памяти. Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один её экземпляр, доступный этим приложениям.

– разделение ресурсов. DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам.
– решение проблем, связанных с особенностями различных платформ. В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Ваша версия Windows не поддерживает эти функции, Вам не удастся запустить такое приложение: загрузчик попросту откажется его запускать. Но если эти функции будут находиться в отдельной библиотеке (DLL), то Вы загрузите программу даже в более ранних версиях Windows.» (Джеффри Рихтер Windows для профессионалов, стр. 476-477).

1.1.6 Разработка DLL библиотека для некоторых задач

Разработаем учебную DLL библиотеку для задач, связанных с некоторой обработкой целых чисел в одномерном массиве. Это чисто учебная задача, в которой рассматривается технология создания DLL библиотеки. Одновременно рассмотрены вопросы использование DLL библиотеки для работы в консольном приложении языка C# .

Пусть наша DLL библиотека будет содержать только три метода работы с массивом целых чисел:
- поиск максимального значения;
- сортировку в порядке убывания;
- вычисления общей суммы всех значений массива.

Запустим Visual Studio 2008и перейдем к созданию проекта. В качестве типа проекта укажем тип "Class Library". Переопределим поля окна создания DLL библиотеки в соответствии с рисунком 1.1.1.

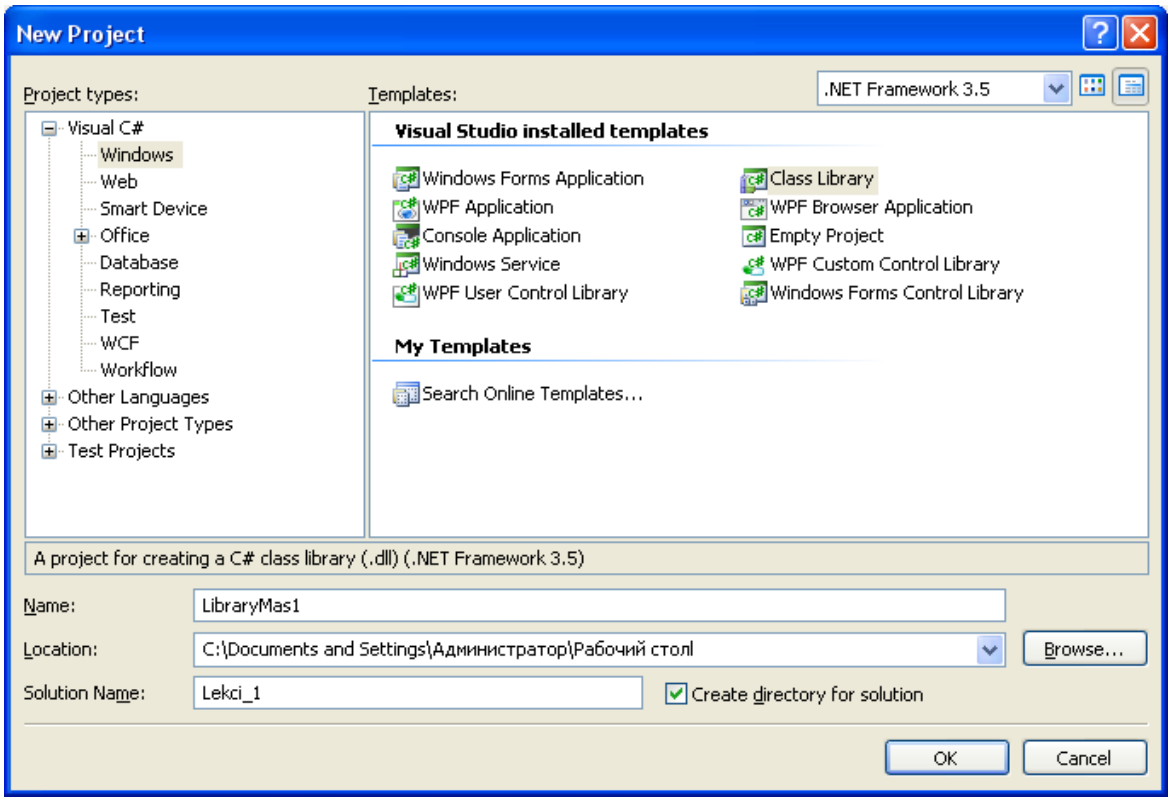


Рисунок 1.1.1 – Создание DLL библиотеки

В поле Namезадается имя строящейся DLL. Пусть имя создаваемой DLL будет LibraryMas1.

В полеLocationуказывается путь к папке, где будет храниться Решение, содержащее проект. Укажем рабочий стол компьютера.

В окне SolitionName задается имя «Решения». Предлагается использовать имя Lekci_1, указывающее на то, что все проекты первой лекции вложены в одно «Решение».

В поле SolitionName выбран элемент "Create directory for solution", указывающий, что нужно создать новую папку для размещения «Решение».
После подтверждения типа проекта нажатием кнопки ОК получим следующий код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LibraryMas1
{
publicclassClass1
    {
}
}
```

Изменим имя "Class1" на имя "MyMas" для этого в окне кода проекта выделяем имя изменяемого идентификатора, затем в главном меню выбираем

пункт Refactor и подпункт Rename. В открывшемся окне указываем новое имя. Тогда будут показаны все места, требующие переименования идентификатора. В нашем случае будет только одна замена, но, в общем случае, замен может быть много, так что автоматическая замена всех вхождений крайне полезна.

Наполним создаваемую DLL библиотеку кодом трех методов в соответствии с условием задания.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace LibraryMas1
{
publicclassMyMas
    {
publicstaticint maxMas(refint[] masi, int n)
        {
int max = masi[0];
for (int i = 1; i < n; i++)
if (max < masi[i]) max = masi[i];
return max;
        }
publicstaticvoid sortMas(refint[] masi, int n)
        {
int b = 0;
for (int i = 0; i < n - 1; i++)
for (int j = i + 1; j < n; j++)
if (masi[i] < masi[j])
            { b = masi[i]; masi[i] = masi[j]; masi[j] = b; }
}
publicstaticint symMas(refint[] masi, int n)
        {
int sym = 0;
for (int i = 0; i < n; i++)
sym=sym+masi[i];
return sym;
} }}
```

На заключительном этапе создания DLL библиотеки необходимо выполнить два действия.

Во-первых, необходимо переименовав имя файла «Class1.cs» в имя «MyMas.cs». Так как имя файла, хранящего класс, и имя класса должны совпадать. Переименование имени файла делается непосредственно в окне проектов Solition Explorer. После выделения имени файла нажимаем правую кнопку мышки и выбираем команду Rename. Вводим новое имя – изменения происходят во всех элементах проекта.

Во-вторых, необходимо скомпилировать созданную DLL библиотеку для чего в Главном меню среды выберем пункт Build ->Build Solition. В результате успешной компиляции будет построен файл с расширением dll.

1.1.7 Использование созданной DLL библиотеки

Рассмотрим чисто учебную программу для работы с массивом, в которой используем созданную DLL библиотеку. Программа предназначена для решения следующей задачи.

Задача 1.1 Сформировать массив из 20 случайных целых чисел в диапазоне от минус 50 до 50. Напечатать его. Найти и напечатать сумму всех элементов массива. Выполнить сдвиг значений массива на 1 разряд влево. Выполнить сортировку элементов массива в порядке убывания. Предусмотреть меню программы.

Один из вариантов использования созданной DLL библиотеки предполагает создание приложения в одном с DLL «Решении». Для этого обычным способом открываем проект созданной ранее DLL библиотеке и в нем создаем проект консольного приложение в соответствии с рисунком 1.1.2.

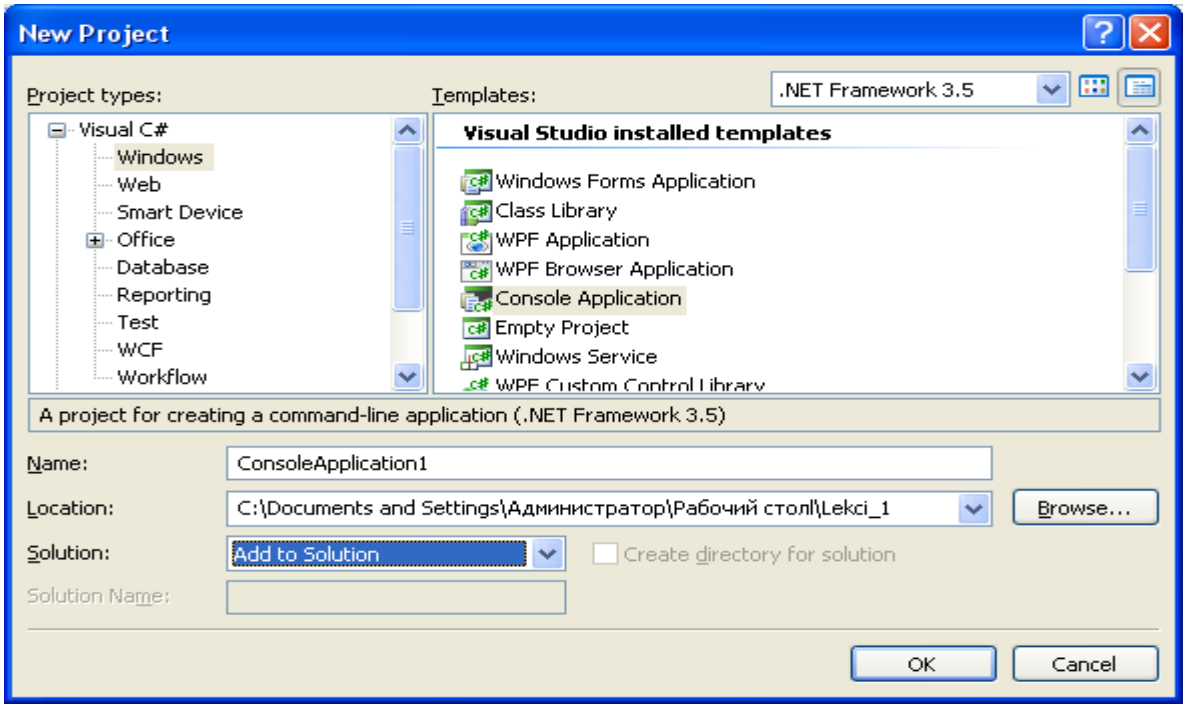


Рисунок 1.1.2 – Создание консольного приложения

В поле Name оставляем значение по умолчанию.

В поле Location оставляем значение по умолчанию.

В поле Solition выбираем значение "Add to Solution", указывающий, что создаваемое приложение будет добавлено в существующее «Решение».

Заготовка кода консольного приложения представлена на рисунке 1.1.3.

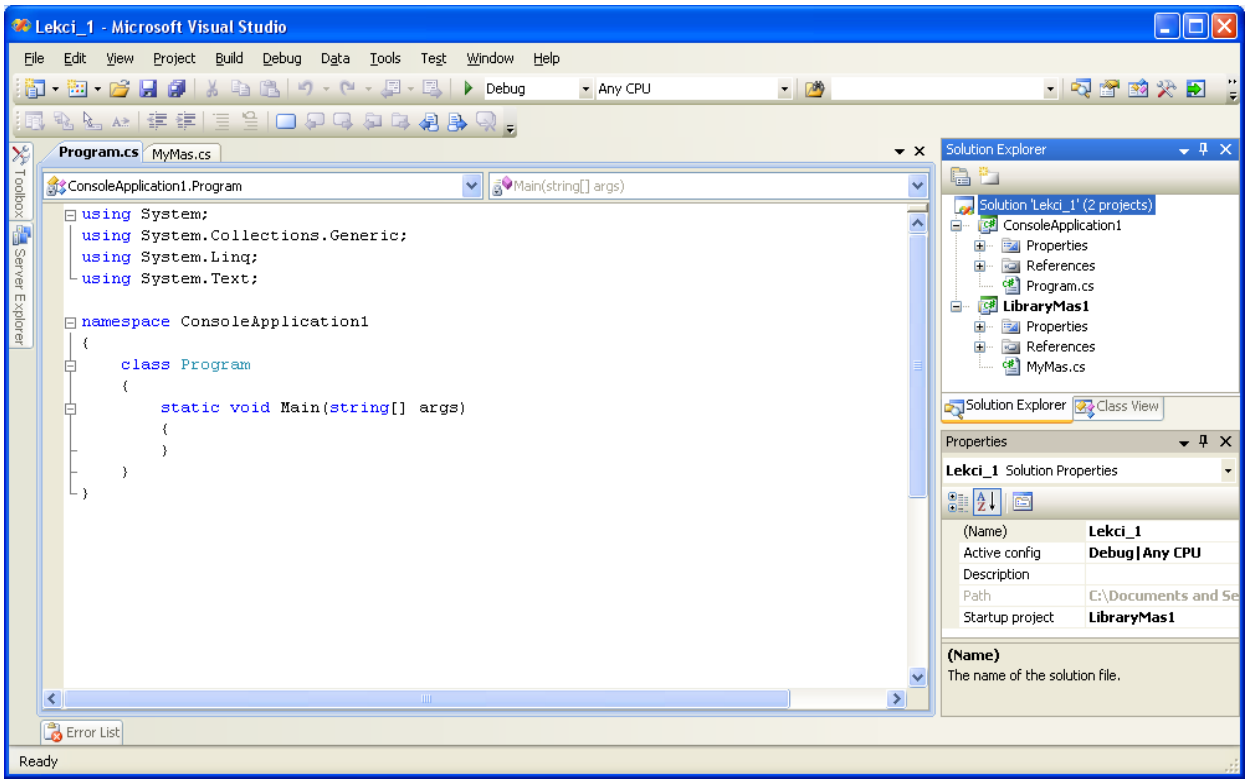
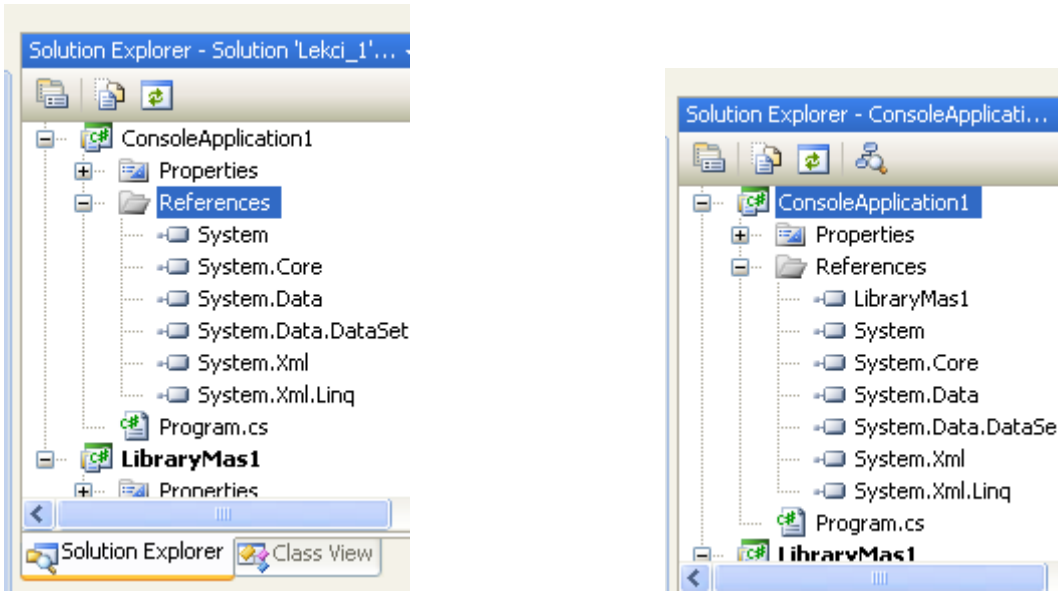


Рисунок 1.1.3 – Начальное окно создания консольного приложения

Обратите внимание, что окно «Solition Explorer» содержит два проекта. Оба проекта находятся в одном решении, но не связаны друг с другом. Это легко проверить, если в окне «Solition Explorer» папку «References» - смотрите рисунок 1.1.4–a.



a b

Рисунок 1.1.4 – Связи консольного приложения

Отсутствие связей делает невозможным использования методов DLL библиотеки в нашем консольном проекте.

Для организации связи консольного приложения с DLL (добавить ссылку на проект с DLL LibraryMas1) необходимо в окне «Solition Explorer» выбрать имя консольного приложения (Console Application1) и из контекстного меню, появляющегося при щелчке правой кнопки, выберем пункт меню "Add Reference". В открывшемся окне добавления ссылок выберем вкладку "Projects". Поскольку проект LibraryMas1 включен в «Решение», то он автоматически появится в открывшемся окне. Подтверждаем установку связи нажатием кнопки ОК. Ссылка на DLL LibraryMas1 появится в папке "References" консольного приложения. На рисунке 1.1.4 – b видно, что ссылка на DLL LibraryMas1 появилась вверху списка ссылок. Теперь проекты связаны и из консольного проекта доступны методы, предоставляемые DLL.

Ссылку на DLL LibraryMas1 можно установить, используя режим Project->Add Reference.

Если ссылку нужно установить на проект, не включенный в «Решение», то в окне добавления ссылок нужно задать путь к проекту.

В соответствии с условиями задачи разрабатываем код программы.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
classProgram
 {
publicstaticvoid sozd(refint[] ma)
  {
Random rnd = newRandom();
for (int i = 0; i < 20; i++)
    ma[i] = rnd.Next() % 101 - 50;
Console.WriteLine("Массивсоздан !!");
  }
publicstaticvoid zadvig(refint[] ma)
  {
int k;
for (int i = 0; i < 19; i++)
{
    k = ma[i]; ma[i] = ma[i + 1]; ma[i + 1] = k;
}
Console.WriteLine("Сдвиг массива на 1 разряд выполнен !");
}
publicstaticvoid prinmas(int[] ma)
  {
for (int i = 0; i < 20; i++)
Console.Write(" {0}", ma[i]);
```

```
Console.WriteLine();
  }
staticvoid Main()
  {
int[] a = newint[20];
int k = 0;
int s = 0,n = 20;
string buf;
while (k < 6)
   {
Console.WriteLine("1 - Создатьмассив 20 чисел");
Console.WriteLine("2 - Напечатать массив");
Console.WriteLine("3 - Найти и напечатать сумму всех элементов
массива");
Console.WriteLine("4 - Выполнить сдвиг значений массива на 1
разряд влево");
Console.WriteLine("5 - Выполнить сортировку элементов массива в
порядке убывания");
Console.WriteLine("6 - Выход из программы");
Console.WriteLine("Введите пункт меню программы");
buf = Console.ReadLine();
    k = Convert.ToInt32(buf);
switch (k)
    {
case 1: sozd(ref a); break;
case 2: prinmas(a); break;
case 3:
      {
       s = LibraryMas1.MyMas.symMas(ref a, n);
Console.WriteLine("Суммаэлементовмассива = {0}",s);
      }; break;
case 4: zadvig(ref a); break;
case 5: LibraryMas1.MyMas.sortMas(ref a,n); break;
default: break;
    }
   }
  }
 }
}
```

Если попытаться запустить проект на выполнение, то появится сообщение, что не назначен запускаемый проект. Это возможно когда в «Решении» несколько проектов. Чтобы определить первый запускаемый проект необходимо находиться в окне редактора этого проекта и задать следующую команду: Project ->Set as Start Up Project.
Работа программы:

1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево

```
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
1
Массив создан !!
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
2
 -36 -2 -3 35 -10 28 -32 11 -21 -32 -35 13 6 3 -45 32 -27 -5 -36 15
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
3
Сумма элементов массива = -141
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
4
Сдвиг массива на 1 разряд выполнен !
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
2
 -2 -3 35 -10 28 -32 11 -21 -32 -35 13 6 3 -45 32 -27 -5 -36 15 -36
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
5
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
```

```
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
2
 35 32 28 15 13 11 6 3 -2 -3 -5 -10 -21 -27 -32 -32 -35 -36 -36 -45
1 - Создать массив 20 чисел
2 - Напечатать массив
3 - Найти и напечатать сумму всех элементов массива
4 - Выполнить сдвиг значений массива на 1 разряд влево
5 - Выполнить сортировку элементов массива в порядке убывания
6 - Выход из программы
Введите пункт меню программы
```

Тема 2 ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ В ЯЗЫКЕС#

2.1 Побитовые операции в языке C#

2.1.1 Системы счисления

Существует два основных способа представления информации – с помощью непрерывной или аналоговой формы и дискретной или цифровой формы.

Непрерывная форма является основной для представления информации в живой природе, например, звуковые сигналы, вкусовые качества, болевые ощущения, оттенки цветов и т.д. все они представляются аналоговыми сигналами. Очень часто непрерывные сигналы изображаются на плоскости в координатах функции от времени в виде непрерывных графиков.

Дискретная форма представления информации чаще используется в технических устройствах, например, в системах автоматики, компьютерах. На графиках функции от времени дискретная форма представляется в виде «ступенек» – уровней, причем переход с одного уровня на другой происходит за очень короткие промежутки времени (мгновенно), например, график работы некоторого исполнительного механизма может быть представлен двумя уровнями – включен или выключен.

Аналоговая форма сигнала обладает очень большой информативностью (в зависимости от измеряемой точности непрерывные функции могут принимать миллиарды различных значений), но очень низкой помехозащищенностью (влияние внешних помех может приводить к небольшим изменениям формы сигнала). Для живой природы эти влияния помех не имеют значения, но для технических устройств они недопустимы.

Поэтому, в технических устройствах, например, в компьютерах, применяются только дискретные формы представления информации, среди которых (из-за высокой помехозащищенности) наибольшее распространение получила двоичная форма.

В двоичной форме информация передается с помощью двух уровней сигнала – нуля и единицы (есть сигнал, нет сигнала). Считается, что двоичная форма представления сигнала обладает наивысшей помехозащищенностью.

Единицей информации в компьютерах является бит (один разряд), в который можно записать 0 или 1. Восемь битов образуют байт информации. Байт – это минимально адресуемая единица информации во многих компьютерах. Следующей единицей информации является слово. Для 32-х разрядных компьютеров слово содержит 32 бита или 4 байта, а для 64-х разрядных компьютеров слово содержит 64 бита или 8 байт.

Обработка информации в компьютерах среди прочих операций предполагает и выполнение некоторых вычислений. Поэтому возникла необходимость в использовании системы счисления, «понимаемой» компьютером. Для этих целей идеально подошла двоичная система счисления, цифры которой могут принимать только два значения 0 и 1.

При обработке чисел в памяти компьютера обычно используется позиционная форма представления – число представляется в виде последовательности цифр (0 или 1), позиция каждой из которых имеет свой вес кратный основанию системы счисления. Например, двоичному числу 1001 соответствует запись $1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$ в позиционной форме представления. Если выполнить вычисления, то получим десятичное число 9.

Позиционная форма записи чисел применяется и для вещественных чисел, в дробной части которых порядок имеет отрицательное значение. Например, десятичное число 45,36 в позиционной форме записи будет представлено как $4*10^1 + 5*10^0 + 3*10^{-1} + 6*10^{-2}$. Естественно вещественное число может быть представлено и в двоичной системе счисления.

Таким образом, двоичная система счисления позволяет представлять числа, в форме «понятной» компьютеру.

Представление двоичных чисел на экране монитора требует много места, поэтому принято использовать системы счисления кратные двоичной – восьмеричную или шестнадцатеричную.

Для записи чисел в восьмеричной системе счисления требуется 8 цифр от 0 до 7. Поскольку последовательность из трех двоичных цифр имеет восемь различных комбинаций, то каждое такое сочетание двоичных цифр (триады) можно однозначно представить с помощью одной восьмеричной цифры:

000 – 0
001 – 1
010 – 2
011 – 3
100 – 4
101 – 5
110 – 6
111 – 7

Например, двоичное число 01011110 для удобства записываем с помощью триад 01 011 110 и представляем в восьмеричной системе счисления 136, которое равно $1*8^2+ 3*8^1 + 6*8^0= 94$ в десятичной системе.

Для записи чисел в шестнадцатеричной системе требуется 16 цифр от 0 до 9 и латинские буквы A, B, C, D, E, F. Поскольку последовательность из четырех двоичных цифр имеет 16 различных комбинаций, то каждое такое сочетание двоичных цифр (тетрада) можно однозначно представить с помощью одной шестнадцатеричной цифры:

0000 – 0	1000 – 8
0001 – 1	1001 – 9
0010 – 2	1010 – A
0011 – 3	1011 – B
0100 – 4	1100 – C
0101 – 5	1101 – D
0110 – 6	1110 – E
0111 – 7	1111 – F

Например, тоже двоичное число 01011110 для удобства записываем с помощью тетрад 0101 1110 и представляем в шестнадцатеричной системе счисления 5E, которое равно $5*16^1+ 14*16^0= 94$ в десятичной системе.

При работе с компьютерной техникой системные программисты, как правило, используют шестнадцатеричную систему счисления, так как 1 байт информации легко представляется двумя шестнадцатеричными цифрами.

В языке C# нет формы представления чисел в двоичной или восьмеричной системе счисления, но широко представлена шестнадцатеричная система счисления.

Реализуем программное преобразование чисел десятичной системы счисления от 0 до 127 в шестнадцатеричную систему счисления.

Исходный код программы:

```
usingSystem;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main(string[] args)
{
uint i;
Console.WriteLine("Таблица представлений чисел в 10-ой и 16-ой
системах счисления:");
for (i = 0; i <= 15; i++)
Console.WriteLine("{0,2} - {1:X}  {2} - {3:X}  {4} - {5:X}
{6} - {7:X}  {8} - {9:X}  {10} - {11:X}  {12,3} - {13:X}
      {14,3} - {15:X}",i,i,i+16,i+16,i+32,i+32,i+48,i+48,i+64,
     i+64,i+80,i+80,i+96,i+96,i+112,i+112);
Console.ReadLine();
  }
 }
```

```
}
```

Работа программы:

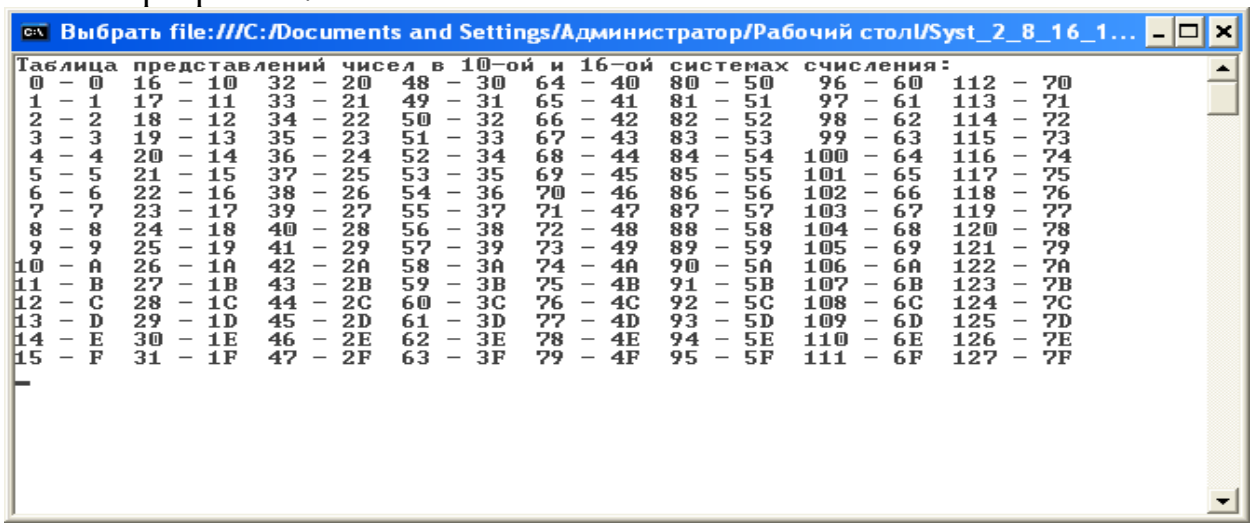


Рисунок 2.1.1 – Преобразование десятичных чисел в шестнадцатеричные числа

Если необходимо получить представление двоичных чисел в «чистом» виде, то обычно для этого применяется строковая форма представления чисел, которая «наполняется» программно, например:

```
using System;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main(string[] args)
  {
uint i;
string d;
char[] b = newchar[8] {'0','0','0','0','0','0','0','0'};
Console.WriteLine("Таблица представлений чисел в 10-ой, 2-ой и
16-ой системах счисления:");
for (i = 0; i <= 63; i++)
   {
    d = "";
for (int j = 0; j < 8; j++) d = d + b[j];
Console.WriteLine("{0,2} - {1} - {2:X}  ", i, d, i);
if (b[7] == '0') b[7] = '1';
else {b[7] = '0'; if (b[6] == '0') b[6] = '1';
else {b[6] = '0'; if (b[5] == '0') b[5] = '1';
else {b[5] = '0'; if (b[4] == '0') b[4] = '1';
else {b[4] = '0'; if (b[3] == '0') b[3] = '1';
else {b[3] = '0'; if (b[2] == '0') b[2] = '1';
else {b[2] = '0'; if (b[1] == '0') b[1] = '1';
else {b[1] = '0'; if (b[0] == '0') b[0] = '1';
else { b[0] = '0'; }}}}}}}}
Console.ReadLine();
```

```
}
  }
}
```

Таблица представлений чисел в 10-ой, 2-ой и 16-ой системах счисления:

0 - 00000000 – 0	17 - 00010001 – 11	34 - 00100010 – 22	51 - 00110011 - 33
1 - 00000001 – 1	18 - 00010010 – 12	35 - 00100011 – 23	52 - 00110100 - 34
2 - 00000010 – 2	19 - 00010011 – 13	36 - 00100100 – 24	53 - 00110101 - 35
3 - 00000011 – 3	20 - 00010100 – 14	37 - 00100101 – 25	54 - 00110110 - 36
4 - 00000100 – 4	21 - 00010101 – 15	38 - 00100110 – 26	55 - 00110111 - 37
5 - 00000101 – 5	22 - 00010110 – 16	39 - 00100111 – 27	56 - 00111000 - 38
6 - 00000110 – 6	23 - 00010111 – 17	40 - 00101000 – 28	57 - 00111001 - 39
7 - 00000111 – 7	24 - 00011000 – 18	41 - 00101001 – 29	58 - 00111010 - 3A
8 - 00001000 – 8	25 - 00011001 – 19	42 - 00101010 - 2A	59 - 00111011 - 3B
9 – 00001001 – 9	26 - 00011010 - 1A	43 - 00101011 - 2B	60 - 00111100 - 3C
10 - 00001010 – A	27 - 00011011 - 1B	44 - 00101100 - 2C	61 - 00111101 - 3D
11 - 00001011 – B	28 - 00011100 - 1C	45 - 00101101 - 2D	62 - 00111110 - 3E
12 - 00001100 – C	29 - 00011101 - 1D	46 - 00101110 - 2E	63 - 00111111 - 3F
13 - 00001101 – D	30 - 00011110 - 1E	47 - 00101111 - 2F	
14 - 00001110 – E	31 - 00011111 - 1F	48 - 00110000 - 30	
15 - 00001111 – F	32 - 00100000 – 20	49 - 00110001 - 31	
16 - 00010000 – 10	33 - 00100001 – 21	50 - 00110010 - 32	

Реально программа выводит все значения таблицы в один длинный столбец, но с целью экономии места в лекции таблица представлена 4 столбцами.

2.1.2 Поразрядные логические операции в языке C#

Поразрядные логические операции представлены операцией поразрядного логического умножения или конъюнкции, поразрядного логического сложения или дизъюнкции и поразрядного логического исключения или исключающего ИЛИ. К логическим операциям также относится операция поразрядного логического отрицания или дополнение.

При выполнении операции поразрядного логического умножения (обозначается &) над двумя числами осуществляется их побитное «логическое умножение» и результат записывается в соответствующий бит. Естественно, в соответствии с правилом логического умножения, результат равен 1, если равны 1 оба участвующих в операции бита.

При выполнении операции поразрядного логического сложения (обозначается |) над двумя числами осуществляется их побитное «логическое сложение» и результат записывается в соответствующий бит. Естественно, в соответствии с правилом логического сложения, результат равен 1, если равен 1 хотя бы один из участвующих в операции битов.

При поразрядном исключении, исключающем ИЛИ (операция обозначается ^) над двумя числами осуществляется их побитное сравнение и результат равен 1, если 1 равен только один любой из участвующих в операции битов.

При выполнении операции поразрядного логического отрицания или дополнения (обозначается ~), во всех разрядах числа, участвующего в операции, единицы заменяются нулями, а нули заменяются единицами.

Для закрепления материала рассмотрим работу учебной программы, в которой используем все перечисленные поразрядные логические операции.

```
using System;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main()
  {
uint a, b, c;
Console.Write("Введите целое значение a ");
a = Convert.ToUInt32(Console.ReadLine());
Console.Write("Введите целое значение b ");
b = Convert.ToUInt32(Console.ReadLine());
   c = a & b;
Console.WriteLine("a  &  b = {0}   & {1}  = {2}", a, b, c);
Console.WriteLine("ax & bx = {0:X} & {1:X}= {2:X}", a, b, c);
   c = a | b;
Console.WriteLine("a  |  b = {0}   | {1}  = {2}", a, b, c);
Console.WriteLine("ax | bx = {0:X} | {1:X}= {2:X}", a, b, c);
   c = a ^ b;
Console.WriteLine("a  ^  b = {0}   ^ {1}  = {2}", a, b, c);
Console.WriteLine("ax ^ bx = {0:X} ^ {1:X}= {2:X}", a, b, c);
Console.WriteLine("~a  = {0}    ~b  = {1}", ~a, ~b);
Console.WriteLine("~ax = {0:X}  ~bx = {1:X}", ~a, ~b);
Console.ReadKey();
  }
 }
}
```

Работа программы:

Введите целое значение a 42
Введите целое значение b 23
a&b = 42 & 23 = 2
ax&bx = 2A & 17 = 2
a | b = 42 | 23 = 63
ax | bx = 2A | 17 = 3F
a ^ b = 42 ^ 23 = 61
ax ^ bx = 2A ^ 17 = 3D
~a = 4294967253 ~b = 4294967272
~ax = FFFFFFD5 ~bx = FFFFFFE8

Проверим работу операции поразрядного логического умножения, представив значение чисел аи b в двоичной системе счисления:

ax = 2A = 0010 1010 – значения взяты из результатов работы программы
bx = 17 = 0001 0111 предыдущего пункта лекции

ax&bx = 0000 0010 – результат равен десятичному (и шестнадцатеричному) числу 2.

2.1.3 Операции побитового сдвига в языке C#

Операции побитового сдвига чисел часто применяются в различных алгоритмах, например, в алгоритмах преобразование последовательности битов в параллельный код и наоборот параллельного кода в последовательность битов. Некоторые устройства компьютера реализуют этот алгоритм аппаратно с помощью специальных сдвиговых регистров, например, схемы записи и чтения информации с магнитных или оптических дисков. В некоторых ситуациях этот алгоритм реализуется программно, например, при программной реализации некоторого протокола передачи данных через разъем USB.

Арифметическая операция умножения на 2 двоичного числа эквивалентна сдвигу двоичного числа на один разряд влево, а операция деления на 2 – сдвигу двоичного числа на один разряд вправо.

Для программной реализации операций сдвига в языке C# имеется две операции:

–операция сдвига двоичного числа влево (обозначается <<);

– операция сдвига двоичного числа вправо (обозначается >>).

Форматы записи обеих операций похожи и требуют до знака операции указывать число, в котором будет осуществляться сдвиг, а после операции задавать количество разрядов, на которое будет осуществляться сдвиг, например:

```
c = a<< 2;          // сдвиг числа a на два разряда влево.
c = b>> 1;          // сдвиг числа b на один разряда вправо.
```

Операции сдвига предназначены для работы только с целочисленными типами значений, например, int, uint и т.д.

При сдвиге влево освободившиеся разряды обнуляются (задвигается 0).

При сдвиге вправо освободившиеся разряды обнуляются, но сдвиг осуществляется с сохранением знака числа.

В качестве учебного примера рассмотрим сдвиг шестнадцатеричного числа 5 (двоичное число 0000 0000 0101) на 8 разрядов влево – должно получиться двоичное число 0101 0000 0000 или шестнадцатеричное 500.

Исходный код программы:

```
usingSystem;
namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main()
  {
uint a;
Console.Write("Введитецелоезначение a ");
   a = Convert.ToUInt32(Console.ReadLine());
```

```
for (int i = 1; i <= 8; i++)
    {
     a = a << 1;
Console.WriteLine("ax = {0:X}       {1}", a, a);
    }
for (int i = 4; i > 0; i--)
    {
     a = a >> 2;
Console.WriteLine("ax = {0:X}       {1}", a, a);
}
Console.ReadKey();
  }}
}
```

Работа программы:

Введите целое значение a 5
ax = A 10
ax = 14 20
ax = 28 40
ax = 50 80
ax = A0 160
ax = 140 320
ax = 280 640
ax = 500 1280
ax = 140 320
ax = 50 80
ax = 14 20
ax = 5 5

Сдвиг влево числа a осуществляется в цикле с шагом сдвига равном 1. Результаты работы цикла показывают, что число a после каждого сдвига удваивалось от 5 до 1280.

Сдвиг вправо числа a также осуществлялся в цикле, но шаг сдвига был задан 2. Результаты работы этого цикла показывают, что число a после каждого сдвига уменьшалось в 4 раза.

2.2 Небезопасное программирование в языке C#

2.2.1 Обозначение идентификаторов в системе Windows

Все системные программисты, работающие с функциями системы Windows должны понимать обозначение идентификаторов, принятые в системе (которые называют венгерскими записями). Эти обозначения начал применять один из лучших программистов Microsoft Чарльз Симонии (родом из Венгрии), когда создавалась система Windows. В шутку эти обозначения стали называть венгерскими записями и всем последователям и пользователям (разрабатывающим программы в Windows) приходится знакомиться с особенностями принятых в Windows обозначений. Идея заключается в том, что для имен переменных, типов и т.д. используются префиксы, описывающие

их тип и характер содержащейся информации. Некоторые префиксы представлены следующим списком:

Префикс	Значение
a	– Массив;
в	– Логический тип (BOOL);
by	– Беззнаковый символьный тип (BYTE);
c	– Символьный тип;
cb	– Счетчик байтов;
cr	– Цвет;
cx,cy	– Короткий тип (SHORT);
dw	– Беззнаковый длинный тип (DWORD);
fn	– Функция;
h	– Логический номер (HANDLE или HINSTANCE);
i	– Целое;
m_	– Переменная класса;
n	– SHORTили int;
np	– Ближний указатель;
p	– Указатель;
l	– Длинный тип(LONG);
lp	– Дальний указатель;
s	– Строка;
cz	– Строка, заканчивающаяся нуль-символом;
w	– Беззнаковое целое(WORD);
x,y	– Короткий тип(координата x или y).

Например, запись lpfnWndProc, означает, что lpfnWndProcявляется дальним указателем на функцию WndProc.

В приведенных примерах несколько раз упоминалось понятие указателя. Технология программирования с использованием указателей очень широко использовалась при написании системы Windows. Поэтому нам необходимо изучить понятие указателя и правил работы с ним. Это особенно важно, так как многие формальные параметров API функций системы Windows, которые мы будем использовать, являются указателями.

«Типы указателей полезны, в основном, при взаимодействии с API-интерфейсами языка C, но их можно применять и для обращения к памяти, находящейся за пределами управляемой кучи, или при реализации высокопроизводительных участков программы». [Джозеф Албахари стр.171.C# 3.0 справочник]

2.2.2 Понятие указателя

Указатели – это переменные, которые содержат в качестве своих значений адреса памяти компьютера.

Здесь следует напомнить определение переменной. Переменная это область памяти, обозначенная идентификатором (именем переменной), в

которой могут храниться изменяемые в процессе работы программы данные. Имя переменной должно содержать информацию об адресе памяти, где находятся данные этой переменной (иначе мы не найдем свои данные в памяти компьютера). Но если физический адрес памяти компьютера, где находятся значения обычной переменной, недоступен пользователю, то переменная указатель специально предназначена для работы с физическими адресами памяти компьютера и в том числе с адресами памяти, где находятся данные обычных переменных.

Т. е. если есть переменная A, то можно представить, что есть переменная Aptr, в которой находится адрес памяти с данными переменной A. Соотношение между переменной A и указателем-переменной Aptr можно представить следующим рисунком:

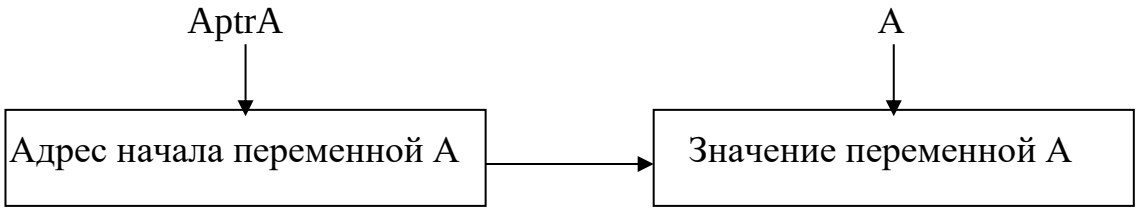


Рисунок 2.2.1 – Переменная A и ее указатель Aptr.

Указатели, подобно любым другим переменным, перед своим использованием в программе должны быть объявлены. Формат объявления указателя имеет следующий вид:

тип* переменная; где

тип – это тип значения в области памяти, адрес которой будет храниться в переменной. Тип не может быть классом, но может быть структурой, перечислением или указателем, а также любым из значимых типов и void. Тип void означает, что указатель будет хранить адрес переменной неизвестного типа. Примеры объявления указателей:

int* aptr;

inta;

в этом примере мы объявляем две переменные: a – обычная переменная целого типа; aptr –указатель на значение целого типа.

float* xptr, yptr;

в этом случае переменные xptr и yptr являются указателями на значения вещественного типа.

int*[] mas;

в примере рассмотрен вариант объявления массива указателей на значения целого типа.

int** iptr;

в примере объявлен указатель на указатель целого типа.

Символ «*» в объявлении переменных показывает, что соответствующая ей переменная является указателем (т.е. предназначена для хранения адреса). Еще раз напомним, что значение указателя – это адрес памяти компьютера.

2.2.3 Операция инициализации указателя

Указатель это переменная, поэтому с ней можно выполнять некоторые операции.

Главной операцией любой переменной является операция инициализации (присваивания) – переменной присваивается некоторое значение. Указатель не исключение, но в качестве значения необходимо иметь адрес памяти компьютера. Для получения физического адреса памяти компьютера в языке C# имеется специальная операция – операция адресации «&», которая возвращает адрес памяти переменной или других данных, перед которыми эта операция установлена. Например,

Aptr = &a; .

В этом примере указателю Aptr будет присвоен адрес переменной a – начальный адрес области памяти компьютера, где хранятся данные переменной a. При этом Aptr должна быть объявлена как указатель переменной того же типа, что и переменная a.

Значению указателя можно присвоить значение другого указателя, но только соответствующего типа, например, если Aptr и Bptr указатели на значения целого типа и значение Bptr уже определено (иначе в присваивании нет смысла), то допустима следующая запись:

Aptr = Bptr; .

Объявление переменных в языке C# должно сопровождаться их инициализацией, поэтому для указателей в язык C# была введена константа фиктивного «нулевого» адреса памяти компьютера – null.Значение указателя любого типа может быть присвоено этому «нулевому» адресу памяти компьютера, как при инициализации, так и в процессе работы программы, например:

```
int* Aptr = null;
float Fptr;
Fptr = null;
```

В этом примере указателям Aptr и Fptr присвоен фиктивный «нулевой» адреса памяти компьютера – null.

В процессе работы программы (или при инициализации указателей) указателям можно присваивать адрес памяти компьютера в явном виде, но только в том случае если этот адрес Вам точно известен, иначе может произойти сбой программы, например:

```
byte* Aptr = (byte *) 0x3F0ECD4;
char* Cptr = (char *) 0x3F0ECD4;
```

где, 0x3F0ECD4 – шестнадцатеричная константа адреса памяти, (byte *) или (char *) – операции приведения типов, в которых константа приводится к соответствующему типу указателя.

Примеры инициализации массива указателей и выделение памяти под стек с помощью указателей рассмотрим на конкретных примерах позже.

2.2.4 Другие операции с указателями

В языкеC# существует операция разыменование, обозначаемая символом «*». Эта операция обратная операции адресации. Если операция адресации по данным получает их адрес, то операция разыменования по адресу получает данные. Естественно в этой операции должен участвовать указатель. Например,

a=*Aptr;

переменной априсваиваются данные, адрес которых находятся в указателе Aptr. При этом Aptr должна быть объявлена как указатель переменной того же типа, что и переменная a.

При работе с данными типа структура в языке C# имеется специальная операция «указатель на элемент» или операция доступа к элементу структуры через указатель, которая обозначается комбинацией символов «->». Например, если PtrCtyd обозначает указатель на некоторую структуру, а Name является полем этой структуры, то запись PtrCtyd->Name эквивалентна (*PtrCtyd).Name. Т.е. по адресу *PtrCtyd мы обращаемся ко всей структуре и в ней выбираем поле Name.

Основные операции с указателями представлены в таблице 2.2.1.

Таблица 2.2.1 Основные операции с указателями

Операция	Описание
*	Разыменования – получение значения, которое находится по адресу, хранящемуся в указателе
->	Доступ к элементу структуры через указатель
[]	Доступ к элементу массива через указатель
&	Получение адреса переменной
++, --	Увеличение и уменьшение значения указателя на один адресуемый элемент
+, -	Сложение с целой величиной и вычитание указателей
==, !=, <>, <=, >=	Сравнение адресов, хранящихся в указателях. Выполняется как сравнение беззнаковых целых величин
Stackalloc	Выделение памяти в стеке под переменную, на которую ссылается указатель

2.2.5 Понятие небезопасного кода

Одним из основных достоинств языка C# является его схема работы с памятью: автоматическое выделение памяти под объекты и автоматическая уборка мусора. При этом невозможно обратиться по несуществующему адресу памяти или выйти за границы массива, что делает программы более надежными и безопасными и исключает возможность появления целого класса ошибок.

Указатели же позволяют работать напрямую с адресами областей памяти, что входит в противоречие с принципами языка C#. Поэтому программный код, использующий указатели, стали называть небезопасным.

В общем случае небезопасным называется код, выполнение которого среда CLR не контролирует.

Для разрешения конфликта кода со средой CLR такой код должен быть явным образом помечен с помощью ключевого слова unsafe, которое определяет так называемый небезопасный контекст выполнения.

Ключевое слово unsafe может использоваться как спецификатор при объявлении класса, структуры или поля данных структуры, например,

Public unsafe struct Ctyd {}.

В этом случае все поля структуры помечаются как небезопасные. Если ключевое слово unsafeиспользуется как спецификатор для объявления только одного поля структуры, например,

Public struct Ctyd {… public unsafe int * Kol; … }

то условие небезопасности распространяется только на отмеченное поле этой структуры.

Ключевое слово unsafe может использоваться как сложный оператор, например: unsafe{ блок }, при этом все операторы, входящие в указанный блок будут являться небезопасными – т.е. их работа не будет контролироваться средой CLR и, что особенно важно, динамическая память, выделяемая под небезопасные переменные не будет «зачищаться» сборщиком мусора языка C#.

2.2.6 Пример программы работы с указателями

Для лучшего понимания правил работы с указателями и выполнения операций над ними рассмотрим чисто учебный пример, в котором рассмотрим работу операций адресации и разыменования.

Для успешной компиляции кода консольного приложения, содержащего небезопасный фрагмент, необходимо изменить настройки среды Visual Studio следующим образом: выбрать команды Project->ConsoleAplication1 Properties->Buildи в открывшемся окне установить «птичку» в пункте Allow Unsafe Code. Работу с окном закончить через кнопку Advanced, подтверждая изменения нажатием кнопки OK.

В коде приложения ключевое слово unsafe можно использовать как спецификатор метода Main, например, `staticunsafevoid Main()`, тогда все операторы этого метода будут рассматриваться как небезопасные.

Исходный код программы (отдельные фрагменты кода взяты из книги Павловской):

```
using System;

namespace ConsoleApplication1
{
classProgram
```

```
 {
staticunsafevoid Main()
  {
int a, c;
uint adr;
   a = 10;
int* aPtr = null;
   aPtr = &a;
Console.WriteLine("*aPtr = " + *aPtr);
Console.WriteLine("a = " + a);
Console.Write("aPtr = ");
   adr = (uint)aPtr;
byte* b = (byte*)&adr;
   b = b + 3;
Console.Write("0x");
for (int i = 0; i < 4; i++)
       { Console.Write("{0:X}", *b); b--; }
Console.WriteLine();
Console.Write("&a   = ");
   adr = (uint)&a;
   b = (byte*)&adr;
   b = b + 3;
Console.Write("0x");
for (int i = 0; i < 4; i++)
       { Console.Write("{0:X}", *b); b--; }
Console.WriteLine();
Console.WriteLine("adr = " + adr);
   c = *& a;
Console.WriteLine("*& a = " + c);
//c = &* a;
//Console.WriteLine("&* a = " + c);
Console.Write("&* aPtr = ");
   adr = (uint)&* aPtr;
   b = (byte*)&adr;
   b = b + 3;
Console.Write("0x");
for (int i = 0; i < 4; i++)
       { Console.Write("{0:X}", *b); b--; }
Console.WriteLine();
Console.Write("*& aPtr = ");
   adr = (uint)*& aPtr;
   b = (byte*)&adr;
   b = b + 3;
Console.Write("0x");
for (int i = 0; i < 4; i++)
       { Console.Write("{0:X}", *b); b--; }
Console.WriteLine();
Console.ReadKey();
  }
 }
}
```

Работа программы:

```
*aPtr = 10
a = 10
aPtr = 0x3F0ECD4
&a   = 0x3F0ECD4
adr = 66120916
*&a = 10
&* aPtr = 0x3F0ECD4
*&aPtr = 0x3F0ECD4
```

Две подряд операции разыменования и адресации взаимно исключают друг друга, как для указателя, так и для переменной целого типа (смотри *&a = 10).

Две подряд операции адресации и разыменования взаимно исключают друг друга только для указателя. Попытка использовать эту комбинацию для переменной целого типа приводит к ошибке компиляции программы со следующим сообщением: Error1The * or ->operator mustbe appliedtoapointer.

При работе программы с указателями, хорошим стилем программирования считается, если указатели в начале программы инициализируются, т.е. либо им присваивается некоторые значения, либо указателям присваиваются нулевые значения равные null.

2.2.7 Использование указателя для типа void*

При объявлении указателя можно использовать тип void*, что означает объявление указателя с отложенным типом.

Для указателя такого типаvoid* нельзя применять арифметические операции и операцию разыменования.

Однако он может быть использован для неявного преобразования указателя типа void* в указатель любого типа. В тоже время указателю типа void* можно присвоить адрес переменной любого типа. Таким образом, указатель типа void* можно использовать как промежуточное хранилище адресов переменных любого типа. Рассмотрим чисто учебный пример, в котором указателю типа void* поочередно будем присваивать адреса переменных целого и вещественного типов, а затем передавать эти адреса указателям, соответственно для целого и вещественного типа.

Исходный код программы:

```
using System;

namespace ConsoleApplication1
{
classProgram
 {
staticunsafevoid Main()
  {
int a = 10;
double b = 15.67;
int* aPtr = null;
```

```
double* bPtr = null;
void* cPtr=null;
for (int i = 1; i <= 5; i++)
   {
if (i % 2 == 0)
    {
     cPtr = &a; aPtr = (int*)cPtr;
Console.WriteLine("i = {0} По адресу указателя находится число
{1} ", i, *aPtr);
}
else
    {
     cPtr = &b; bPtr = (double*)cPtr;
Console.WriteLine("i = {0} По адресу указателя находится число
{1} ", i, *bPtr);
    }
   }
Console.ReadKey();
  }
 }
}
```

Работа программы:

i = 1 По адресу указателя находится число 15,67
i = 2 По адресу указателя находится число 10
i = 3 По адресу указателя находится число 15,67
i = 4 По адресу указателя находится число 10
i = 5 По адресу указателя находится число 15,67

2.3 Использование указателей при работе с массивами

2.3.1 Использование операций сложения и вычитания для перемещения указателя по массиву данных

К указателям применимы и некоторые арифметические операции, но их использование требует хороших знаний по размещению данных в памяти компьютера, т.к. увеличение указателя на единицу означает переход на адрес следующего данного, к типу которого относится указатель.

Для простоты в качестве данных будем использовать массив переменных типа **char**, в которые при инициализации массива запишем первые 10 букв латинского алфавита. Перемещая указатель по массиву, и печатая отдельные буквы, мы можем напечатать некоторое слово, например, «BEHEFIC».

Исходный код программы:

```
usingSystem;

namespace ConsoleApplication1
{
classProgram
 {
staticunsafevoid Main()
```

```
  {
char[] masc = newchar[]
                {'A','B','C','D','E','F','G','H','I','J'};
fixed (char* Mptr = masc)
   {
char* ptr = Mptr;
    ptr++;
Console.Write(*ptr);
    ptr = ptr +3;
Console.Write(*ptr);
    ptr += 3;
Console.Write(*ptr);
    ptr -= 3;
Console.Write(*ptr);
    ptr++;
Console.Write(*ptr);
    ptr += 3;
Console.Write(*ptr);
    ptr -= 6;
Console.Write(*ptr);
   }
Console.WriteLine();
Console.ReadKey();
  }
 }
}
```

Работа программы:

BEHEFIC

В программе использован оператор fixed, который имеет следующий формат записи:

fixed ([type][*] [имя указателя] = [&][имя объекта])

{ действия с фиксированным объектом }

«Оператор **fixed** фиксирует объект, адрес которого заносится в указатель, для того чтобы его не перемещал сборщик мусора, и таким образом указатель остался корректным. Фиксация происходит на время выполнения блока, который записан после круглых скобок.» [Павловская стр.352].

Дело в том, что сборщик мусора работает постоянно и занимается оптимизацией памяти в куче – удалением объектов, закончивших работу, и перемещением работающих объектов с целью оптимизации памяти кучи.

Если наш объект, в результате такой оптимизации, переместится в другую область памяти, то изменятся его адреса и применение указателей со старыми адресами теряет всякий смысл. Фактически оператор **fixed** предупреждает сборщик мусора, что указанный объект нельзя перемещать в куче.

Зафиксировав начальный адрес массива символов с помощью указателя **Mptr**, мы как бы создаем заголовок списка элементов массива. Для перемещения по списку элементов массива необходим еще один указатель –

текущий указатель **ptr**, который при инициализации устанавливается на заголовок. Попытки изменить заголовок внутри блока оператора **fixed** приводит к ошибке компиляции программы, так как изменение заголовка зафиксированной области данных могло бы приводить к потере данных.

Использование арифметических операции над указателями допускает только действия над целыми значениями, которые соответствуют адресам данных в памяти компьютера. При этом размер типа данных, адресуемых указателями, настраивается автоматически. Рекомендуется использовать арифметические операции при работе с массивами, значения элементов которых размещаются в памяти компьютера последовательно.

2.3.2 Использование операции **stackalloc** для размещения данных в памяти

В языке C# выделение памяти под данные указателей с помощью операции **new** не используется. Основным способом выделения памяти под данные, адресуемые с помощью указателей, является операция **stackalloc**, при этом память выделяется в стеке.

Формат записи операции stackalloc:

тип* имя указателя = **stackalloc** тип [количество];

В качестве учебной программы рассмотрим выделение памяти для 10 переменных целого типа и их сортировку методом выбора.

```
using System;

namespace ConsoleApplication1
{
classProgram
 {
staticunsafevoid Main()
  {
int i, j, b;
int* masptr = stackallocint[10];
   // Проверка инициализации массива
for (i = 0; i < 10; i++)
   {
Console.Write(" {0}", masptr[i]);
   }
Console.WriteLine();
Random rnd = newRandom();
// формирование и печать массива
Console.Write("Массив до сортировки:    ");
for (i = 0; i < 10; i++)
   {
    masptr[i] = rnd.Next() % 101 - 50;
Console.Write(" {0}", masptr[i]);
}
Console.WriteLine();
// сортировка элементов массива методом выбора
for (i = 0; i < 9; i++)
```

```
for (j = i + 1; j < 10; j++)
if (masptr[i] < masptr[j])
      { b = masptr[i]; masptr[i] = masptr[j]; masptr[j] = b; }
// печать массива после сортировки
Console.Write("Массив после сортировки: ");
for (i = 0; i < 10; i++)
Console.Write(" {0}", masptr[i]);
Console.WriteLine();
Console.ReadKey();
  }
 }
}
```

Работа программы:

0 0 0 0 0 0 0 0 0 0

Массив до сортировки: 23 -27 -35 33 -31 -31 23 4 42 40

Массив после сортировки: 42 40 33 23 23 4 -27 -31 -31 -35

Если не знать, что для выделения памяти использован указатель, то можно предположить, что в программе использован обычный массив со странным именем `masptr`.

При обращении к массиву через указатель среда не контролирует нарушение границ массива и есть опасность, чрезмерно изменяя индекс, «выйти» за границу памяти, выделенную под массив и нарушить некоторые данные или программу. Вторым неудобством при обращении к массиву с помощью указателя является невозможность определения длины массива с помощью свойства Length, которое не работает с указателями.

При выделении памяти в стеке всем элементам массива присваиваются нулевые значения – выполняется начальная инициализация данных.

С помощью операции **stackalloc** можно выделять память под отдельные переменные – как массивы с одним элементом. В следующем фрагменте программы показано использование двух указателей, память для которых выделялась в стеке с помощью операции **stackalloc**.

```
int* Aptr = stackallocint[1];
int* Bptr = stackallocint[1];
. . .
Aptr[0] = 5;
Bptr[0] = Aptr[0];
Console.WriteLine("Bptr[0] =" + Bptr[0]);
```

При работе фрагмента будет напечатано `Bptr[0] = 5`

Освобождение памяти в стеке осуществляется автоматически при завершении работы блока, в котором находится операция **stackalloc** – в нашем случае это программа.

2.3.3 Использование массива указателей

Указатели, как и обычные переменные, могут храниться в массива. Объявление массива указателей имеет следующий формат:

тип* [] имя_массива_ указателей = newтип*[кол-во_элементов];

Например, int* [] masiPtr = new int*[10];

В качестве учебного примера использования массива указателей, рассмотрим формирование 10 случайных чисел в диапазоне минус 50 до 50, запись их в память, а их адресов в массив указателей.

Исходный код программы:

```
usingSystem;

namespace ConsoleApplication1
{
classProgram
 {
staticunsafevoid Main()
  {
int i, j, b;
int*[] masiPtr = newint*[10];
Randomrnd = newRandom();
// формирование массива
for (i = 0; i < 10; i++)
   {
int* aPtr = stackallocint[1];
    aPtr[0] = rnd.Next() % 101 - 50;
    masiPtr[i] = &aPtr[0];
}
// печатьмассива
Console.Write("Массив :    ");
for (i = 0; i < 10; i++)
Console.Write(" {0}", *masiPtr[i]);
Console.WriteLine();
Console.ReadKey();
  }
 }
}
```

Работа программы:

```
Массив :    -8 -43 31 19 23 31 -36 -46 -2 -21
```

Выделение памяти для каждого значения элемента массива осуществляется с помощью операции `int* aPtr = stackallocint[1];`, а адреса памяти значений элементов массива записываются в массив указателей.

2.3.4 Использование указателей при работе со структурами

В качестве учебного примера работы со структурами с помощью указателей создадим стек записей, полями которых будут некоторая

переменная целого типа и указатель на следующий элемент стека (указатель на аналогичную запись).

```
using System;

namespace ConsoleApplication1
{
classProgram
 {
unsafestructzveno
  {
int nom;
zveno* next;
staticunsafevoid Main()
{
zveno zap = newzveno();
zveno* tek1, tek2;
zveno* zagol;
    zagol = &zap;
    tek2 = zagol;
zagol->nom = 10;
    zagol->next = null;
for (int i = 1; i < 5; i++)
    {
zveno* zap1 = stackalloczveno[1];
     tek1 = &zap1[0];
     tek2->next = tek1;
     tek1->next = null;
     tek1->nom = i;
     tek2 = tek1;
    }
    tek1 = zagol;
while (tek1 != null)
{
Console.Write("  " + tek1->nom);
     tek1 = tek1->next;
    }
Console.WriteLine();
Console.ReadKey();
   }
  }
 }
}
```

Работа программы:

```
 10 1  2  3  4
```

Тема 3 РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ В ЯЗЫКЕ C#

3.1 Понятие регулярных выражений

В любом языке программирования существуют специальные средства для работы с текстовой информацией. Обычно это переменные типа string, char и набор операторов, функций или классов для работы с этими переменными.

Регулярные выражения представляют собой специализированный язык программирования, предназначенный для обработки текстовой информации.

Необходимо отметить, что регулярные выражения библиотеки .NET основаны на регулярных выражениях языка Perl5 и включают большую часть его функциональных возможностей.

Естественно любая задача, решаемая регулярными выражениями, может быть реализована с помощью методов классов System.String и System.Text.StringBuilder, но объемы кода этих решений будут в разы больше объема кода, записанного с помощью регулярных выражений.

Необходимо также отметить, что некоторые элементы регулярных выражений успешно используются системными программистами при работе с операционными системами в режиме «командной строки».

3.2 Основные задачи, решаемые регулярными выражениями

Регулярные выражения библиотеки .NET обеспечивают решение следующих основных задач при работе с текстовой информацией:

проверку наличия в тексте заданного фрагмента;

проверку допустимости ввода текста, например, паролей;

поиск в тексте по заданному шаблону;

редактирование, замену и удаление фрагментов текста;

извлечение данных из файлов любой структур, например, HTML-страницы;

формирование итоговых отчетов по результатам работы с текстом.

Естественно это только часть задач, решаемых с помощью регулярных выражений, некоторые из которых мы будем рассматривать в наших лекциях.

При этом необходимо помнить, что строки в языке C# являются неизменяемыми объектами, так что регулярное выражение не способно изменить исходную строку.

3.3 Символика языка регулярных выражений

Символика языка регулярных выражений представлена символами двух видов: обычными и метасимволами.

Обычные символы представляют в тексте сами себя, например, шаблон «Само*», используемый при поиске в тексте будет выделять слова, которые начинаются символами «Сам». Например, «Сам», «Само» или «Самоо». Символ «*» является метасимволом и обозначает, что предыдущий символ «о» может повторяться 0 или более раз. Таким образом, обычные символы позволяют записывать текстовую информацию.

Метасимволы имеют специальный, а не символьный смысл.

Естественно в языке регулярных выражениях есть «зарезервированные» символы, использование которых воспринимается языком как метасимвол, а не символ. Например, символ «*» в предыдущем примере.

Если в тексте необходимо использовать символ «*», а не метасимвол *, то перед его использованием необходимо установить символ «\», например, 3*a – 3 умножить на a.

Если в качестве метасимвола необходимо использовать букву, то перед ее использованием необходимо установить символ «\», например, запись «Lab\d» означает, что на месте «\d» в тексте может находиться любая десятичная цифра – Lab1 или Lab6.

Метасимволов очень много. Обычно в литературе метасимволы рассматриваются по группам, в зависимости от их функционального назначения. Например, уточняющие метасимволы, повторители или квантификаторы, заменители или «Классы символов» и т.д. Для их детального изучения обычно используется справочная литература, а не лекции. Однако многие студенты «не умеют читать справочную литературу» и преподаватели традиционно вынуждены приводить в лекциях основные метасимволы языка регулярных выражений. Не будем отступать от традиций и рассмотрим некоторые метасимволы в зависимости от их функционального назначения.

3.4 Повторители

Метасимволы повторители (иногда их в литературе называют квантификаторы) располагаются после обычного символа или класса символов и определяют число возможных повторений в тексте.

Рассмотрим наиболее часто используемые повторители, которые приведены в таблице 3.1.[книга Павловской стр.357].

Таблица 3.1 Повторители

Метасимвол	Описание	Пример
*	Ноль или более повторений предыдущего элемента	Выражение ca*t соответствует фрагментам ct, cat, caat, caaaaaaaaaat и т. д.
+	Одно или более повторений предыдущего элемента	Выражение ca+t соответствует фрагментам cat, caat, caaaaaaaaaat и т. д.
?	Ни одного или одно повторение предыдущего элемента	Выражение ca?t соответствует фрагментам ct и cat
{n}	Ровно n повторений предыдущего элемента	Выражение ca{3}t соответствует фрагменту caaat, а выражение (cat){2} — фрагменту catcat
{n,}	По крайней мере n повторений предыдущего элемента	Выражение ca{3,}t соответствует фрагментам caaat, caaaat, caaaaaaaaaaat и т. д.

{n,m}	От n до m повторений предыдущего элемента	Выражение ca{2,4}t соответствует фрагментам caat, caaat и caaaat

Из приведенных примеров видно, что метасимволы «повторители» определяют число возможных повторений символа, после которого установлен метасимвол, в соответствующем месте текста.

3.5 Уточняющие метасимволы

Метасимволы этой группы уточняют место в строке текста, в котором следует искать совпадение с регулярным выражением. Наиболее часто используемые уточняющие метасимволы приведены в таблице 3.2. [стр.356].

Таблица 3.2 Уточняющие метасимволы

Метасимвол	Описание
^	Фрагмент, совпадающий с регулярным выражением, следует искать только в начале строки
$	Фрагмент, совпадающий с регулярным выражением, следует искать только в конце строки
\A	Фрагмент, совпадающий с регулярным выражением, следует искать только в начале многострочной строки
\Z	Фрагмент, совпадающий с регулярным выражением, следует искать только в конце многострочной строки
\b	Фрагмент, совпадающий с регулярным выражением, начинается или заканчивается на границе слова (то есть между символами, соответствующими метасимволам \w и \W)
\B	Фрагмент, совпадающий с регулярным выражением, не должен встречаться на границе слова

Например, выражение ^cat соответствует символам cat, встречающимся в начале строки, выражение cat$ — символам cat, встречающимся в конце строки (то есть за ними идет символ перевода строки), а выражение ^$ представляет собой пустую строку, то есть начало строки, за которым сразу же следует ее конец.

3.6 Заменители или «Классы символов»

Метасимволы этой группы действуют как заменители для конкретной последовательности символов в тексте. Наиболее часто используемые заменители приведены в таблице 3.3. [стр.356].

Таблица 3.3 Заменители или «Классы символов»

Класс символов	Описание	Пример

.	Любой символ, кроме \n	Выражение c.t соответствует фрагментам cat, cut, c1t, c{t и т. д.
[]	Любой одиночный символ из последовательности, записанной внутри скобок. Допускается использование диапазонов символов	Выражение c[au1]t соответствует фрагментам cat, cut и c1t, а выражение c[a-z]t — фрагментам cat, cbt, cct, cdt, ..., czt
[^]	Любой одиночный символ, не входящий в последовательность, записанную внутри скобок. Допускается использование диапазонов символов	Выражение c[^au1]t соответствует фрагментам cbt, c2t, cXt и т. д., а выражение c[^a-zA-Z]t — фрагментам cит, c1t, cЧt, c3t и т. д.
\w	Любой алфавитно-цифровой символ, то есть символ из множества прописных и строчных букв и десятичных цифр	Выражение c\wt соответствует фрагментам cat, cut, c1t, cЮt и т. д., но не соответствует фрагментам c{t, c;t и т. д.
\W	Любой не алфавитно-цифровой символ, то есть символ, не входящий в множество прописных и строчных букв и десятичных цифр	Выражение c\Wt соответствует фрагментам c{t, c;t, c t и т. д., но не соответствует фрагментам cat, cut, c1t, cЮt и т. д.
\s	Любой пробельный символ, например, символ пробела, табуляции (\t, \v), перевода строки (\n, \r), новой страницы (\f)	Выражение \s\w\w\w\s соответствует любому слову из трех букв, окруженному пробельными символами
\S	Любой не пробельный символ, то есть символ, не входящий в множество пробельных	Выражение \s\S\S\s соответствует любым двум непробельным символам, окруженным пробельными
\d	Любая десятичная цифра	Выражение c\dt соответствует фрагментам c1t, c2t, …, c9t
\D	Любой символ, не являющийся десятичной цифрой	Выражение c\Dt не соответствует фрагментам c1t, c2t, …, c9t

В таблице 3.3 описаны наиболее употребительные метасимволы, представляющие собой классы символов.

3.7 Специальные (управляющие) символы

Эти символы мы уже рассматривали при изучении основ программирования на языке C# в первом семестре, но методически правильно будет повторить их в данной лекции. В основном эти символы используются для управления вывода информации на внешнее устройство компьютера, например, монитор. Наиболее часто используемые специальные символы приведены в таблице 3.4.

Таблица 3.4 Некоторые специальные символы регулярных выражений

\a	Звуковой сигнал (предупреждение)
\b	Символ возврата на одну позицию (забой)
\t	Горизонтальная табуляция
\r	Возврат каретки
\v	Вертикальная табуляция
\f	Перевод страницы
\n	Символ новой строки (перевод строки)
\e	Escape-символ

Особый случай: внутри регулярного выражения последовательность \b обозначает границу слова, за исключением множества, заключенного в квадратные скобки [ ], где она обозначает символ забоя.

3.8 Регулярные выражения

Под регулярным выражением мы будем понимать последовательность символов, записанная с помощью языка регулярных выражений. Обычно это некоторый шаблон, который используется для поиска фрагментов в некотором тексте. Например, следующее простое регулярное выражения предназначено для описания целых чисел:

```
[–+]?\d+
```

где указание «[–+]» означает, что на этом месте может находиться «любой одиночный символ из последовательности, записанной внутри скобок», а указание «?» – что этот символ может встречаться «ни один или ровно один раз». Указание «\d» означает, что на этом месте может находиться «любая десятичная цифра», а указание «+», что возможно «одно или более повторений предыдущего элемента», то есть цифры.

Регулярное выражение, предназначенное для описания вещественных чисел, имеет следующий вид:

```
[–+]?\d+\.?\d*
```

в котором добавлены указания, что в вещественном числе может быть символ «.» и цифра дробной части имеющей «ноль или более повторений предыдущего элемента».

Все типы, предназначенные для работы с регулярными выражениями, находятся в пространство имен System.Text.RegularExpressions.

Главным классом этого пространства имен является класс Regex, экземпляр которого представляет регулярное выражение. Также как и класс String класс Regex является неизменяемым, то есть после создания его экземпляра корректировка не допускается.

Основными свойствами класса Regex являются Index и Length, которые содержат начальное значение позиции и длину обнаруженного соответствия в тексте. Найденное значение помещается в свойство Value.

Основными методами выполняющим поиск в тексте являются, IsMatch, Match и Matches.

Метод IsMatch класса Regex выполняет поиск в тексте и возвращает true, если фрагмент, соответствующий регулярному выражению, найден в заданном тексте, и false если его нет.

Метод Match класса Regex, в отличие от метода IsMatch, дополнительно создает объект класса Match первого фрагмент текста, совпавшего с заданным регулярным выражением. По умолчанию поиск в тексте выполняется слева на право.

Метод Matches класса Regex возвращает объект класса MatchCollection коллекцию (массив) всех фрагментов текста, совпавших с заданным регулярным выражением.

Рассмотрим работу этих методов при поиске в некотором тексте.

```
using System;
using System.Text.RegularExpressions;
using System.Text;
namespace ConsoleApplication1
{
 class Program
 {
  static void Main()
  {
   string text    = "Язык регулярных выражение определяет
шаблоны символов.";
   if (Regex.IsMatch(text, ”ред|выр")) Console.WriteLine("Есть
совпадение!");    else Console.WriteLine("Совпадений нет.");
   Console.WriteLine();
   Match Obj = Regex.Match(text, @”ред|выр");
   Console.WriteLine( Obj.Success);
   Console.WriteLine( Obj.Index);
   Console.WriteLine( Obj.Length);
   Console.WriteLine( Obj.Value);
   Console.WriteLine( Obj.ToString());
   //Console.WriteLine(Obj);
   Console.WriteLine();
   MatchCollection Kol = reg.Matches(text);
   Console.WriteLine("Число совпадений = {0}", Kol.Count);
   foreach(Match T in Regex.Matches(text, "ред|выр"))
    Console.WriteLine("{0} - {1}",T.Index, T.Value);
   Console.ReadLine();
  }
 }
```

```
}
```

Работа программы:

```
Есть совпадение!

True
16
3
выр
выр

Число совпадений = 2
16 – выр
28 – ред
```

Некоторые комментарии к программе.

При использовании метода IsMatch класса Regex мы получаем в распоряжение только логический результат поиска true или false.

При использовании метода Match класса Regex создается объект, значения полей которого мы выводим на экран монитора. Отдельной строкой выведено значение всего объекта, преобразованного в строковый вид. Изменил последнюю строку вывода для этого метода:

```
Console.WriteLine(Obj);
```

Сообщений об ошибках нет, и вывод не изменился.

В этом методе перед регулярным выражением установлен символ «@». Это сделано для обхода механизма «экранирования», действующего в языке C# при использовании в управляющих символах, которые начинаются с символа бэкслеша «\». В нашем регулярном выражении нет символов «\» поэтому символ «@» можно не ставить. Без префикса «@» и наличия символа «\» в тексте регулярного выражения каждый символ «\» пришлось бы «экранировать» четырьмя символами обратная косая черта. Например, нам необходимо написать регулярное выражение, содержащее бэкслеш – \sector. Но комбинация «\s» является заменителем, поэтому необходимо экранировать символ «\» и в строке появляется двойной символ «\\sector». Однако в регулярном выражении оба бэкслеша должны быть экранированы снова, то есть "\\\\sector". Одним словом, чтобы сопоставить бэкслеш, нужно писать в качестве строки регулярного выражения '\\\\', потому что регулярное выражение должно быть \\, и каждая обратная косая черта должна быть переведена в обычную строку как \\.

При работе с регулярными выражениями рекомендуется использовать символ «@» – то есть регулярные выражения представлять в виде @-констант.

При использовании метода Matches класса Regex создается массив объектов типа Match, свойства которых можно просматривать с помощью цикла foreach.

В нашем примере регулярные выражения задавались в методах класса Regex в качестве второго параметра после анализируемого текста. Однако возможен вариант создания объекта класса Regex (создание регулярного

выражения) и использование перечисленных методов для этого объекта. В этом случае методы имеют только один формальный параметр – анализируемый текст. Например:

```
Regex reg = new Regex(@"ред|выр");
Match Obj = reg.Match(text);
```

Для описания регулярного выражения в классе Regex определено несколько перегруженных конструкторов:

Regex() – создает пустое регулярное выражение;

Regex(String) – создает заданное регулярное выражение;

Regex(String, RegexOptions) – создает заданное регулярное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (опции).

Необходимо отметить, что класс Regex содержит не только методы поиска в тексте, но множество других методов обработки текстов. Например, метод Regex.Replace выполняет замены фрагментов текста в соответствии с заданным регулярным выражением, а метод Regex.Split задает регулярным выражением разделитель текста.

Каждый из методов класса Regex может использовать уточняющие опции, некоторые из которых приведены в таблице 3.5.

Таблица 3.5 Некоторые опции методов класса Regex

Опция	Описание
CultureInvariant	Предписывает игнорировать национальные установки (культуру) строки.
ExplicitCapture	Модифицирует способ поиска соответствия, обеспечивая только буквальное соответствие.
IgnoreCase	Игнорирует регистр символов во входной строке.
IgnorePattern-Whitespace	Удаляет из строки не защищенные управляющими символами пробелы и разрешает комментарии, начинающиеся со знака фунта или хеша.
Multiline	Изменяет значение символов ^ и $ так, что они применяются к началу и концу каждой строки, а не только к началу и концу всего входного текста.
RightToLeft	Предписывает читать входную строку справа налево вместо направления по умолчанию – слева на право.

Статические методы класса приведены в таблице 3.6.

Таблица 3.6 Статические методы класса Regex

	Escape	Преобразует минимальный набор метасимволов (\, *, +, ?, |, {, [, (,), ^, $,., # и пробел), заменяя их escape-кодами.

	IsMatch	Перегружен. Указывает на то, обнаруживает ли регулярное выражение соответствие во входной строке.
	Match	Перегружен. Ищет во входной строке вхождение регулярного выражения и возвращает точный результат в качестве отдельного объекта Match.
	Matches	Перегружен. Ищет во входной строке все вхождения регулярного выражения и возвращает все успешные соответствия, как если бы Match вызывался несколько раз.
	Replace	Перегружен. В указанной входной строке заменяет строки, соответствующие шаблону регулярного выражения, указанной строкой замены.
	Split	Перегружен. Разделяет входную строку в массив подстрок в позициях, определенных соответствием регулярного выражения.
	Unescape	Отменяет преобразование преобразованных символов во входной строке.

3.9 Группирование элементов регулярного выражения

Для группирования элементов регулярного выражения используются круглые скобки. Группировка применяется для запоминания в специальном массиве (коллекции) значений (фрагментов) анализируемого текста, совпавших со смысловой частью регулярного выражения, которое находится в круглых скобках.

```
      . . .
string text = " 540-356 54-03-56 ili 49-65-78.";
      . . .
foreach (Match T in Regex.Matches(text, @"(\d\d)-(\d\d)-
(\d\d)"))
 {
   Console.WriteLine(T);
   Console.WriteLine(T.Groups[0]);
```

```
    Console.WriteLine(T.Groups[1]);
    Console.WriteLine(T.Groups[2]);
    Console.WriteLine(T.Groups[3]);
    Console.WriteLine(T.Groups[4]);
    Console.WriteLine(T.Groups[5]);
 }
```

. . .

Работа программы:

```
54-03-56
54-03-56
54
03
56

49-65-78
49-65-78
49
65
78
```

Класс Group является наследником класса Capture и, одновременно, родителем класса Match. От своего родителя он наследует свойства Index, Length и Value, которые и передает своему потомку.

«Давайте рассмотрим чуть более подробно, когда и как создаются группы в процессе поиска соответствия. Если внимательно проанализировать предыдущую таблицу, которая описывает символы, используемые в регулярных выражениях, в частности символы группирования, то можно понять несколько важных фактов, связанных с группами:

при обнаружении одной подстроки, удовлетворяющей условию поиска, создается не одна группа, а коллекция групп;

группа с индексом 0 содержит информацию о найденном соответствии;

число групп в коллекции зависит от числа круглых скобок в записи регулярного выражения. Каждая пара круглых скобок создает дополнительную группу, которая описывает ту часть подстроки, которая соответствует шаблону, заданному в круглых скобках;

группы могут быть индексированы, но могут быть и именованными, поскольку в круглых скобках разрешается указывать имя группы.

В заключение отмечу, что создание именованных групп крайне полезно при разборе строк, содержащих разнородную информацию.»[Основы программирования на C# Биллиг Владимир Арнольдович стр.143]

Из распечатки результатов поиска видно, что коллекция групп состоит из двух элементов. Число групп в элементе коллекции соответствует числу пар круглых скобок в записи регулярного выражения (группа с индексом 0 содержит всю информацию о найденном элементе). В остальных группах элемента коллекции находится информация, которая соответствует шаблонам, заданным в круглых скобках групп.

В коде программы дополнительно записан вывод информации от «несуществующих» групп с индексами 4 и 5. На экране монитора появились пустые строки, но сообщения об ошибках нет.

Группирование применяется в случаях, когда требуется задать повторитель для отдельного символа или для последовательности символов в регулярном выражении.

Имя переменной (повторителя) задается в угловых скобках или апострофах, например:

(?<имя переменной>смысловая часть регулярного выражения)

Пусть, например, требуется выделить из текста номера телефонов, записанных в виде nn–nn–nn. Регулярное выражение для поиска номера можно записать следующим образом:

```
@"(?<num>\d{2}-\d{2}-\d{2})"
```

В этом выражении в угловых скобках после символа «?» задается имя группы, а затем шаблон регулярного выражения группы – \d{2}-\d{2}-\d{2}. Фрагмент программы, выполняющей поиск телефонных номеров по шаблону регулярного выражения в некотором тексте, имеет следующий вид:

```
string text = " 540-356 54-03-56 ili 49-65-78.";
     . . .
foreach (Match T in Regex.Matches(text, @"(?<num>\d{2}-\d{2}-
\d{2})"))
     Console.WriteLine( T.Groups["num"]);
     . . .
```

Работа программы:

```
54-03-56
49-65-78
```

При анализе текста в Groups коллекции будут последовательно записываться найденные номера телефонов. Переменная num (номер) предварительно не объявлялась и является частью коллекции – именем первой группы коллекции. Можно просматривать результаты поиска, задавая нулевой индекс каждой группы коллекции. Например:

```
foreach (Match T in Regex.Matches(text, @"(?<num>\d{2}-\d{2}-
\d{2})"))
 Console.WriteLine(T.Groups[0]);
```

Работа программы:

```
54-03-56
49-65-78
```

Рассмотрим еще один вариант применения группирования для формирования обратных ссылок. Все конструкции, заключенные в круглые скобки, автоматически нумеруются, начиная с 1 (нулевой индекс используется для обозначения всего регулярного выражения). Эти номера, предваренные обратной косой чертой, можно использовать для ссылок на соответствующую конструкцию. Например, выражение (\w)\1, точнее @”(\w)\1”, позволяет найти все пары одинаковых алфавитно-цифровых символов. Обычно такое

выражение используется для поиска сдвоенных символов в словах, например, class, mass.

```
      . . .
string text = "Язык регуляррррных выражжений опредееляет шшаблоны символов.";
      . . .
foreach (Match T in Regex.Matches(text,@"(\w)\1"))
  Console.WriteLine("{0} - {1}", T.Index, T.Value);
      . . .
```

Работа программы:

12 - рр
23 - жж
35 - ее
42 - шш

Переменную, имя которой задается внутри выражения в угловых скобках, также можно использовать для обратных ссылок в последующей части выражения. Например, поиск двойных символов в словах можно выполнить с помощью выражения @”((?<s>\w)\k<s>)”. Рассмотрим это регулярное выражение. Группа задана выражением «(?<s>\w)». После знака вопроса в угловых скобках задано имя группы «<s>». После имени группы идет шаблон, описывающий данную группу, в нашем примере шаблон задается выражением «\w», которое означает, что группа может состоять из одного любого алфавитно-цифрового символа, то есть символа из множества прописных и строчных букв и десятичных цифр. За описанием группы находится признак обратной ссылки – пара символов «\k», после которой идет имя группы «<s>».

```
      . . .
string text = "Язык регуляррррных выражжений определяет шаблоны символов.";
      . . .
foreach (Match T in Regex.Matches(text, @"((?<s>\w)\k<s>)"))
 {
   Console.WriteLine(T.Groups["s"]);
   Console.WriteLine(T.Groups[0]);
 }
      . . .
```

Работа программы:

р
рр
ж
жж
е
ее
ш
шш

Вывод результатов поиска для «имени группы» и группы с нулевым индексом различны. Имени группы соответствует только один символ – образец, а группе с нулевым индексом соответствует все, что было найдено.

Можно организовать поиск без учета регистра символа:

```
      . . .
string text = "Яязык регуляррррных выражжений опредееляет шшаблоны символов.";
      . . .
foreach (Match T in Regex.Matches(text,
                    @"((?<s>\w)\k<s>)", RegexOptions.IgnoreCase)
 {
   Console.WriteLine(T.Groups["s"]);
   Console.WriteLine(T.Groups[0]);
 }
      . . .
```

Работа программы:

Я
Яя
р
рр
ж
жж
е
ее
ш
шш

Для поиска в тексте повторяющихся слов (независимо от регистра), расположенных подряд и разделенных произвольным количеством пробелов, можно использовать следующий конструктор регулярного выражения:

```
      . . .
Regex reg = new Regex(@"\b(?<word>\w+)\s+(\k<word>)\b", RegexOptions.IgnoreCase);
string text = "Class  class Regex это Это class regex регулярных выражений.";
      . . .
foreach (Match T in reg.Matches(text))
 {
   Console.WriteLine(T.Groups["word"]);
   Console.WriteLine(T.Groups[0]);
 }
      . . .
```

Работа программы:

Class
Class class
это
это Это

Для поиска в тексте повторяющихся слов (независимо от регистра), расположенных в произвольных позициях строки необходимо «разделители пробелы – \s» заменить на любые символы – «.».

```
. . .
Regex reg = new Regex(@"\b(?<word>\w+).+(\k<word>)\b",
RegexOptions.IgnoreCase);
string text = "Class  class Regex это Это class regex регулярных
выражений.";
     . . .
foreach (Match T in reg.Matches(text))
 {
   Console.WriteLine(T.Groups["word"]);
   Console.WriteLine(T.Groups[0]);
 }
     . . .
```

Работа программы:

```
Class
Class  class Regex это Это class
```

Результат поиска – только одна группа, хотя, кажется, что их должно быть больше. Обратим внимание на работу повторителя «.+» (его часто называют жадным «greedy»), означающего одно или более повторений предыдущего элемента. Для первого слова текста, а это слово «Class», повторитель ищет все совпадения до конца текста и найденный фрагмент «вырезается» из текста и не участвует в дальнейшем его анализе.

Изменим текст нашего примера следующим образом:

```
string text = "Class Это class Regex это Это  regex это
регулярные Это  выражения.";
```

Работа программы:

```
Class
Class Это class
Regex
Regex это Это  regex
это
это регулярные Это
```

В результате поиска найдены три группы коллекции.

3.10 Применение методов Split и Replace в регулярных выражениях

Класс Regex содержит не только методы поиска в тексте, но и некоторые другие методы обработки текстов. Из непоисковых методов класса Regex наибольшее применение имеют метод Split, позволяющий задавать регулярным выражением разделитель текста, и метод Replace, выполняющий замены фрагментов текста в соответствии с заданным регулярным выражением.

Рассмотрим использование этих методов в учебных примерах.

Предположим, что для кодирования некоторого текста на русском языке используются десятичные числа от 00 до 32 (значение регистра игнорируется). Символ «запятая» кодируется числом 33, «точка» – числом 34, а символ «пробел» кодируется числом 35. Остальные символы текста, включая цифры и буквы других алфавитов, кодируются случайными числами в диапазоне от 40 до 99. В закодированном тексте числа отделяются символом «пробел». Необходимо исходный текст закодировать с помощью приведенных правил и записать в новый временный текст. Предусмотреть декодирование временного текста. При декодировании выполнять преобразование чисел от 00 до 35 в соответствии с приведенными правилами. Числа в диапазоне от 40 до 99 игнорировать. Предусмотреть просмотр исходного, кодированного и декодированного текстов на экране монитора. В программе использовать метод Split и Replace класса Regex.

Создаем отдельный класс «PreobText», в котором организуем конструктор и строковые методы кодирования и декодирования текста. Сам текст будет передаваться конструктору при инициализации полей класса.

Метод кодирования текста включает «разделении» исходного текст на «слова» с помощью метода Split класса Regex. Далее выполняется кодирование каждого «слова» текста в соответствии с заданием. Полученные последовательности двухразрядных чисел «объединяются» в новый текст, который возвращается методом в программу.

Метод декодирования преобразует последовательности двухразрядных чисел в новый текст с использованием Replace класса Regex. Далее текст декодируется в соответствии с заданием.

На каждом этапе работы программы (меню программы) предусмотрена печать результата работы пункта меню.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Text;
namespace ConsoleApplication1
{
 class Program
 {
  class PreobText
  {
   string text;
   public string rez;
   string alfavit = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя,. ";
   //Конструктор
   public PreobText(string txt)
   {
    text = txt;
    rez = "";
   }
```

```
  // метод кодирования
  public string Codirovanie()
  {
   bool ok = false;
   text = text.ToLower();
   int m = 0, k = 0, Kol = 0, r = 0;
   Random rnd = new Random();
   string p, clo;
   //Console.WriteLine("nacalo preobrazovanij!");
   //Console.WriteLine(text);
   //Console.WriteLine(rez);
List<string> clova=new List<string>(Regex.Split(text,@"[ ]"));
   m = clova.Count;
   string[] newclova = new string[m];
   foreach (string ss in clova)
   {
    Kol = ss.Length;
    //  Console.WriteLine(Kol);
    clo = "";
    if (Kol != 0)
    {
     for (int j = 0; j < Kol; j++)
     {
      ok = false;
      for (int i = 0; i < 36; i++)
      {
       if (ss[j] == alfavit[i])
       {
        ok = true;
        if (i <= 9)
        { p = "0" + i.ToString() + " "; }
          else p = i.ToString() + " ";
        clo = clo + p;
       }
      }
      if (ok == false)
      {
       r = rnd.Next(40, 99);
       clo = clo + r.ToString()+ " ";
      }
     }
     clo.Trim();
     if (clo != "") { newclova[k] = clo; k++; }
     //Console.WriteLine(newclova[k - 1]);
    }
   }
   // Console.WriteLine("konec preobrazovanij!");
   // Console.WriteLine();
   for (int j = 0; j < m; j++)
   {
    //Console.WriteLine(newclova[j]);
    rez = rez + newclova[j] + "35" + " ";
```

```
     };
     return rez;
    }
    // метод декодирования
    public string Decodirovanie()
    {
     string dec = "";
     int i=0,Kol=0;
     dec = Regex.Replace(rez, @"\d{2}", @"<$0");
     Console.WriteLine(dec);
     List<string> clova=new List<string>(Regex.Split(dec,@"<"));
     Kol = clova.Count;
     // Console.WriteLine(Kol);
     rez = "";
     foreach (string ss in clova)
     {
      Kol = ss.Length;
      if (Kol != 0)
      {
       i = Convert.ToInt32(ss);
       if (i < 36) rez = rez + alfavit[i];
      }
     }
     // Console.WriteLine("Konec");
     return rez;
    }
   }
   static void Main(string[] args)
   {
    string txt = "CG1но34в55а! н41ас веGamesдуZ23т куТід44а 8767
то, с62нкwовtta я таскалщу росюакотзукак.";
    PreobText ob = new PreobText(txt);
    string ntxt;
    int w = 0;
    string buf;
    while (w < 4)
    {
     Console.WriteLine("1 - Просмотр исходного текста");
     Console.WriteLine("2 - Кодирование текста");
     Console.WriteLine("3 - Декодирование текста");
     Console.WriteLine("4 - Выход из программы");
     Console.WriteLine("Введите пункт меню программы");
     buf = Console.ReadLine();
     w = Convert.ToInt32(buf);
     switch (w)
     {
      case 1: Console.WriteLine(txt); break;
      case 2:
             {
              ntxt = ob.Codirovanie();
              Console.WriteLine(ntxt);
              break;
```

```
                }
            case 3:
                {
                 ntxt = ob.Decodirovanie();
                 Console.WriteLine(ntxt);
                 break;
                }
            default: break;
          }
        }
      }
    }
}
```

Работа программы:

1 - Просмотр исходного текста
2 - Кодирование текста
3 - Декодирование текста
4 - Выход из программы
Введите пункт меню программы
1
CG1но34в55а! н41ас веGamesдуZ23т куTid44a 8767 то, с62нkwовtta я tackaлщу росюак omзykак.
1 - Просмотр исходного текста
2 - Кодирование текста
3 - Декодирование текста
4 - Выход из программы
Введите пункт меню программы
2
18 41 79 14 15 50 87 02 80 59 00 53 35 14 59 60 00 18 35 02 05 70 54 88 77 47 04
20 43 47 77 19 35 11 20 55 98 04 46 52 00 35 46 70 97 76 35 19 15 33 35 18 54 9
0 14 68 79 15 02 54 77 00 35 32 35 19 00 85 86 62 56 26 20 35 17 91 54 31 98 11
74 79 72 92 45 00 11 34 35
1 - Просмотр исходного текста
2 - Кодирование текста
3 - Декодирование текста
4 - Выход из программы
Введите пункт меню программы
3
<18 <41 <79 <14 <15 <50 <87 <02 <80 <59 <00 <53 <35 <14 <59 <60 <00 <18 <35 <02
<05 <70 <54 <88 <77 <47 <04 <20 <43 <47 <77 <19 <35 <11 <20 <55 <98 <04 <46 <52
<00 <35 <46 <70 <97 <76 <35 <19 <15 <33 <35 <18 <54 <90 <14 <68 <79 <15 <02 <54
<77 <00 <35 <32 <35 <19 <00 <85 <86 <62 <56 <26 <20 <35 <17 <91 <54 <31 <98 <11
<74 <79 <72 <92 <45 <00 <11 <34 <35
снова нас ведут куда то, снова я тащу рюкзак.
1 - Просмотр исходного текста
2 - Кодирование текста
3 - Декодирование текста
4 - Выход из программы
Введите пункт меню программы

Тема 4 ПЕРЕДАЧА ДАННЫХ МЕЖДУ НИТЯМИ В ПРОЦЕССЕ НА ЯЗЫКЕ C#

4.1 Понятие нитей в системном программировании.

4.1.1 Понятие нити.

Работа процессора компьютера заключается в выполнении команд (инструкций), которые формируются после компиляции программы. Эта последовательность инструкций программы называется потоком команд управления программы. Термин «поток управления» применяется в языке C++. В языке C# принято называть поток команд управления программы нитью – сравнивают с нитью (thread), на которую «нанизаны» инструкции выполняемые процессором компьютера.

Операционная система называется однопрограммной, если в ней может существовать только одна нить.

Операционная система называется мультипрограммной, если в ней может одновременно существовать несколько нитей разных программ.

Эффективность работы компьютера во многом определяется загрузкой процессора компьютера. Поэтому важнейшей задачей мультипрограммной ОС является диспетчеризация нитей программ таким образом, чтобы обеспечить эффективную загрузку процессора.

Нитью программы может являться любой независимо выполняемый фрагмент программы.

Рассмотрим пример учебной программы, в которой вычисляется сумма двух чисел аи b.

```
using System;
namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main(string[] args)
  {
int a,b,c;
string buf;
Console.Write("Введите целое значение a ");
buf = Console.ReadLine();
   a = Convert.ToInt32(buf);
Console.Write("Введите целое значение b ");
buf = Console.ReadLine();
   b = Convert.ToInt32(buf);
   c = a + b;
Console.WriteLine("a+b={0}", c);
Console.WriteLine("Для продолжения нажмите клавишу Enter");
Console.ReadLine();
  }
 }
```

```
}
```

Каждое действие этой программы всегда однозначно определено и выполняется одно за другим – программа содержит только одну нить.

Изменим программу, включив в нее функцию вычисления суммы aиb.

```
using System;
namespace ConsoleApplication1
{
classProgram
 {
publicstaticint sum(int a, int b)
  { return a + b; }
staticvoid Main(string[] args)
  {
int a, b, c;
string buf;
Console.Write("Введите целое значение a ");
buf = Console.ReadLine();
   a = Convert.ToInt32(buf);
Console.Write("Введите целое значение b ");
buf = Console.ReadLine();
   b = Convert.ToInt32(buf);
   c = sum(a,b);
Console.WriteLine("a+b={0}", c);
Console.WriteLine("Для продолжения нажмите клавишу Enter");
Console.ReadLine();
  }
 }
}
```

Если при выполнении программы функция**Main** ждет выполнение функции **sum**, то программа остается с одной нитью.

Если при выполнении программы функция**Main**не ждет выполнение функции **sum**, а продолжает выполняться, то в программе образуются две нити – главная, нить функции **Main,**и нить функции **sum**. В этом случае значение переменной **c**, выведенное на экран монитора функцией **Main**, может не соответствовать сумме переменных **a и b**.

Еще больше нитей может существовать в программах, использующих операторы условных переходов.

Для организации правильной работы программы необходимо научится передавать данные между нитями, совместно использовать данные и синхронизировать(упорядочить) работу различных нитей или нитевых функций.

4.1.2 Требования к методам, используемым параллельными нитями

Рассмотрим, каким требованиям должны удовлетворять методы, чтобы их можно было «безопасно» вызывать параллельными нитями.

Для этого определим понятие контекста нити – как область памяти компьютера, к которой может обращаться нить во время своего исполнения.

Рассмотрим работу следующей функции:

```
public static int funk1(int n)
{
if (n >= 0) n --;
if (n  < 0) n ++;
return n;
}
```

При вызове этой функции создается копия переменной **n,** которая изменяется в соответствии с условием и результат возвращается нити. Далее копия переменной удаляется из памяти. Такая функция называется безопасной для нити.

Изменим функцию **funk1** так, чтобы она работала с глобальной переменной **n**.

```
int n;
. . .
public static void funk2()
{
if (n>= 0) n --;
if (n< 0) n ++;
}
```

При параллельном вызове функции **funk2** несколькими нитями может произойти некорректное изменение значения переменной **n**(может возникнуть изменение переменной **n**больше чем на единицу). Такие функции называют не безопасными для нити.

В некоторых языках программирования, например, в C++, существует понятие статических переменных функций. Статические переменные объявляются внутри функции (локальные переменные), но значение этих переменных сохраняется в памяти по завершении работы функции и может быть использовано при повторном вызове функции. То есть такая статическая переменная ведет себя внутри функции как глобальная переменная программы. При параллельном обращении к такой функции нескольких нитей значение статической переменной может не соответствовать количеству вызовов функции той или иной нити.

В общем случае функция называется безопасной или реентерабельной, если она удовлетворяет следующим требованиям:

- не использует глобальные переменные, значения которых изменяются параллельно работающими нитями;
- не использует статические переменные, определенные внутри функции и изменяемые параллельно работающими нитями;
- не возвращает указатели на статические данные, определенные внутри функции.

Перечисленные требования к безопасным функциям можно обеспечить, если использовать различные приемы блокировки доступа к ресурсам функций различных нитей, которые они совместно используют.

4.1.3 Состояния нити

Состояние нити зависит от готовности процессора компьютера работать с программой, включающей нить и от готовности самой программы.

Г5отовность процессора определяется его «выделением» на исполнение данной программы – т.е. процессор получает квант времени для работы с данной программой.

Готовность программы зависит от получения ей всех ресурсов необходимых для выполнения программы. В зависимости от процессора и программы возможны следующие состояния потока:

- нить готова к выполнению («не выделен», «готова»);
- нить блокирована («не выделен», «не готова»);
- нить выполняется («выделен», «готова»).

Обычно к перечисленным состояниям нити добавляются еще два – новая (нить создается, но не готова) и завершено (состояние соответствующее окончанию работы нити).

В программах, содержащих несколько нитей, работа которых взаимосвязана, существует еще два состояния нити – приостановка выполнения нити и возобновление выполнения нити.

С учетом вех перечисленных состояний нити можно нарисовать следующую модель ее поведения.

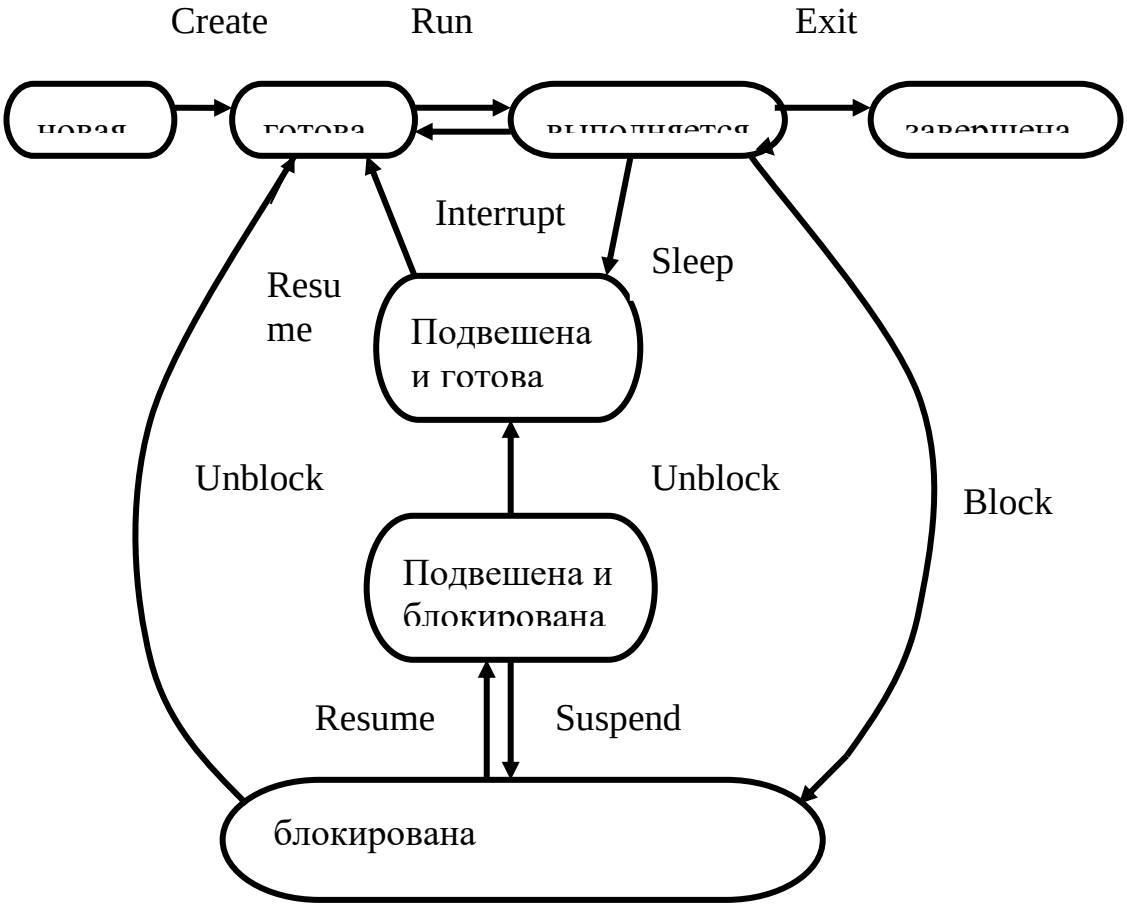


Рисунок 4.1.1 – Модель семи состояний нити.

Операция Create переводит нить из состояния «новая» в состояние «готова».

Операция Run переводит нить из состояния «готова» в состояние «выполняется» - нити выделяется процессорное время.

Операция Exit переводит нить из состояния «выполняется» в состояние «завершена».

Операция Interrupt переводит нить из состояния «выполняется» в состояние «готова» - обычно эта операция выполняется над нитью, когда истекло процессорное время, выделенное на исполнение нити.

Операция Resume возобновляет исполнение нити.

Операция Sleep приостанавливает исполнение нити.

Операция Unblock переводит нить из состояния «блокирована» в состояние «готова».

Операция Block переводит нить из состояния «выполняется» в состояние «блокирована» - обычно эта операция выполняется над нитью, когда она ждет наступление некоторого события, например, ввода данных или освобождение некоторого ресурса.

Операция Suspend приостанавливает исполнение нити.

4.1.4 Диспетчеризация и планирование нитей

Для пояснения принципов диспетчеризация и планирование нитей будем считать, что компьютер имеет только один процессор.

Для организации мультипрограммного режима работы процессора его время работы делится на кванты – интервалы, которые выделяются нитям для их работы (в Windows это примерно 20 – 30 миллисекунд). По истечении кванта времени исполнение нити прерывается и процессор назначается другой нити. Распределением квантов времени между нитями занимается специальная программа – менеджер нитей.

Когда менеджер нитей переключает процессор на исполнение другой нити, он должен выполнить следующие действия:

- сохранить контекст прерываемой нити;
- восстановить контекст запускаемой нити;
- передать управление запускаемой нити.

Если все обслуживаемые нити имеют одинаковый приоритет исполнения, то алгоритм управления нитями легко реализуется с помощью «очереди» по принципу FIFO (firstin – firstout) пришел первым- первым вышел, и прерванные нити становятся в конец очереди.

Схематично обслуживание нитей представлено на рисунке 4.1.2

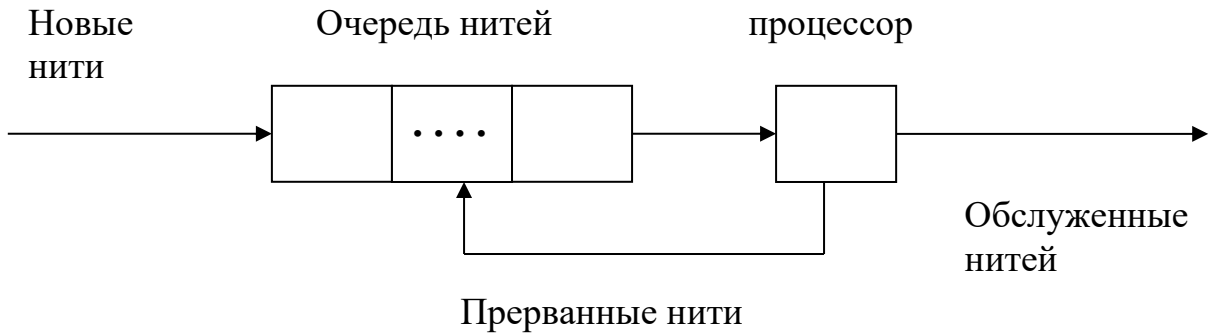


Рисунок 4.1.2 – Обслуживание нитей

Если нити имеют разные приоритеты, то формируются очереди нитей с одинаковыми приоритетами. Простейший алгоритм обслуживания нескольких очередей основан на обслуживании очередей с учетом приоритета их нитей. Схематично обслуживание нитей представлено на рисунке 4.1.3

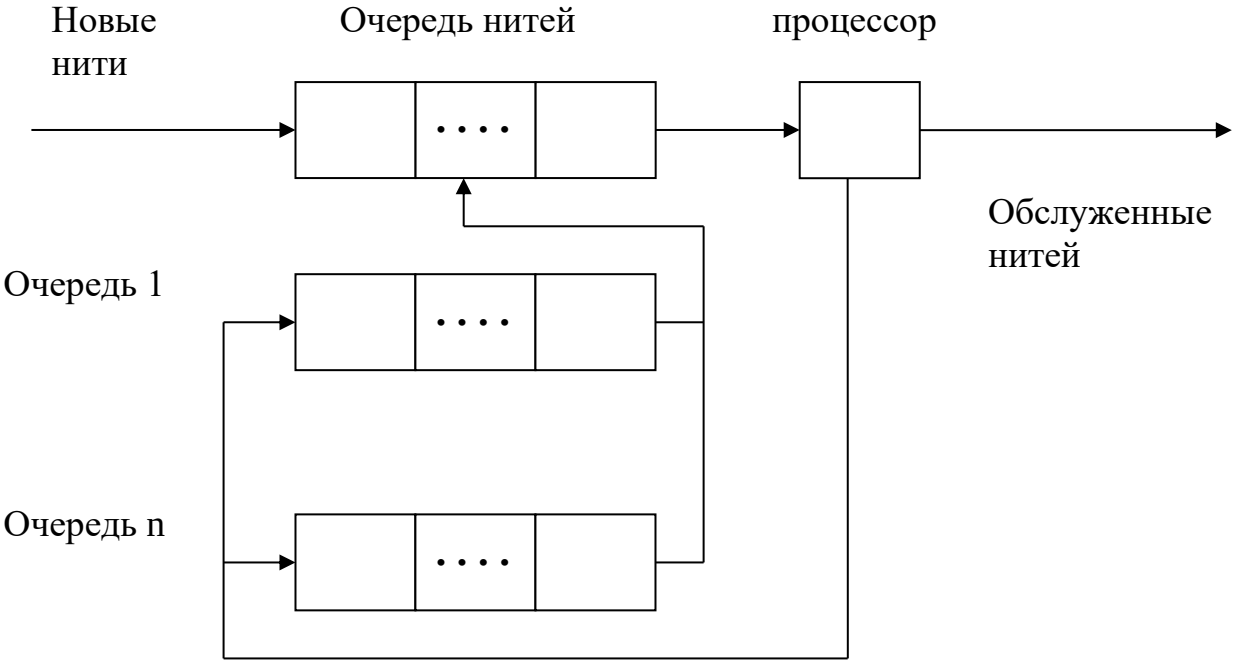


Рисунок 4.1.3 – Обслуживание нескольких очередей с разными приоритетами нитей

Алгоритмы управления нитями должны учитывать следующие параметры системы:

- время загрузки микропроцессора должно быть максимальным;
- время нахождения нити в системе должно быть минимальным;
- время ожидания нитей должно быть минимальным;
- пропускная способность системы должна быть максимальной;
- время реакции системы на обслуживание заявки должно быть минимальным.

На многопроцессорных компьютерах менеджер нитей использует как квантование времени работы нитей, так и параллельность работы процессоров, когда разные нити выполняют код на разных процессорах.

Необходимость квантования времени все равно остается, так как операционная система должна обслуживать как свои собственные (системные) нити, так и нити других приложений.

4.1.5 Определение нити

Приложение в Windows может состоять из одного или нескольких процессов, каждый из которых, может порождать одну или несколько нитей. Каждой нити выделяются некоторые ресурсы операционной системы, которые представляются в виде объекта.

Нитью в Windows называется объект, которому операционная система выделяет процессорное время для работы с системными ресурсами.

Каждой нити принадлежат следующие ресурсы:

- код исполняемой функции;
- набор регистров процессора;
- стек для работы приложения;
- стек для работы операционной системы;
- маркер доступа с информацией для системы безопасности.

Эти ресурсы образуют контекст нити.

Любому объекту в системе Windows присваивается свой регистрационный номер – дескриптор. Кроме дескриптора каждой нити в Windows присваивается свой идентификатор. Идентификаторы нитей используются служебными программами и позволяют отслеживать работу нитей.

В Windows различают нити двух типов – системные и пользовательские.

Системные нити запускаются ядром операционной системы и служат для выполнения различных системных задач.

Пользовательские нити запускаются приложениями и служат для решения задач пользователя.

В технологии .NET эта схема выделения ресурсов несколько изменена – каждый процесс приложения .NET может состоять из одного или нескольких доменов приложения, а в рамках домена может работать одна или несколько нитей (все свойства нити как объекта при этом сохраняются).

Ресурсы каждого домена полностью изолированы друг от друга. Различные домены приложения не могут совместно использовать никакие данные, будь то статические поля или глобальные переменные, которые могут использоваться разными нитями в рамках только общего домена.

Таким образом, домен это дополнительный уровень «изоляции» или «защиты» данных в .NET, который создается между процессом и нитью.

Необходимо отметить, что для некоторых приложений в .NET в рамках домена можно создавать пулы нитей – это специальные мини очередь для

текущих нитей, которые создаются методом ThreadPool.QueueUserWorkItem вместо создания и запуска объекта Thread.

Как правило, диспетчер пула нитей устанавливает количество нитей в пуле автоматически (по умолчанию это значение равно 50). Можно самим установить верхний предел количества нитей в пуле методом ThreadPool.SetMaxThreads.

Обычно пулы применяются, когда необходимо организовать работу произвольного числа параллельно работающих приложений, например, на веб-серверах. При работе WebServices, ASP.NET, серверов WCF и т.п. пулы нитей выделяются автоматически.

4.1.6 Создание нити

При запуске любой программы на языке C# автоматически создается главная нить программы. В консольном приложении главная нить программы соответствует методу Main.

Для запуска дополнительной нити необходимо создать объект классаThread. При создании объекта нити конструктору класса Thread передается делегат, в котором описан метод ThreadStart, позволяющий запускать в дополнительной нити метод, имя которого указано в качестве формального параметра метода делегата. Например,

Thread Pervij_dop = new Thread(new ThreadStart(Poick));

где Pervij_dop – имя первой дополнительной нити;

Poick – имя метода, который будет запущен в дополнительной нити.

Обычно делегат работает «по умолчанию» и создание объекта нити выглядит следующим образом:

Thread Pervij_dop = new Thread(Poick);.

Pervij_dop.Start();

Создавая консольное приложение, работающее с нитями, необходимо подключать пространство имен System.Threading.

Класс Thread имеет набор свойств и методов, некоторые из которых представлены следующим списком:

CurrentThread – статическое свойство, предназначенное только для чтения, которое возвращает ссылку на нить, выполняемую в настоящее время;

Sleep() – это статический метод, который приостанавливает выполнение текущей нити на указанное пользователем время;

IsAlive – это обычное свойство (требует создание объекта нити), возвращает **true** или **false** в зависимости от того, запущена нить или нет;

IsBackground – это обычное свойство, предназначенное для получения или установки значения, которое показывает, является ли эта нить фоновой;

Name – это обычное свойство для установки дружественного текстового имени нити;

Priority –обычное свойство, позволяющее получить/установить приоритет нити (используются значения из перечисления ThreadPrority);

ThreadState – обычное свойство, позволяющее получить информацию о состоянии нити (используются значения из перечисления ThreadState);

Interrupt() – обычный метод, прерывающий работу текущей нити;

Join() – обычный метод, который ждет появления другой нити или указанный промежуток времени;

Resume() –обычный метод, который продолжает работу после приостановки нити;

Start() –обычный метод, который начинает выполнение нити, определенной делегатом ThreadStart

Suspend() –обычный метод, который приостанавливает выполнение нити. Если выполнение нити уже приостановлено, то метод игнорируется.

Все свойства и методы класса Thread приведены в приложении данной лекции.

Рассмотрим чисто учебный пример создания объекта дополнительной нити и ее запуска из главной нити программы. Для визуального контроля работы главной и дополнительной нитей, традиционно в них используется цикл forc выводом некоторых символов или цифр. Не будем отступать от традиций:

```
usingSystem;
usingSystem.Threading;

namespace ConsoleApplication1
{
classThread1
    {
staticvoid WriteCimB()
        {
for (int i = 1; i <= 100; i++) Console.Write("B");
        }
staticvoid Main()
        {
Thread t_dop = newThread(WriteCimB);
            t_dop.Start();
for (int i = 1; i <= 100; i++) Console.Write("A");
Console.ReadLine();
        }
    }
}
```

Работа программы:

```
AAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

Главная нить предназначена для вывода в консольное окно 100 символов «А», одновременно она создает дополнительную нить, которая также выводит

в консольное окно один символ 100 раз – «B». Из работы программы видно, что они работают одновременно – прерываются циклы **for** только квантами времени, выделяемыми нитям. На компьютере с одним процессором каждой нити для работы выделяется квант времени (в Windows это 20 мс.).

Можно исследовать состояние дополнительной нити, включив в программу следующие строки:

```
Console.WriteLine("состояние доп нити ={0}", t_dop.IsAlive);
Console.WriteLine("приоритет доп нити ={0}", t_dop.Priority);
```

Работа программы:

```
состояние доп нити =True
приоритет доп нити =Normal
AAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

4.2 Использование данных разными нитями одного процесса

4.2.1 Использование локальных переменных разными нитями

В предыдущей лекции мы рассмотрели работу двух нитей, которые выводили символы «A» и «B» в консольное окно экрана монитора. Нити работали независимо друг от друга. Это происходит, потому что в языке C# каждому объекту выделяется свой стек данных и локальные переменные хранятся отдельно.

Если создать один объект для некоторого класса, содержащего метод вывода чисел в консольное окно и локальную переменную, то можно проверить, можно ли изменять локальную переменную объекта этого класса при работе разных нитей. Это можно организовать, запустив метод нашего объекта в главной нити и в одной дополнительной.

Исходный код программы:

```
usingSystem;
using System.Threading;

namespace ConsoleApplication1
{
classDanie_1
    {
classPervaj_1
        {
bool flag;
publicvoid Niti_1()
            {
Console.WriteLine("Работаетнить");
if (!flag)
                {
                    flag = true;
for (int i = 0; i < 10; i++)
```

```
Console.Write(i + " ");
Console.WriteLine();
                }
            }
        }
staticvoid Main()
        {
Pervaj_1 tt = newPervaj_1();
newThread(tt.Niti_1).Start();
tt.Niti_1();
Console.ReadLine();
        }
    }
}
```

Результат работы программы:

Работает нить
Работает нить
0 1 2 3 4 5 6 7 8 9
или

Работает нить
0 1 2 3 4 5 6 7 8 9
Работает нить

Как видно из работы нашей программы обе нити работают с локальной переменной **flag**. Выводится только одна последовательность чисел, а другая блокируется – нить «видит» значение переменной **flag**.

Если создать два объекта и для каждого запустить свою нить, то для каждого объекта (соответственно каждой нити) будет создаваться своя локальная переменная **flag** и взаимное использование локальной переменной разными нитями прекратиться.

Исходный код программы:

```
usingSystem;
using System.Threading;

namespace ConsoleApplication1
{
classDanie_1
    {
classPervaj_1
        {
bool flag;
publicvoid Niti_1()
            {
Console.WriteLine("Работаетнить");
if (!flag)
                {
                    flag = true;
for (int i = 0; i < 10; i++)
Console.Write(i + " ");
```

```
Console.WriteLine();
                }
            }
        }
staticvoid Main()
        {
Pervaj_1 tt = newPervaj_1();
newThread(tt.Niti_1).Start();
Pervaj_1 vv = newPervaj_1();
newThread(vv.Niti_1).Start();
//tt.Niti_1();
Console.ReadLine();
        }
    }
}
```

Результат работы программы:

Работает нить
Работает нить
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
или

Работает нить
0 1 2 Работает нить
0 1 2 3 4 5 6 7 8 9
3 4 5 6 7 8 9
и т.д.

Таким образом, о совместном использовании данных разными нитями имеет смысл говорить, если нити используют методы или данные, принадлежащие одному объекту.

В теории использования разными нитями данных и методов, как правило, рассматриваются и вопросы работы нитей со «статическими» элементами, т.е. элементами, имеющими спецификатор доступа **static**. Объявим в нашей программе одну статическую переменную и один метод и попробуем их использовать в разных нитях.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classDanie_1
 {
staticint k = 0;
staticvoid cikl()
  {
   k++;
Console.WriteLine("Работает поток " + k);
int n1, n2;
   n1 = k * 10; n2 = (k + 1) * 10;
for (int i = n1; i < n2; i++)
```

```
Console.Write(i + " ");
Console.WriteLine();
  }

staticvoid Main()
  {
newThread(cikl).Start();
newThread(cikl).Start();
newThread(cikl).Start();
newThread(cikl).Start();
   cikl();
Console.ReadLine();
  }
 }
}
```

Работа программы:

Работает поток 1
Работает поток 3
50 Работает поток 2
50 51 52 53 54 55 56 57 58 59
50 51 52 53 54 55 56 57 58 59
51 52 53 54 55 56 57 58 59
Работает поток 4
Работает поток 5
50 51 52 53 54 55 56 57 58 59
50 51 52 53 54 55 56 57 58 59

Все запущенные нити используют одну и ту же статическую переменную и работают с одним методом. И если использовать термин безопасности работы нитей – рентабельность, то все они не рентабельны.

Один из вариантов обеспечения безопасности работы нитей и использования общих данных заключается в использовании в них оператора **lock**, позволяющего блокировать работу всех остальных нитей (фактически блокирование всей программы) до завершения работы текущей нити.

Оператор **lock** относится к специальным блокирующим конструкциям языка C#.

Рассмотрим, как оператор **lock** определяют различные авторы учебников по программированию на языке **C#.**

Оператор **lock** является синтаксическим сокращением для вызова методов **Enter** и **Exit** класса **Monitir** с блоком **try/finally**[Албахари стр.743].

Ключевое слово **lock** позволяет блокировать блок кода таким образом, что в каждый момент времени его может использовать только одна нить [Троелсен стр.312].

Павловская Т.А. определяет **lock** как оператор со следующим форматом записи:

lock (выражение)

{ блок операторов}

Выражение определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово **this**, для

статических методов – **typeof(**класс). Блок операторов задает критическую секцию кода, которую требуется заблокировать [Павловская стр. 242].

Во всех определениях присутствует единое утверждение, что **lock** включает некоторый блок кода, который блокируется для использования многими нитями. С методической точки зрения лучшим и понятным определением является определение Павловской Т.А., которого мы будем придерживаться в наших лекциях.

Итак, **lock** это оператор, позволяющий блокировать секцию кода таким образом, что в каждый момент времени его может использовать только одна нить.

Исходный код программы:

```
usingSystem;
using System.Threading;

namespace ConsoleApplication1
{
classDanie_1
 {
staticint k = 0;
staticobject t = typeof(object);
staticvoid cikl()
  {
Console.WriteLine("Работаетнить ---- " + k);
lock (t)
   {
    k++;
Console.WriteLine("Работаетнить № " + k);
int n1, n2;
    n1 = k * 10; n2 = (k + 1) * 10;
for (int i = n1; i < n2; i++)
Console.Write(i + " ");
Console.WriteLine();
   }
  }

staticvoid Main()
  {
newThread(cikl).Start();
newThread(cikl).Start();
newThread(cikl).Start();
newThread(cikl).Start();
   cikl();
Console.ReadLine();
  }
 }
}
```

Работа программы:

```
Работает нить ---- 0
Работает нить ---- 0
Работает нить ---- 0
```

```
Работает нить ---- 0
Работает нить ---- 0
Работает нить № 1
10 11 12 13 14 15 16 17 18 19
Работает нить № 2
20 21 22 23 24 25 26 27 28 29
Работает нить № 3
30 31 32 33 34 35 36 37 38 39
Работает нить № 4
40 41 42 43 44 45 46 47 48 49
Работает нить № 5
50 51 52 53 54 55 56 57 58 59
```

В нашей программе одновременно запускаются все нити, но выполнение очередной нити приводит к блокировке выполнения остальных нитей программы с помощью оператора **lock**.

Подобные задачи возникают, когда необходимо, чтобы при работе нити значения некоторых переменных, используемых и другими нитями программы, не изменялись до окончания работы нити, например, при выполнении операций со счетами клиентов банка.

Нить, закончившая работу, не может быть начата заново.

4.2.2 Передача данных нитям

Рассмотренный нами делегат **ThreadStart**, используемый при создании объектов нити, не имеет формальных параметров для передачи данных нитям – делегат способен только определять имя запускаемого метода. Однако в языке **C#** конструктор нити перегружен и допускает использование и другого делегата платформы **.NET Framework**, способного не только создавать объект, но и передавать некоторый параметр –**ParameterizedThreadStart**. Этот делегат имеет следующий формат записи:

public delegate void ParameterizedThreadStart(object obj);

Метод **ParameterizedThreadStart** способен принимать только один параметр типа **object** – любой объект в языке **C#** является производным от **object**.

Рассмотрим учебный пример, в котором будем явно указывать название используемого делегата (в учебных целях) и передавать некоторый аргумент методу, для которого запускаются нити.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classNit_Par
 {
staticvoid cikl(object ob)
  {
int k = (int)ob;
Console.WriteLine("Работаетнить ---- " + k);
```

```
   k++;
Console.WriteLine("Работаетнить № " + k);
for (int i = 0; i <= k; i++)
Console.Write(i + " ");
Console.WriteLine();
  }

staticvoid Main()
  {
Thread[] name = newThread[5];
for (int j = 0; j < 5; j++)
      {
       name[j] = newThread(newParameterizedThreadStart(cikl));
       name[j].Start(j);
      }
Console.ReadLine();
}
 }
}
```

Работа программы:

Работает нить ---- 0
Работает нить № 1
0 1
Работает нить ---- 1
Работает нить ---- 2
Работает нить ---- 3
Работает нить № 4
0 1 2 Работает нить № 3
0 1 2 3
3 4
Работает нить ---- 4
Работает нить № 5
0 1 2 3 4 5
Работает нить № 2
0 1 2

В приведенном примере создается массив объектов класса **Thread** с помощью делегата ParameterizedThreadStart(object obj);, который позволяет во время запуска нитевого метода передавать ему один параметр типа **object**.

Запускаемый в нити метод должен выполнить явное преобразование полученного параметра в целое число.

Как мы уже отмечали, в языке **C#** конструктор нити перегружен и допускает использование нескольких делегатов платформы **.NET Framework** по умолчанию. Поэтому обычно в программах не прописывается название используемого делегата, например, в нашей программе метод **Main** обычно записывается следующим образом:

```
staticvoid Main()
  {
Thread[] name = newThread[5];
for (int j = 0; j < 5; j++)
```

```
        {
            name[j] = newThread(cikl);
            name[j].Start(j);
        }

Console.ReadLine();
}
```

Упорядочение работы нитей можно осуществлять с помощью, например, оператора **lock**.

Другой способ передачи данных в нить состоит в запуске в нити метода определенного экземпляра объекта, который позволяет работать с полями объекта, а не статического метода, используемого без объекта и без его полей.

Свойства выбранного экземпляра объекта будут определять поведение нитей.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
publicstaticint kol=0;
classT
  {
privatedouble data;
privateint c, n;
publicThread thread1;

public T()
   {
    thread1 = newThread(run);
    thread1.Start(kol);
   }
publicvoid run(object ob)
   {
lock (this)
    {
     kol++; n = kol;
Console.WriteLine("Нить {0} стартовала ",kol);
Console.WriteLine("Введитечисло");
     c = int.Parse(Console.ReadLine());
     if (kol == 4) data = c*40;
if (kol < 4)
     {
switch (kol)
      {
case 1: data = Math.Cos(c); break;
case 2: data = Math.Exp(c); break;
case 3: data = c + 30; break;
      }
```

```
T thr2 = newT();
      thr2.thread1.Join();
     }
Console.WriteLine("Нить {0} завершена! Значениеравно:
                                                    {1}",n,data);
    }
   }
  }
staticvoid Main()
  {
T thr1 = newT();
   thr1.thread1.Join();
Console.ReadKey();
  }
 }
}
```

Работа программы:

```
Нить 1 стартовала
Введите число
1
Нить 2 стартовала
Введите число
2
Нить 3 стартовала
Введите число
3
Нить 4 стартовала
Введите число
4
Нить 4 завершена! Значение равно: 160
Нить 3 завершена! Значение равно: 33
Нить 2 завершена! Значение равно: 7,38905609893065
Нить 1 завершена! Значение равно: 0,54030230586814
```

В нашем примере запускаются 4 нити, каждая из которых использует метод **run** класса **T** и, в качестве параметра, получает переменную **kol**, значение которой соответствует номеру запущенной нити. Метод **run** является методом класса **T**, поэтому он может работать с полями объекта этого класса, например, вычислять некоторое выражение, используя значение поля **c**, индивидуально заданное для каждой нити. В нашей программе в каждой нити использовано свое выражение, результат которого присваивается вещественной переменной **data**.

В программе использован рекурсивный способ запуска очередной нити – ограничение числа запускаемых нитей осуществляется с помощью условия if (kol< 4). Переменная **kol** изменяется от 0. В каждой запущенной нити создается свой объект T thr2 = new T().

Внутри созданного объекта запускается метод Join() (thr2.thread1.Join();), который болкирует вызывающий объект до его завершения.

4.3 Режимы работы нитей. Процессы в Windows

4.3.1 Понятие основной и фоновой нити

По умолчанию нити создаются как основные, что означает, что приложение не будет завершено, пока одна из таких нитей будет исполняться. В языке C# существует понятие фоновых нитей, которые завершаются сразу же, как только все основные нити будут завершены (считается, что они не «продлевают» жизнь приложению).

Статус нити – основная или фоновая определяется состоянием свойства IsBack ground, которое имеет логический тип.

В качестве учебной программы рассмотрим работу двух нитей с разным временем задержки:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
publicstaticvoid Niti_1()
  {
Console.WriteLine("Работаетнить 1 :");
for (int i = 0; i < 10; i++)
   {
Console.Write(i + " ");
Thread.Sleep(1000);
   }
Console.WriteLine();
  }
publicstaticvoid Niti_2()
  {
Console.WriteLine("Работаетнить 2 :");
for (int i = 10; i < 20; i++)
   {
Console.Write(i + " ");
Thread.Sleep(1000*2);
   }
Console.WriteLine();
  }
staticvoid Main(string[] args)
  {
Thread nt1 = newThread(Niti_1);
Thread nt2 = newThread(Niti_2);
   nt1.IsBackground = false;
   nt2.IsBackground = true;
//nt1.IsBackground = true;
//nt2.IsBackground = false;
   nt1.Start();
```

```
   nt2.Start();
Console.ReadLine();
}
 }
}
```

Рабрта программ:

```
Работает нить 1 :
0 Работает нить 2 :
10 1 11 2 3 12 4 5 13 6 7 14 8 9 15
16 17 18 19

Работает нить 1 :
0 Работает нить 2 :
10 1 11 2 3 12 4 5 13 6 7 14 8 9
```

В первом случае, когда `nt1.IsBackground = true;` - первая нить фоновая, программа заканчивает свою работу по окончании работы второй нити. Во втором случае (фоновой является вторая нить) программа заканчивает свою работу до завершения работ второй нити. «Когда фоновый поток завершается таким способом, все блоки **finally** внутри потока игнорируются. Поскольку невыполнение кода в **finally** обычно нежелательно, будет правильно ожидать завершения всех фоновых потоков перед выходом из программы, назначив нужный таймаут (при помощи **Thread.Join**). Если по каким-то причинам рабочий поток не завершается за выделенное время, можно попытаться аварийно завершить его (**Thread.Abort**).

Превращение рабочего потока в фоновый может быть последним шансом завершить приложение, так как не умирающий основной поток не даст приложению завершиться. Зависший основной поток особенно коварен в приложениях WindowsForms, так как приложение завершается, когда завершается его главный поток (по крайней мере, для пользователя), но его процесс продолжает выполняться. В диспетчере задач оно исчезнет из списка приложений, хотя имя его исполняемого файла останется в списке исполняющихся процессов. Пока пользователь не найдет и не прибьет его, процесс продолжит потреблять ресурсы и, возможно, будет препятствовать запуску или нормальному функционированию вновь запущенного экземпляра приложения».

4.3.2 Понятие приоритета нити

При определении характеристик нити можно задавать приоритет нити с помощью свойства Priority. В языке C# существует 5 градаций приоритета, которые можно задавать нити: Lowest, BelowNormal, Normal, AboveNormal, Highest. Приоритет нити определяет, сколько времени на выполнение выделяется нити относительно других нитей. Для исследований приоритетов нитей используем предыдущую программу с двумя основными нитями, но с разными приоритетами.

Исходный код программы:

```
using System;
using System.Diagnostics;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
publicstaticvoid Niti_1()
  {
Console.WriteLine("Работаетнить 1 :");
for (int i = 0; i < 10; i++)
   {
Console.Write(i + " ");
   }
Console.WriteLine();
  }
publicstaticvoid Niti_2()
  {
Console.WriteLine("Работаетнить 2 :");
for (int i = 10; i < 20; i++)
   {
Console.Write(i + " ");
   }
Console.WriteLine();
  }
staticvoid Main(string[] args)
  {
Thread nt1 = newThread(Niti_1);
Thread nt2 = newThread(Niti_2);
   nt1.Priority = ThreadPriority.BelowNormal;
   nt2.Priority = ThreadPriority.BelowNormal;
   nt1.Start();
   nt2.Start();
Console.ReadLine();
}
 }
}
```

Работа программы при различных значениях приоритетов нитей:

A)

```
nt1.Priority = ThreadPriority.AboveNormal;
nt2.Priority = ThreadPriority.Normal;
```

Работает нить 1 :
Работает нить 2 :
10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9

B)

```
nt1.Priority = ThreadPriority.Highest;
nt2.Priority = ThreadPriority.Lowest;
```

Работает нить 1 :
0 1 2 3 4 5 6 7 8 9
Работает нить 2 :
10 11 12 13 14 15 16 17 18 19
C)

```
nt1.Priority = ThreadPriority.BelowNormal;
nt2.Priority = ThreadPriority.BelowNormal;
```

Работает нить 2 :
10 11 12 13 Работает нить 1 :
0 1 2 3 4 5 14 15 16 17 18 19
6 7 8 9

Соотношение приоритета и времени выполнения нити?

4.3.3 Определение процесса

Процессом в Windows называется объект ядра, которому принадлежат системные ресурсы, используемые в исполняемом приложении. Обычно говорят, что процесс это исполняемое приложение.

Выполнение каждого процесса начинается с выполнения главной нити – для консольных приложений это метод Main.

Во время выполнения процесса могут создаваться и другие нити процесса или даже другие процессы.

Процесс заканчивается при завершении работы всех его нитей.

Каждый процесс в Windows владеет следующими системными ресурсами:

- виртуальным адресным пространством;
- маркером доступа с информацией системы безопасности;
- таблицей для хранения дескрипторов объектов ядра.

По аналогии с нитью каждый процесс имеет свой идентификатор, который используется служебными программами.

4.3.4 Создание процесса из работающего приложения

Для запуска приложения (создание процесса) из уже рабочего приложения необходимо использовать класса Process.

Класс Process имеет набор свойств и методов, которые представлены в приложении этой лекции. Перечень получен с помощью справки среды при выделении слова `Process` и нажатии кнопки F1.

Для успешного запуска процесса к приложению необходимо подключить пространство имен System.Diagnostics.

Класс Process можно использоваться для запуска процессов путем вызова метода Process.Start(FileName) или Process.Start().

Перед запуском метода Process.Start(FileName) необходимо в качестве параметра метода задать имя файла процесса (свойстваFileName) для запуска или полный путь к нему.

Можно создать объект нового процесса, задать значение свойству FileName и запустить процесс с помощью метода Process.Start(), без параметров.

Предложенные варианты запуска реализованы в следующей учебной программе:

```
using System;
using System.Diagnostics;
namespace ConsoleApplication1
{
classsoz_pro
 {
staticvoid Main()
  {
Process.Start("sol.exe");
Console.ReadLine();
Process myProcess = newProcess();
   myProcess.StartInfo.FileName = "Notepad";
   myProcess.Start();
Console.ReadLine();
  }
 }
}
```

Работа программы заключается в запуске игры «Пасьянс Косынка». По окончании игры необходимо нажать клавишу «Enter» и запустить редактор «Блокнот». После окончания работы с блокнотом необходимо закрыть консольное окно нажатием клавиши «Enter».

Можно использовать методы класса Process, например, GetProcesses для исследования характеристик работающего процесса или процессов.

В следующей учебной программе просматриваются некоторые характеристики первого процесса на компьютере работающего на текущий момент:

```
using System;
using System.Diagnostics;
using System.ComponentModel;

namespace ConsoleApplication1
{
classsoz_pro
 {
staticvoid Main()
  {
foreach (Process p inProcess.GetProcesses())
   {
Console.WriteLine("Name:         " + p.ProcessName);
```

```
Console.WriteLine("PID:          " + p.Id);
Console.WriteLine("Started:      " + p.StartTime);
Console.WriteLine("Memory:       " + p.WorkingSet64);
Console.WriteLine("CPU time:     " + p.TotalProcessorTime);
Console.WriteLine("Threads:      " + p.Threads.Count);
Console.WriteLine(); break;
    }
Console.ReadLine();
}
  }
}
```

Работа программы:

```
Name:       dwm
PID:        1292
Started:    31.07.2011 7:07:47
Memory:     31358976
CPU time:   00:00:03.2604209
Threads:    6
```

5 СИНХРОНИЗАЦИЯ РАБОТЫ НИТЕЙ В ЯЗЫКЕ C#

5.1 Синхронизация нитей

5.1.1 Понятие действия

Действием называется любая последовательность команд, которая изменяет контекст нити. Определение контекста нити из 2 лекции – это область памяти компьютера, к которой может обращаться нить во время своего исполнения.

Действие называется непрерывным, если оно не прерывается во время своего исполнения и может быть изменено только самим действием.

Действие может быть прервано только сигналом прерывания микропроцессора.

Первое требование справедливо только для однопроцессорных систем, в которых можно даже запрещать прерывание на время выполнения действия.

Для мультипроцессорных систем действия могут выполняться параллельно разными нитями и даже пересекаться. Поэтому для мультипроцессорных систем непрерывность действия обеспечивается запретом действию, исполняемому на одном процессоре, изменять действия, исполняемого на другом процессоре.

Таким образом, в мультипроцессорных системах важнейшей задачей обеспечения непрерывности действия является согласование работы параллельных нитей.

Одним из вариантов решения этой задачи является синхронизация работы нитей.

В общем случае под синхронизацией нитей понимается достижение некоторого фиксированного порядка их выполнения с помощью сигналов, которыми нити обмениваются между собой.

Синхронизация это координация действий нитей для получения предсказуемого результата [Албахари стр.739.]

5.1.2 Классификация средств синхронизации нитей

В этой лекции рассмотрим некоторые теоретические аспекты синхронизации на примерах синхронизации нитей в одном процессе.

Обычно основной метод программы (Main) называется главной нитью, а методы решающие некоторые задачи для главной нити называют дополнительными или вторичными нитями.

Естественно работа этих нитей должна быть упорядочена. Кроме упорядочения работы нитей реальные (не учебные) программы должны решать задачи совместного использования некоторых данных (отдельные вопросы передачи данных нитям и между нитями мы рассматривали в предыдущих лекциях), но для изучения теории синхронизации целесообразно рассматривать эти задачи по отдельности.

«Все средства синхронизации условно можно разделить на четыре категории:

- простые средства синхронизации нитей, к которым относятся глобальные переменные и простые методы приостановки выполнения нити (Sleep иJoin);
- специальные блокирующие конструкции, к которым относятся объекты синхронизации (Mutexи Semaphore) и оператор lock;
- конструкции, основанные на подаче и приеме специальных сигналов. Эти конструкции приостанавливают выполнения нити до получения сигнала от другой нити. Обычно используются две подобные конструкции, использующие сигнальный механизм, это дескрипторы ожидания событий и методы Wait/Pulse класса Monitor;
- средства, не требующие приостановки выполнения нитей (незадерживающие средства синхронизации), к которым относятся класс Interlockedи специальное ключевое слово volatile, позволяющее запрещать кэширование операций чтения – записи данных.»[Албахари стр. 739-740].

Кроме перечисленных средств синхронизации нитей существует вариант автоматической синхронизации (блокировки) доступа к ресурсам нити, который реализуется классом ContextBoundObject и атрибутом Synchronization, которые мы будем называть средствами пятой категории.

Выбор того или иного средства синхронизации определяется решаемой задачей и зависит от программиста.

Начнем изучение средств синхронизации с наиболее простых категорий.

В одной из своих статей, опубликованных в Интернете, Албахари приводит следующий перечень средств синхронизации нитей (переводчик статьи называет их потоками):

«Важнейшие средства синхронизации

В следующих таблицах приведена информация об инструментах .NET для координации (синхронизации) потоков:

Простейшие методы блокировки

Конструкция	**Назначение**
Sleep	Блокировка на указанное время
Join	Ожидание окончания другого потока

Блокировочные конструкции

Конструкция	**Назначение**	**Доступна из других процессов?**	**Скорость**
Lock	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода.	нет	быстро
Mutex	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода. Может использоваться для предотвращения запуска нескольких экземпляров приложения.	да	средне
Semaphore	Гарантирует, что не более заданного числа потоков может получить доступ к ресурсу или секции кода.	да	средне

(Для автоматической блокировки также могут использоваться контексты синхронизации.)

Конструкция	**Назначение**	**Доступна из других процессов?**	**Скорость**
EventWaitHandle	Позволяет потоку ожидать сигнала от другого потока.	да	средне

WaitandPulse*	Позволяет потоку ожидать, пока не выполнится заданное условие блокировки.	нет	средне

Сигнальные конструкции

Конструкция	Назначение	Доступна из других процессов?	Скорость
Interlocked*	Выполнение простых не блокирующих атомарных операций.	Да – через разделяемую память	очень быстро
volatile*	Для безопасного не блокирующего доступа к полям.	Да – через разделяемую память	очень быстро

»

5.1.3 Простые средства синхронизации нитей

Традиционно, при рассмотрении теории синхронизации нитей, для визуального контроля очередности их работы используется циклический вывод некоторых различных значений, например, различных символов или чисел. Не будем отступать от традиции, и работа наших нитей будет заключаться только в выводе числовых значений в консольное окно экрана монитора (реальные нити выполняют работу для главной нити).

Предположим, что кроме главной нити наша программа содержит две дополнительные. Пусть главная нить выводит в консольное окно числа от 0 до 9, первая дополнительная нить – числа от 10 до 19, а вторая – числа от 20 до 29 (в этом их работа).

Рассмотрим работу нескольких учебных программ, в которых нити не синхронизированы, синхронизированы с помощью глобальной переменно и синхронизированы с помощью методов **Sleep, Join**и **lock**– средства синхронизации нитей одного процесса.

В программе с не синхронизированными нитями первая нить объявлена внутри своего класса, а вторая – представлена статическим методом.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classNot_Cinxr
    {
```

```
classPervaj_1
        {
publicvoid Niti_1()
            {
for (int i = 10; i < 20; i++)
Console.Write(i + " ");
            }
        }
staticvoid Niti_2()
        {
for (int i = 20; i < 30; i++)
Console.Write(i + " ");
        }
staticvoid Main()
        {
Pervaj_1 Pe = newPervaj_1();
Thread t1 = newThread(Pe.Niti_1);
            t1.Start();
Thread t2 = newThread(Niti_2);
            t2.Start();
for (int i = 0; i < 10; i++)
Console.Write(i + " ");
Console.ReadLine();
}
    }
}
```

Работа программы:

0 20 21 10 11 12 13 14 15 16 17 18 19 1 2 3 4 5 6 7 8 9 22 23 24 25 26 27 28 29

Работа нитей не синхронизирована – вывод осуществляется «вперемешку».

5.1.3.1 Использование глобальной переменной

Рассмотрим пример синхронизации нитей с помощью глобальной переменной **Flag**. В работу запускаются все нити, но «тела» нитей блокируются операторами **goto** и **if**, которые «ждут» разрешение на работу нити от глобальной переменной.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classCin_Gl
    {
publicstaticint flag = 0;
classPervaj_1
        {
publicvoid Niti_1()
            {
               m1: if (flag == 0)
                 {
for (int i = 10; i < 20; i++)
```

```
Console.Write(i + " ");
                    flag++;
                } elsegoto m1;
            }
        }
staticvoid Niti_2()
        {
          m1:  if (flag == 2)
            {
for (int i = 20; i < 30; i++)
Console.Write(i + " ");
                flag++;
            } elsegoto m1;
        }
staticvoid Main()
        {
Pervaj_1 Pe = newPervaj_1();
Thread t1 = newThread(Pe.Niti_1);
            t1.Start();
Thread t2 = newThread(Niti_2);
            t2.Start();
        m1: if (flag == 1)
            {
for (int i = 0; i < 10; i++)
Console.Write(i + " ");
                flag++;
            }
elsegoto m1;
Console.ReadLine();
}
    }
}
```

Работа программы:

10 11 12 13 14 15 16 17 18 19 0 1 2 3 4 5 6 7 8 9 20 21 22 23 24 25 26 27 28 29

Сначала работает первая нить (**Flag**= 0), затем главная нить (**Flag**= 1) и затем работает вторая нить (**Flag**= 2).

5.1.3.2 Использование метода Sleep

Рассмотрим пример синхронизации нитей с помощью метода Thread.Sleep(), который блокирует текущую нить на заданное количество миллисекунд.

Метод Thread.Sleep(0) – отказывается от выделенного кванта времени.

Метод Thread.Sleep(100) – блокирует выполнение нити на 100 миллисекунд – «спать» 100 миллисекунд.

```
Thread.Sleep(0);   // отказаться от одного кванта времени CPU
Thread.Sleep(1000);                  // заснуть на 1000 миллисекунд
Thread.Sleep(TimeSpan.FromHours(1));  // заснуть на 1 час
```

```
Thread.Sleep(Timeout.Infinite);        // заснуть до прерывания
```

Необходимо отметить, что после использования блокировочных конструкций нить перестает получать кванты времени до окончания блокировки.

В примере реализована блокировка нитей после каждого вывода в консольное окно программы.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classCin_sleep
    {
publicstaticint flag = 0;
classPervaj_1
        {
publicvoid Niti_1()
            {
for (int i = 10; i < 20; i++)
                { Console.Write(i + " "); Thread.Sleep(1); }
            }
        }
staticvoid Niti_2()
        {
for (int i = 20; i < 30; i++)
                { Console.Write(i + " "); Thread.Sleep(1); }

        }
staticvoid Main()
        {
Pervaj_1 Pe = newPervaj_1();
Thread t1 = newThread(Pe.Niti_1);
            t1.Start();
Thread t2 = newThread(Niti_2);
            t2.Start();
for (int i = 0; i < 10; i++)
            { Console.Write(i + " "); Thread.Sleep(1); }

Console.ReadLine();
        }
    }
}
```

Работа программы:

0 20 10 21 1 11 2 12 22 13 23 3 4 24 14 25 5 15 6 26 16 27 17 7 18 28 8 19 9 29

В процессе проверки программы задержки метода Sleep() изменялись от 1 до 150 миллисекунд. Во всех результатах работы программы вывод осуществлялся «тройками», но внутри «тройки» очередность не соблюдалась.

Т.о. после очередного вывода нить на некоторое время «засыпала» разрешая работу (только один вывод) другой нити.

5.1.3.3 Использование метода Join

Рассмотрим пример синхронизации нитей с помощью метода Join(), который блокирует вызывающую (следующую) нить до завершения текущей (предыдущей) нити.

Существует три варианта использования методаJoin():

	Имя	Описание
	Join()	Блокирует вызывающий поток до завершения потока, продолжая отправлять стандартные сообщения COM и SendMessage.
	Join(Int32)	Блокирует вызывающий поток до завершения потока или истечения указанного времени, продолжая отправлять стандартные сообщения COM и SendMessage.
	Join(TimeSpan)	Блокирует вызывающий поток до завершения потока или истечения указанного времени, продолжая отправлять стандартные сообщения COM и SendMessage.

В примере реализована «блокировка» первого варианта для первой и второй нитей.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classCin_sleep
    {
publicstaticint flag = 0;
classPervaj_1
        {
publicvoid Niti_1()
            {
for (int i = 10; i < 20; i++)
                { Console.Write(i + " "); Thread.Sleep(10); }
            }
        }
```

```
staticvoid Niti_2()
        {
for (int i = 20; i < 30; i++)
                { Console.Write(i + " "); Thread.Sleep(10); }

        }
staticvoid Main()
        {
Pervaj_1 Pe = newPervaj_1();
Thread t1 = newThread(Pe.Niti_1);
            t1.Start();
            t1.Join();
Thread t2 = newThread(Niti_2);
            t2.Start();
            t2.Join();
for (int i = 0; i < 10; i++)
            { Console.Write(i + " "); Thread.Sleep(10); }

Console.ReadLine();
        }
    }
}
```

Работа программы – только **`t1.Join();`**:
10 11 12 13 14 15 16 17 18 19 0 20 1 21 22 2 23 3 4 24 25 5 6 26 27 7 28 8 9 29
Работа программы – только**`t2.Join();`**:
10 20 11 21 12 22 13 23 14 24 15 25 16 26 17 27 18 28 29 19 0 1 2 3 4 5 6 7 8 9
Работа программы – и **`t1.Join();`** и **`t2.Join();`**:
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 0 1 2 3 4 5 6 7 8 9

В первом случае первая нить «блокирует» выполнение второй и главной нитей, которые по окончании блокировки работают не синхронно.

Во втором случае «блокируется» только главная нить, а первая и вторая нити работают не синхронно.

В третьем случае сначала первая нить «блокирует» выполнение второй и главной нитей, а затем вторая нить «блокирует» главную нить.

Данное средство блокировки позволяет упорядочить работу нитей.

5.1.3.4 Использование оператора **lock**

Оператор **lock** относятся к специальным блокирующим конструкциям вместе с объектами синхронизации **Mutex** и **Semaphore.**

Оператор **lock** позволяет блокировать работу нитей только внутри одного процесса, в отличие от объектов синхронизации **Mutex** и **Semaphore**, которые могут блокировать нити разных процессов. Поэтому работу оператора **lock** мы рассмотрим в этой лекции.

В предыдущей лекции мы уже рассматривали работу оператора **lock** для передачи данных нитям, в этом примере мы запустим три нити, для вычисления выражения y = a*x + b/x + c. Первая нить получает значение си ждет результатов вычислений a*xи b/x, которые выполняются во второй и третьей нитях соответственно. Переменная x объявляется в программе как глобальная переменная.

Реализацию всей программы в учебных целях выполним в виде 4 файлов – файла программы и трех файлов классов, для методов которых будут запускаться нити.

Исходный код первого файла – программы имеет следующий вид:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
    {
publicstaticdouble rez,x;

staticvoid Main()
        {
Console.WriteLine("Введите x");
            x = double.Parse(Console.ReadLine());
Klass_1 Thr1 = newKlass_1();
            Thr1.thread1.Join();

Console.ReadKey();
        }
    }
}
```

Для создания и подключения файла Klass_1 к проекту необходимо в режиме Project выбрать команду Add Class и в появившемся окне указать имя создаваемого файла – Klass_1. Среда создаст файл Klass_1.cs подключит его к проекту и перейдет в режим редактирования кода программы файла (смотри рисунок 5.1.1).

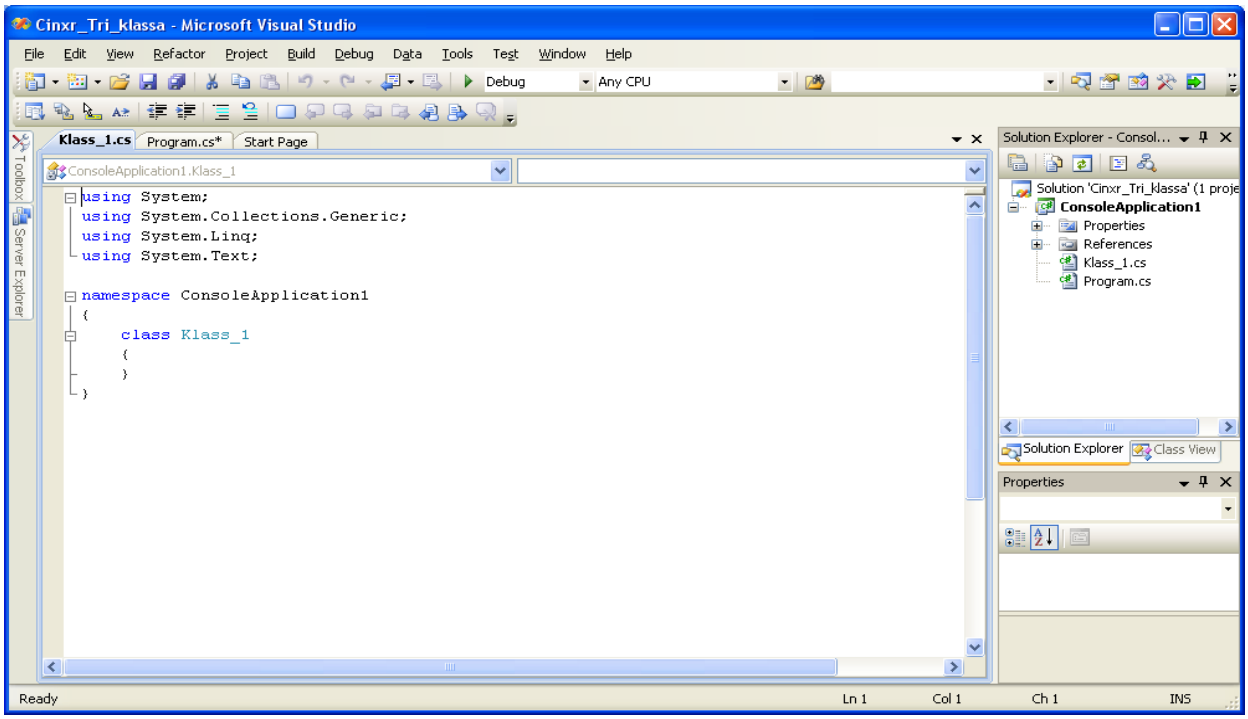


Рисунок 5.1.1 – Создание и подключение файла Klass_1.cs к проекту

В файле Klass_1.cs мы вводим в режиме диалога значение переменной **c** и запускаем в работу вторую нить проекта, результат работы которой мы будем использовать для вычисления всего выражения.
Файл Klass_1.cs имеет следующий исходный код:

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
classKlass_1
 {
privatedouble c;
publicThread thread1;
public Klass_1()
  {
   thread1 = newThread(Rab);
   thread1.Start();
  }
publicvoid Rab()
  {
lock (this)
   {
Console.WriteLine("Нить 1 стартовала");
Console.WriteLine("Введите C");
    c = double.Parse(Console.ReadLine());
Klass_2 Thr2 = newKlass_2();
    Thr2.thread2.Join();
Program.rez = Program.rez + c;
```

```
Console.WriteLine("Нить 1 завершила работу. Результат =" +
Program.rez);
    }
   }
  }
}
```

Из кода видно, что для завершения работы 1 нити необходим результат работы 2 нити (работающей с методом второго класса). Создадим файл Klass_2.cs и подключит его к проекту.

В файле Klass_2.cs мы вводим в режиме диалога значение переменной b, вычисляем b/x и запускаем в работу третью нить проекта, результат работы которой мы будем использовать для вычисления всего выражения. Файл Klass_2.cs имеет следующий исходный код:

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
classKlass_2
 {
privatedouble b;
publicThread thread2;
public Klass_2()
  {
   thread2 = newThread(Rab);
   thread2.Start();
  }
publicvoid Rab()
  {
lock (this)
   {
Console.WriteLine("Нить 2 стартовала");
Console.WriteLine("Введите B");
    b = double.Parse(Console.ReadLine());
Klass_3 Thr3 = newKlass_3();
    Thr3.thread3.Join();
Program.rez = Program.rez + b/Program.x;
Console.WriteLine("Нить 2 завершила работу. Результат =" +
Program.rez);
   }
  }
 }
}
```

В файле Klass_3.cs мы вводим в режиме диалога значение переменной a, вычисляем a * x и заканчиваем работу третьей нити проекта. Промежуточный результат ее работы выводим на экран монитора.

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
classKlass_3
```

```
 {
privatedouble a;
publicThread thread3;
public Klass_3()
  {
   thread3 = newThread(Rab);
   thread3.Start();
  }
publicvoid Rab()
  {
lock (this)
   {
Console.WriteLine("Нить 3 стартовала");
Console.WriteLine("Введите A");
    a = double.Parse(Console.ReadLine());
Program.rez = Program.rez + a * Program.x;
Console.WriteLine("Нить 3 завершила работу. Результат =" +
Program.rez);
   }
  }
 }
}
```

Работа программы:

```
Введите x
2,5
Нить 1 стартовала
Введите C
3
Нить 2 стартовала
Введите B
5
Нить 3 стартовала
Введите A
3
Нить 3 завершила работу! Значение равно: 7,5
Нить 2 завершила работу! Значение равно: 9,5
Нить 1 завершила работу! Значение равно: 12,5
```

По результатам работы программы видно, что после ввода значения переменной x запускается первая нить, затем вторая нить и затем третья нить. Завершение работы нитей осуществляется в обратном порядке. Синхронизация работы нитей достигается с помощью оператора lock и запуском очередной нити из нитевой функции предыдущей нити.

5.2 Специальные блокирующие конструкции

5.2.1 Специальная блокирующая конструкция **Mutex**

По определению класс **Mutex** выполняет те же функции, что и оператор **lock**, он разрешает доступ к ресурсу или секции кодатолько одной нитей. Если синхронизация нитей выполняется в одном процессе, то функциональных отличий между ними нет. Но для класса **Mutex** необходимо создать объект. Также считается, что объект **Mutex** работает медленнее, чем оператор **lock**.

Основным достоинством класса **Mutex** является то, что его объект может использоваться для синхронизации нитей разных процессов или для предотвращения запуска нескольких экземпляров приложения. Говорят, что он решает проблемы «взаимного исключения» между параллельными нитями, выполняющимися в различных процессах

Как и всякий класс **Mutex** имеет свойства и методы (см. приложение в конце лекции).

Mutex можно создать с помощью следующих конструкторов:
static Mutex имя в программе = new Mutex();
static Mutex имя в программе = new Mutex(bool);
static Mutex имя в программе = new Mutex(bool, "Имя в ОС");

Первый создает безымянный мьютекс и делает текущую нить его владельцем, поэтому мьютекс блокируется текущей нитью.

Второй принимает только логический аргумент, который определяет, собирается ли создающая мьютекс нить завладеть им (заблокировать его). Если аргумент типа bool установить в false, то это означает, что мьютекс изначально заблокирован – принадлежит нити.

Первые два конструктора обычно применяются для синхронизации нитей в одном процессе. Для нас представляет интерес синхронизации нитей в разных процессах. Поэтому мы будем использовать третий вариант конструктора, позволяющей задавать имя **Mutex** для операционной системы. Например:

static Mutex kl = new Mutex(false, "Kljsik_1");

В качестве учебной программы рассмотрим вывод в консольное окно программы таблицы значений чисел от 0 до 9 для основной программы (сервером) и таблицы чисел от 10 до 19 дополнительной программы (клиентом), которая запускается из основной программы.

Исходный код основной программы:

```
usingSystem;
using System.Diagnostics;
using System.Threading;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
classsoz_pro
 {
staticvoid Prin(int n)
  {
Console.Write(n + " ");
Thread.Sleep(100);
```

```
  }
static Mutex kl = new Mutex(false, "Kljsik_1");
static void Main()
  {
string clientExe = "Dop_pr.exe";
var startInfo = new ProcessStartInfo(clientExe);
   startInfo.UseShellExecute = false;//водноконсольноеокно
Process p = Process.Start(startInfo);
for (int j = 0; j < 10; j++)
   {
       kl.WaitOne();
for (int i = 0; i < 10; i++)
       {
           Prin(j);
       }
Console.WriteLine();
       kl.ReleaseMutex();
   }
Console.ReadLine();
}
 }
}
```

Исходный код дополнительной программы:

```
usingSystem;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Prin(int n)
  {
Console.Write(n + " ");
Thread.Sleep(100);
  }
staticMutex kl_2 = newMutex(false, "Kljsik_1");
staticvoid Main()
  {
int i, j;
for (j = 10; j < 20; j++)
   {
    kl_2.WaitOne();
for (i = 0; i < 10; i++)
         Prin(j);
Console.WriteLine();
    kl_2.ReleaseMutex();
   }
Console.ReadKey();
  }
 }
}
```

Работа основной и дополнительной программ

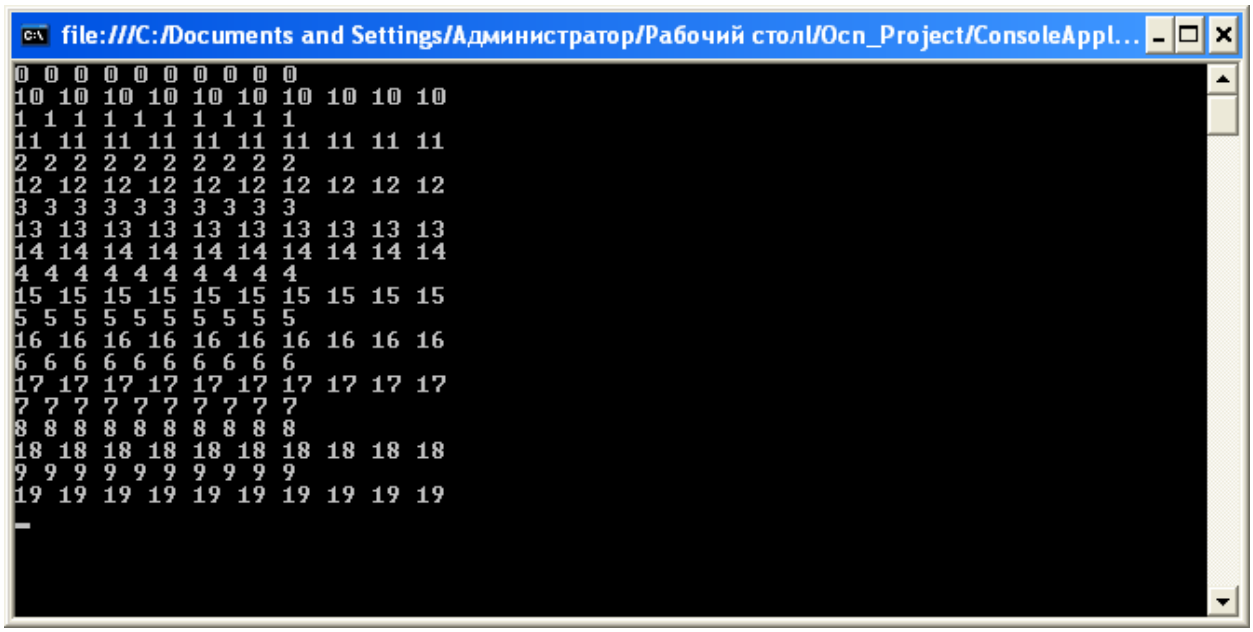


Рисунок 5.2.1 – Работа основной и дополнительной программ

Для нормальной работы программ файл `Dop_pr.exe`, должен находиться в папке с файлом `ConsoleApplication1.exe`. основной программы.

Работа каждой программы отображается в консольном окне. Программы синхронизированы с помощью мьютекса с именем "Kljsik_1" – программы выводят информацию по строкам. Прерывание строки недопустимо (за этим следит мьютекс). После вывода очередной строки обе программы могут захватить **Mutex** и продолжить вывод следующей строки.

При обозначении имени процесса использован тип var:

```
var startInfo = new ProcessStartInfo(clientExe);.
```

Такое задание типа переменным в языке C# возможно, если компилятор языка способен автоматически распознать тип по выражению инициализации, например, следующие записи инициализации строковой переменной эквивалентны:

string st1 = “Привет”;и

var st1 =“Привет”; .

5.2.2 Специальная блокирующая конструкция Semaphore

Следующей специальной блокирующей конструкцией, предназначенной для синхронизации процессов, является объект синхронизации Semaphore. Основная его особенность заключается в том, что он разрешает одновременное использование ресурса несколькими нитями как одного, так и нескольких процессов.

Semaphore можно создать с помощью следующих конструкторов:
static Semaphore имя в программе = new Semaphore(Сч, Max);
static Semaphore имя в программе = new Semaphore(Сч, Max, "ИмявОС");
где Сч – счетчик нитей, использующих ресурсы, защищаемые Semaphore, а Max – максимальное количество нитей, способных получить доступ к ресурсам, посредством Semaphore. Обычно в начальном состоянии счетчик нитей равен максимальному количеству нитей семафора. Например:

```
static Semaphores = new Semaphore(3, 3);
```

В этом примере создается объект stипа Semaphore, который разрешает работу с ресурсами одновременно только трем нитям.

Для получения доступа к ресурсам, защищаемых Semaphore, используется метод WaitOne(), который, если семафор находится в сигнальном состоянии – разрешает доступ к ресурсам, уменьшает счетчик нитей на единицу (если в методе WaitOne() не задано другое число). Если количество нитей, работающих с ресурсами, меньшее максимального количества нитей, определенного в объекте семафора (счетчик нитей больше или равен нулю), то семафор остается в сигнальном состоянии. Как только количество нитей, желающих работать с ресурсом, становится больше допустимого, семафор блокирует доступ к ресурсу. Разблокировать семафор можно методом Release(), который увеличивает счетчик нитей семафора на единицу (если в методе Release() не задано другое число), а это автоматически переводит семафор в сигнальное состояние и он становится доступным нитям.

Таким образом, объект семафор позволяет синхронизировать доступ к ресурсам программы для многих нитей.

В качестве примера синхронизации доступа к ресурсам программы многих нитей рассмотрим работу следующей программы, взятой из [Албахари стр.748].

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
staticSemaphore s = newSemaphore(3, 3);
staticvoid Main()
  {
for (int i = 0; i < 10; i++)
newThread(Go).Start(i);
Console.WriteLine("Работа цикла запуска нитей закончена!");
Console.ReadKey();
  }
staticvoid Go(object id)
  {
   s.WaitOne();
Console.WriteLine(id + " <- нитьначинаетработу!");
Thread.Sleep(1000 * (int)id);
```

```
Console.WriteLine(id + " ->нитьзакончилаработу");
s.Release();
  }
 }
}
```

Работа программы:

```
2 <- нить начинает работу!
4 <- нить начинает работу!
Работа цикла запуска нитей закончена!
5 <- нить начинает работу!
2 -> нить закончила работу
9 <- нить начинает работу!
4 -> нить закончила работу
0 <- нить начинает работу!
0 -> нить закончила работу
6 <- нить начинает работу!
5 -> нить закончила работу
1 <- нить начинает работу!
1 -> нить закончила работу
3 <- нить начинает работу!
3 -> нить закончила работу
8 <- нить начинает работу!
6 -> нить закончила работу
7 <- нить начинает работу!
9 -> нить закончила работу
7 -> нить закончила работу
8 -> нить закончила работу
```

Из результатов работы программы видно, то одновременно с ресурсами программы могут работать только три нити. Объект семафор «превращается» в мьютекс, если счетчик нитей и максимальное количество нитей задать равными единице. В остальных ситуациях объект семафор не имеет аналогов.

Для демонстрации работы объекта семафор с разными процессами используем следующую чисто учебную программу, в которой программа будет запускать сама себя до тех пор, пока не исчерпаны возможности семафора.

```
using System;
using System.Diagnostics;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
staticSemaphore s = newSemaphore(5, 5,"sem_1");
staticvoid Main()
  {
int kol;
   s.WaitOne(); kol = s.Release();
if(kol>=1)
```

```
    {
Process.Start("ConsoleApplication1");
     s.WaitOne();
Console.WriteLine(kol + " <- Процессначинаетработу!");
Thread.Sleep(1000*5);
Console.WriteLine(kol + " ->Процессзакончилработу");
     s.Release();
    } elseConsole.WriteLine(" семафорзакрыт ");
Console.ReadKey();
   }
  }
}
```

Работа программы:

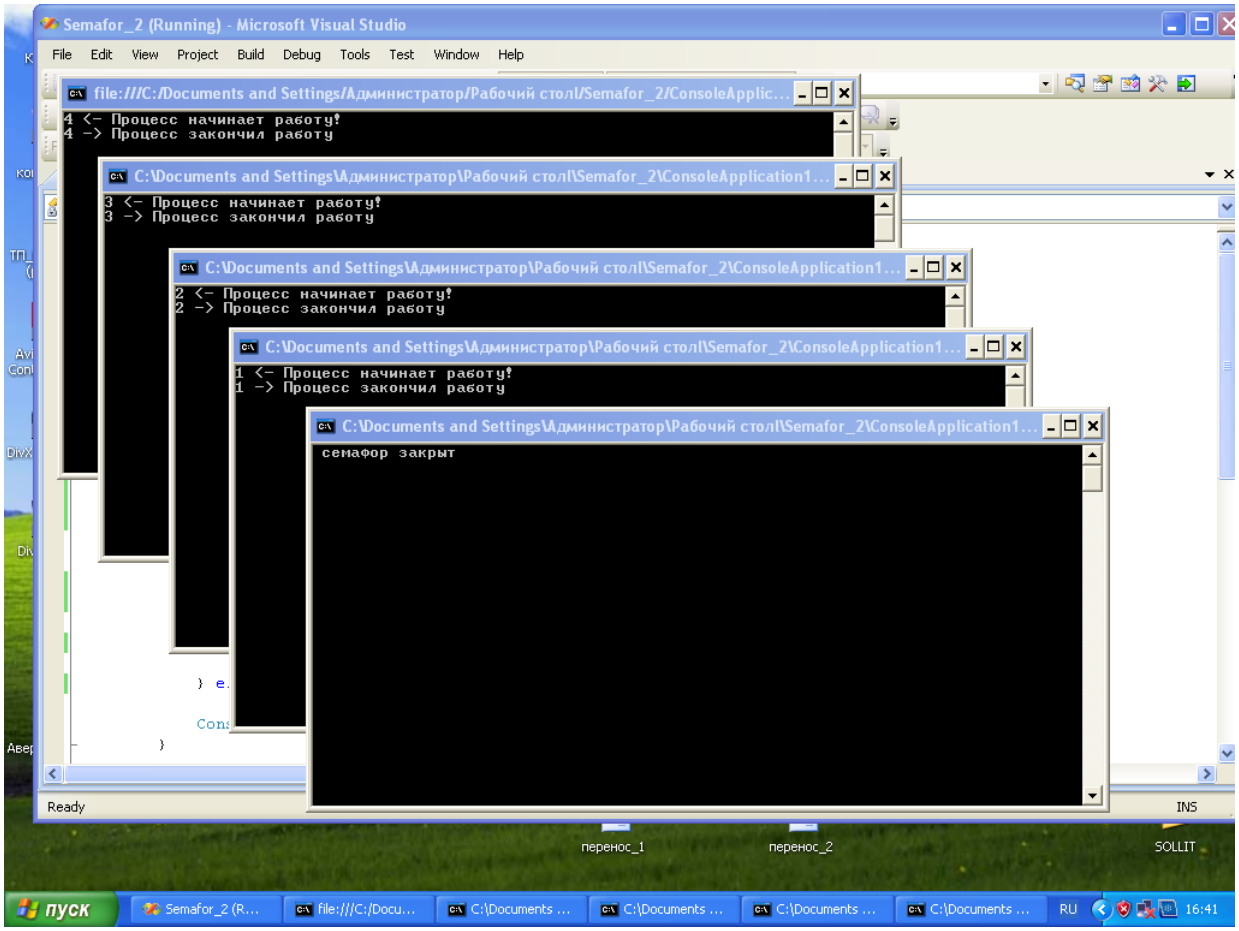


Рисунок 5.2.1 – Распределение ресурсов с помощью Semaphore

В книге [Албахари стр.747] работа объекта семафора очень образно сравнивается с работой ночного клуба.

«Semaphore похож на ночной клуб – он имеет определенную вместимость, которую обеспечивает вышибала. После заполнения никто уже не может войти в ночной клуб, очередь образуется снаружи. Далее, если один человек покидает клуб, один из начала очереди может пройти внутрь.

Конструктор Semaphore принимает минимум два параметра – число еще доступных мест и общую вместимость ночного клуба».

5.3 Автоматическая синхронизация нитей

5.3.1 Понятие контекста синхронизации нитей

Синхронизация нитей некоторого процесса направлена на обеспечения максимальной загрузки компьютера. Часто синхронизация работы нитей отождествляется с организацией параллельной работы нитей. Это не так, хотя присутствуют некоторые общие моменты организации вычислительного процесса – например, организация нитей. Синхронизация нитей особенно важна, когда нити обращаются к одним и тем же данным, например, при работе с базой данных. Параллельная работа нитей, например, веб-сервер, нити которого могут обрабатывать тысячи запросов, не предполагает использования общих данных.

Если создать объект, а это область памяти с некоторыми данными и действиями (контекст объекта), и блокировать доступ к нему (к области памяти), до завершения работы очередной нити, то можно организовать автоматическую синхронизацию (точнее блокировку) нитей.

В языке C# существует специальный класс ContextBoundObject, наследуя который можно создавать контексты объектов с определенными наборами методов и свойств. Для автоматической синхронизации (блокировки) доступа к контексту объекта необходимо перед объявлением такого объекта указать атрибут Synchronization, тогда область действия такого объекта будет называться контекстом синхронизации. Например:

```
[Synchronization]
publicclassNiti : ContextBoundObject
        { . . .}
```

Область действия этого объекта является контекстом синхронизации. Создав контекст синхронизации можно управлять доступом (блокировать) нескольких работающих нитей к ресурсам этого объекта – т.е. в процессе может работать несколько нитей, но в конкретный момент времени к ресурсам объекта допущена только одна нить. При этом, пока нить работающая с контекстом синхронизации не закончит работу, другая нить не получит доступа. Это исключает одновременную работу с объектом нескольких работающих нитей.

5.3.2 Пример использования контекста синхронизации несколькими нитями

В качестве учебного примера рассмотрим одновременную работу четырех нитей с методом, позволяющем выводить в консольное окно значения управляющей переменной цикла for.

Код учебной программы:

```
usingSystem;
using System.Threading;
using System.Runtime.Remoting.Contexts;

namespace ConsoleApplication1
{
[Synchronization]
publicclassNiti : ContextBoundObject
 {
publicvoid Cikl(object ob)
  {
int k = (int)ob;
   Console.Write("Работаетнить {0}  ",k);
for (int i = k * 10; i <= k * 10 + 9; i++)
   {
Console.Write("  " + i);
Thread.Sleep(100);
   }
Console.WriteLine();
  }
 }
classProgram
 {
staticvoid Main(string[] args)
  {
int i=0;
Niti ob_1 = newNiti();
   i++;
   Console.WriteLine("Запущенанить {0}  ", i);
   Thread th1 =
newThread(newParameterizedThreadStart(ob_1.Cikl));
   th1.Start(i);
   i++;
   Console.WriteLine("Запущенанить {0}  ", i);
Thread th2 = newThread(newParameterizedThreadStart(ob_1.Cikl));
   th2.Start(i);
   i++;
   Console.WriteLine("Запущенанить {0}  ", i);
Thread th3 = newThread(newParameterizedThreadStart(ob_1.Cikl));
   th3.Start(i);
   i++;
   Console.WriteLine("Запущенанить {0}  ", i);
   ob_1.Cikl(i);
Console.ReadLine();
}
 }
}
```

Работа программы:

```
Запущена нить 1
Запущена нить 2
Запущена нить 3
Запущена нить 4
```

```
Работает нить 2    20  21  22  23  24  25  26  27  28  29
Работает нить 1    10  11  12  13  14  15  16  17  18  19
Работает нить 4    40  41  42  43  44  45  46  47  48  49
Работает нить 3    30  31  32  33  34  35  36  37  38  39
```

По распечатки видно, что в нашем процессе запущены все четыре нити, но если контекст синхронизации захватывает некоторая нить, то доступ другой нити предоставляется только по окончании всей работы предыдущей нити.

Автоматическая синхронизация не может быть использована для защиты членов статических типов и классов, не являющихся наследниками **ContextBoundObject**(например, WindowsForm).

5.3.3 Понятие взаимоблокировки

Контекст синхронизации может распространяться за пределы одиночного объекта. По умолчанию, если синхронизированный объект создается в коде другого объекта, оба разделяют один контекст (другими словами, одну большую блокировку!) Это поведение можно изменить, задавая флаг в конструкторе атрибута **Synchronization** с использованием констант, определенных в классе **SynchronizationAttribute:**

Константа	Значение
NOT_SUPPORTED	Эквивалентно неиспользованию атрибутов синхронизации.
SUPPORTED	Присоединиться к существующему контексту синхронизации, если создание происходит из синхронизированного объекта, иначе не использовать синхронизацию.
REQUIRED(default)	Присоединиться к существующему контексту синхронизации, если создание происходит из синхронизированного объекта, иначе создать новый контекст.
REQUIRES_NEW	Всегда создавать новый контекст синхронизации.

Так, если объект класса **SynchronizedA** создает объект класса **SynchronizedB**, у них будут разные контексты синхронизации, если **SynchronizedB** декларирован следующим образом:

```
[Synchronization(SynchronizationAttribute.REQUIRES_NEW)]
publicclassSynchronizedB : ContextBoundObject
{ . . . }
```

Чем больше расширяется контекст синхронизации, тем проще управление, но и тем меньше возможностей для полезного параллелизма. С

другой стороны, отдельные контексты синхронизации чреваты взаимоблокировками. Вот пример:

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;

namespace ConsoleApplication1
{
 [Synchronization]
publicclassNiti : ContextBoundObject
 {
publicNiti ob_ni;
publicvoid Zapl(object ob)
  {
int k = (int)ob;
Console.WriteLine("Работаетнить {0}  ", k);
   ob_ni.Cikl1(k);
  }
void Cikl1(int k)
  {
for (int i = k * 10; i <= k * 10 + 9; i++)
   {
Console.Write("  " + i);
Thread.Sleep(100);
   }
Console.WriteLine();
  }
 }

 classProgram
 {
staticvoid Main()
  {
int i=1;
Niti Ni_1 = newNiti();
Niti Ni_2 = newNiti();
   Ni_1.ob_ni = Ni_2;
Ni_2.ob_ni = Ni_1;
Thread th1 = newThread(newParameterizedThreadStart(Ni_1.Zapl));
   th1.Start(i);
   i++;
Thread th2 = newThread(newParameterizedThreadStart(Ni_2.Zapl));
   th2.Start(i);
Console.ReadLine();
}
 }
}
```

Работа программы:

Работает нить 1
Работает нить 2

Поскольку каждый экземпляр Niti создается внутри несинхронизированного классаProgram, каждый экземпляр получает свой контекст синхронизации, и, следовательно, свою собственную блокировку. Когда оба объекта вызывают методы друг друга,практически сразу возникает взаимоблокировка. В этом отличие контекстов от использования явных блокировок, где взаимоблокировки обычно более очевидны.

Если код метода `Cikl1`разместить внутри метода `Zapl`, то взаимоблокировки вообще нет, например:

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;

namespace ConsoleApplication1
{
 [Synchronization]
 publicclassNiti : ContextBoundObject
 {
publicNiti ob_ni;
publicvoid Zapl(object ob)
  {
int k = (int)ob;
Console.WriteLine("Работаетнить {0}  ", k);
for (int i = k * 10; i <= k * 10 + 9; i++)
   {
Console.Write("  " + i);
Thread.Sleep(100);
   }
Console.WriteLine();
  }
 }
 classProgram
 {
staticvoid Main()
  {
int i=1;
Niti Ni_1 = newNiti();
Niti Ni_2 = newNiti();
   Ni_1.ob_ni = Ni_2;
Ni_2.ob_ni = Ni_1;
Thread th1 = newThread(newParameterizedThreadStart(Ni_1.Zapl));
   th1.Start(i);
   i++;
Thread th2 = newThread(newParameterizedThreadStart(Ni_2.Zapl));
   th2.Start(i);
Console.ReadLine();
}
 }
}
```

Работа программы:

```
Работает нить 1
 10Работает нить 2
```

20 11 21 12 22 23 13 24 14 15 25 26 16 17 27 18 28 29 19

5.3.4 Сигнальная конструкция EventWaitHandle

В наборе функций Win32 API, предназначенных для создания объектов синхронизации системы Windows, наряду с мьютексами и семафорами используются функции, предназначенные для создания объектов синхронизации события (event).Этот объект синхронизации обладает простотой и удобством работы с нитями. Он используется для оповещения однойнити о некотором действии, которое совершила другая нить как в одном, так и в разных процессах. Естественно, в .NETFramework для управления (синхронизации) нитей, не могли не использовать достоинств этого объекта. Поэтому следующим классом сигнальных блокирующих конструкций по классификации за **Mutex** и **Semaphore**, которые мы рассмотрим в нашей лекции, является сигнальная конструкция на базе класса **EventWaitHandle.** Сигнальная конструкция, потому что позволяет организовать ожидание сигнал от другой нити одного или нескольких процессов.

Класс **EventWaitHandle** имеет два производных класса – **AutoResetEvent** и **ManualResetEvent** (еще раз напоминаем, что это блокирующая сигнальная конструкции, которые не имеющие никакого отношения к событиям и делегатам C#, рассмотренным ранее). Обоим классам доступны все функциональные возможности базового класса, единственное отличие состоит в вызове конструктора базового класса с разными параметрами.

«AutoResetEventочень похож на турникет – один билет позволяет пройти одному человеку. Приставка “auto” в названии относится к тому факту, что открытый турникет автоматически закрывается или “сбрасывается” после того, как позволяет кому-нибудь пройти. Поток блокируется у турникета вызовом **WaitOne** (ждать (*wait*) у данного (*one*) турникета, пока он не откроется), а билет вставляется вызовом метода **Set**. Если несколько потоков вызывают **WaitOne**, за турникетом образуется очередь. Билет может “вставить” любой поток – другими словами, любой (неблокированный) поток, имеющий доступ к объекту **AutoResetEvent,** может вызвать **Set,** чтобы пропустить один блокированный поток.

Если **Set** вызывается, когда нет ожидающих потоков, хэндл будет находиться в открытом состоянии, пока какой-нибудь поток не вызовет **WaitOne.** Эта особенность помогает избежать гонок между потоком, подходящим к турникету, и потоком, вставляющим билет (“опа, билет вставлен на микросекунду раньше, очень жаль, но вам придется подождать еще сколько-нибудь!”). Однако многократный вызов **Set** для свободного турникета не разрешает пропустить за раз целую толпу – сможет пройти только один человек, все остальные билеты будут потрачены впустую.

WaitOne принимает необязательный параметр **timeout** – метод возвращает **false**, если ожидание заканчивается по таймауту, а не по

получению сигнала. **WaitOne**также можно обучить выходить из текущего контекста синхронизации для продолжения ожидания (если используется режим с автоматической блокировкой) во избежание чрезмерного блокирования.

Метод **Reset** обеспечивает закрытие открытого турникета, безо всяких ожиданий и блокировок.» –ссылка из интернета.

Сравнение блокирующей конструкции AutoResetEvent с турникетом очень удачно, поэтому и приведена ссылка на интернет, однако необходимо отметить еще одно достоинство AutoResetEvent – его можно сделать именным. Т.е. за каждой нитью можно «закрепить» свое событие, которое и будет разрешать работу этой нити. Обычно при запуске приложения с помощью set() устанавливается необходимое событие, которое разрешает работу некоторой нити. По окончании работы этой нити необходимо «сбросить» событие Reset-ом и set-ом установить следующее событие, которое разрешает работу следующей нити и т.д.

В качестве учебного примера, регулирующего работу двух нитей с помощью двух блокирующих конструкций AutoResetEvent, предлагается программа, в которой создаются два объекта ev1 и ev2 типа EventWaitHandle. Объект ev1 предназначен для управления нити 1, а объект ev2 – для управления нитью 2. После запуска в работу обоих нитей (они обе ждут разрешения от своих объектов события - `ev1.WaitOne();` и `ev2.WaitOne();`) с помощью `ev1.Set();`устанавливается первое событие, которое и разрешает работу нити 1. По окончании цикла работы нити 1, событие 1 сбрасывается, а событие 2 устанавливается и т.д.

Исходныйкодпрограммы:

```
usingSystem;
using System.Threading;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
classProgram
 {
staticEventWaitHandle ev1 = newAutoResetEvent(false);
staticEventWaitHandle ev2 = newAutoResetEvent(false);
classPervaj_1
  {
publicvoid Niti_1()
   {
Console.WriteLine("Работаетнить 1 ");
for (int i = 0; i < 10; i++)
    {
     ev1.WaitOne();
for (int j = 0; j < 10; j++)
Console.Write(i + " ");
Console.WriteLine();
     ev1.Reset();
     ev2.Set();
```

```
    }
   }
publicvoid Niti_2()
   {
Console.WriteLine("Работаетнить 2 ");
for (int i = 10; i < 20; i++)
    {
     ev2.WaitOne();
for (int j = 0; j < 10; j++)
Console.Write(i + " ");
Console.WriteLine();
     ev2.Reset();
     ev1.Set();
    }
   }
  }
staticvoid Main(string[] args)
  {
Pervaj_1 Pe = newPervaj_1();
Thread t1 = newThread(Pe.Niti_1);
   t1.Start();
Thread t2 = newThread(Pe.Niti_2);
   t2.Start();
   ev1.Set();
Console.ReadLine();
}
 }
}
```

Работа программы:

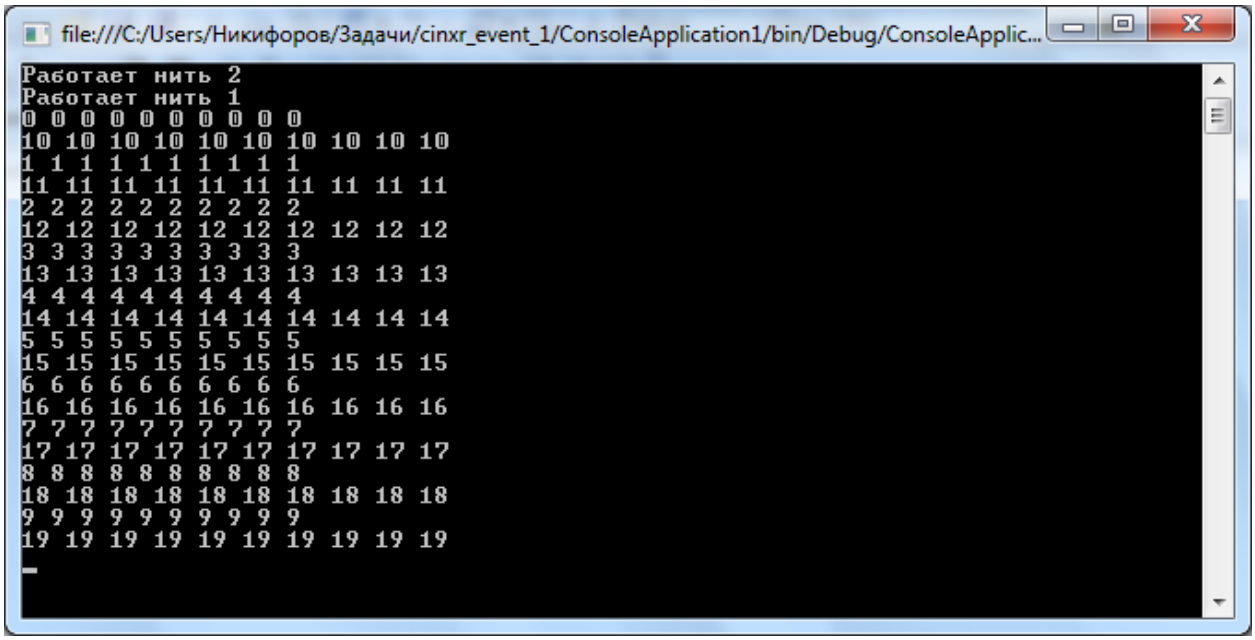


Рисунок 5.3.1 – Синхронизация нитей с помощью AutoResetEvent

Метод **Close** нужно вызывать сразу же, как только **WaitHandle** станет не нужен – для освобождения ресурсов операционной системы. Однако если **WaitHandle** используется на протяжении всей жизни приложения (как в нашем случае), этот шаг можно опустить, так как он будет выполнен автоматически при закрытии приложения.

Тема 6 ИСПОЛЬЗОВАНИЕ КАНАЛОВ ПЕРЕДАЧИ ДАННЫХ В ЯЗЫКЕ C#

6.1 Обмен данных между процессами

В лекции использован материал книги Побегайло А.П. Системное программирование в Windows и книги Албахари Д. и Албахари Б. C# 3.0 Справочник.

6.1.1Способы обмена данных между процессами

Под обменом данных между процессами понимается пересылка данных от одной нити к другой нити. При этом нити должны находиться в различных процессах (пересылку данных между нитями одного процесса мы рассматривали в модуле 2).

Нить, посылающая данные, называется отправителем, а нить, получающая данные, называется получателем или адресантом. Обмен данными между нитями реализуется с помощью специальных средств операционной системы – канала передачи данных.

Структурно канал передачи данных изображен на рисунке 6.1.1.

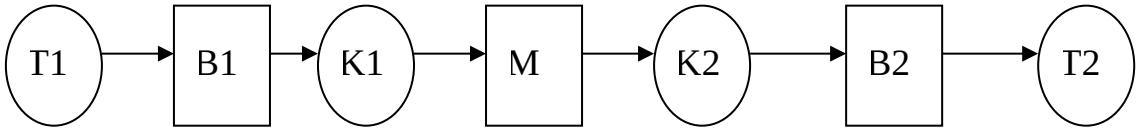


Рисунок 6.1.1 Структурная схема канала передачи данных

T1,T2 – нити различных процессов;
B1,B2 – буферы;
K1,K2 – потоки ядра операционной системы;
M – общая память.

При разработке платформы .NET очень большое внимание было уделено вопросам пересылке информации (одной из задач платформы является упрощения процессов программирования работы в сетях) и в технологии передачи данных появились новые понятия или изменилось назначение некоторых старых.

Определяющим является понятие потокав языке C#.

Поток в языке C# является стандартным классом платформы .NET и представлен набором методов для чтения, записи и позиционирования данных.

Потоки работают с данными последовательно, выполняя операции либо над байтами, либо над блоками небольшого размера. Поток «соединяет»источник и приемник данных. В качестве источника и приемника данных могут быть, например, файлы, процессы (нити) или сетевое соединение компьютера и т.д.

Различают потоки с резервным хранилищем и потоки декораторы.

Потоки с резервным хранилищем, например, FileStream, MemoryStream, NetworkStream это потоки, в которых данные только передаются (говорят, что передаются «сырые» данные) и поэтому для каждого типа данных необходимо использовать соответствующий поток (класс).

Потоки декораторы, например, DeflateStream илиGZipStream это потоки которые обеспечивают некоторые преобразования данных, например их сжатие или шифрование.

Существует понятие адаптеров потоков, которые являются специальными классами, обеспечивающими поток специализированными методами преобразования типов данных, например, StreamReader и StreamWriter обеспечивают потоку работу с текстами (обычно потоки работают с байтами данных).

6.1.2 Связи между процессами

Перед передачей данных между процессами необходимо установить связь между этими процессами.

Кто звонил по номеру телефона, к которому подключен модем, тот знает, что модем пытается установить связь с вами посредством повторяющихся сигналов.

Связь между процессами может устанавливаться как на физическом (аппаратном), так и на логическом (программном) уровнях.

В зависимости от направления передачи данных различают полудуплексную связь – данные передаются только в одном направлении, и дуплексную связь – данные передаются в обоих направлениях.

Обмен данными между параллельными процессами возможен двумя способами: потоком и сообщением.

Если данные передаются непрерывной последовательностью байтов, то такой способ передачи данных называется передача данных потоком. При этом возможна передача данных непосредственно из буфера одной нити в буфер другой нити без потоков ядра операционной системы и общей памяти.

Если данные передаются блоками байтов, то такой блок данных называется сообщением, а передача данных называется передачей данных сообщениями.

Очень часто связь между процессами характеризуется топологией связей – возможной структурой связей между процессами-отправителями и процессами-получателями данных.

С точки зрения топологии связей, например, для полудуплексного вида связей различают следующие виды связей:

1<-->1 – между собой связаны два процесса;

1<-->N – один процесс связан с N процессами;

N<-->1 – каждый из N процессов связан с одним процессом;

N<-->M – каждый из Nпроцессов связан с каждым изM процессов.

При разработке системы обмена данными между процессами первоначально определяется топология связей и направления передачи данных по этим связям. Это позволяет организовать связь на аппаратном или физическом уровне. После этого в программах реализуются выбранные связи между процессами с помощью специальных функций операционной системы, которые предназначены для установки связи между процессами. Эти функции совместно с функциями, предназначенными для обмена данными между процессами, находятся в системе передачи данных, которая является частью ядра операционной системы.

Объект ядра операционной системы, обеспечивающий передачу данных между процессами, выполняющимися на одном компьютере, называется «анонимным каналом».

Объект ядра операционной системы, обеспечивающий передачу данных между процессами, выполняющимися на компьютерах в локальной сети, называется «именованным каналом».

Объект ядра операционной системы, обеспечивающий передачу сообщений от процессов-клиентов к процессам-серверам, выполняющимися на компьютерах в локальной сети, называется «почтовым ящиком».

6.1.3 Передача сообщений

Обмен сообщениями между процессами выполняется при помощи двух функций, которые условно будем называть send (послать сообщение) и receive(получить сообщение).

Само сообщение состоит из двух частей – заголовка и тела сообщения.

В заголовок включается следующая служебная информация:

- тип сообщения;
- имя адресата сообщения;
- имя отправителя сообщения;
- контрольная информация, обычно длина сообщения и контрольная сумма.

Тело сообщения содержит данные пересылаемого сообщения.

Условно будем считать, что функции обмена имеют следующий вид:

send(ProcessA, сообщение); - послать сообщение процессу A и

receive(ProcessB, сообщение); - получить сообщение от процессора B.

При передачи может быть использована прямая или косвенная (по адресу) адресация.

При прямой адресации процессов в функциях send и receiveявно указываются имена процессов отправителя и получателя.

При косвенной адресации в функциях send и receiveуказываются не имена процессов отправителя и получателя, а имена связей, по которым необходимо передавать сообщения.

Адресация процессов может быть симметричной и асимметричной.

Если при обмене сообщениями между процессами используется только прямая или только косвенная адресация, то такая адресация процессов называется симметричной.

Если при обмене сообщениями между процессами используется как прямая, так и косвенная адресация, то такая адресация процессов называется асимметричной.

Асимметричная адресация часто используется в системах «клиент-сервер». Как правило «клиент» знает адрес «сервера» и посылает ему сообщения с использованием прямой адресации процесса. «Сервер» настраивается на канал связи, а не на конкретного «клиента» (говорят, что «сервер слушает» канал связи) – для этого используется косвенная адресация процессов.

Набор правил, по которым устанавливаются связи и передаются данные между процессами, называется протоколом.

По этим правилам, например, нельзя начинать передачу данных, не установив связь между отправителем и адресантом и т.д.

Передача данных возможна синхронным или асинхронным способами.

Если процесс, отправивший сообщение блокируется до получения этого сообщения адресантом (точнее до получения ответа от адресанта, что сообщение получено), то такая передача данных называется синхронной. Обычно отправитель сообщения должен получить информацию о готовности адресанта принять следующее сообщение.

Асинхронный способ передачи сообщений не требует подтверждения о приеме каждого сообщения. Такой способ передачи применяется, когда быстродействие получателя сообщений во много раз больше отправителя сообщений и очень низкая вероятность появления помех. В таких системах после установки связи сообщения пересылаются непрерывным потоком.

При синхронном способе передачи есть возможность проверить каждое сообщение и в случае обнаружения ошибки потребовать повторной передачи сообщения.

В системах с обнаружением ошибок используют избыточные коды. Например, для передачи одного бита информации используется два двоичных разряда: 0 заменяется кодом 00, а 1 – кодом 11. Коды 01 и 10 являются запрещенными. И если получатель сообщения в процессе приема данных получает код 01 или 10, то он определяет, что на сообщение воздействовала помеха и сообщение необходимо повторить.

Существуют системы с автоматическим исправлением ошибок при приеме сообщений. В таких системах для передачи одного бита информации используется трех или более разрядные двоичные коды. Например, для передачи 0 используется 000, а для передачи 1 – 111. Все остальные трех разрядные коды являются запрещенными. В случае приема кодов 011, 101 или 110 система с исправлением ошибки может предположить, что передавался код 111, но на него воздействовала одиночная помеха и принять полученных код за значение 111. Аналогично запрещенные коды 001, 010 и 100 могут быть приняты за код 000. В таких сложных системах просчитывается частота появления одиночной ошибки, контрольные суммы сообщений и другие известные средства обнаружения и исправления ошибок – автоматический переход на дублирование сообщений и т.д.

Контрольная сумма может получаться различными алгоритмами. Наиболее простой алгоритм основан на суммировании всех передаваемых байтов сообщений с отбрасыванием переполнения. Приемник сообщения также суммирует все байты принимаемого сообщения, и полученное значение сравнивается с полученным значением контрольной суммы. Если они различаются, то приемник требует повторить передачу сообщения.

Синхронный обмен данными с использованием прямой адресации процессов называется рандеву (встреча).

Согласование работы различных процессов, имеющих разные скоростные характеристики, осуществляется с помощью буфера.

Буфер характеризуется «вместимостью связи между процессами» - количество сообщений, которое может одновременно пересылаться по этой связи.

Различают три типа буферизации:

- нулевой вместимости связи (нет буфера). В этом случае возможен только синхронный обмен данными между процессами.

- ограниченная вместимость связи (ограниченный буфер). В этом случае если буфер полон, то отправитель сообщения ждет очистки буфера хотя бы от одного сообщения.

- неограниченная вместимость связи (неограниченный буфер). В этом случае отправитель никогда не ждет при отправки сообщений.

Возможности буфера тесно связаны с синхронизацией передачи данных и должны учитываться при разработке систем обмена данными между процессами.

6.1.4 Обмен данными между процессами с помощью файла

Простейший способ передачи данных между процессами (нитями процессов) можно организовать, если общую память (см. рисунок 10.1) заменить файлом. Работа нитей с потоками (в нашем случае это потоки с резервным хранилищем) обычно не является безопасной. Действительно, если нитям разрешить писать и читать данные в потоке одновременно, то нет

гарантии, что не возникнет ошибки. Поэтому работа с потоком нескольких процессов требует выполнение определенных правил. Например, при работе с потоком с резервным хранилищем, представленным файлом, необходимо, выполнив некоторое действие с файлом, его закрывать. Желательно не допускать одновременной работы с файлом нескольких процессов. Рекомендуется выполнять работу с файлом поочередно.

В качестве учебного примера рассмотрим работу двух процессов с одним текстовым файлом, через который первый процесс передает во второй процесс массив байтов, сформированный случайным образом. Второй процесс забирает содержимое файла в массив байтов, сортирует его и возвращает в файл. Первый процесс забирает из файла отсортированный массив и печатает его. Задача реализована двумя программами – основной программой и дополнительной программой.

Код основной программы:

```
using System;
using System.IO;
using System.Diagnostics;
namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main()
  {
byte[] a = newbyte[10];
int k = 0;
string buf;
while (k < 6)
   {
Console.WriteLine("1 - Создатьфайл ");
Console.WriteLine("2 - Создать и напечатать массив из 10
чисел");
Console.WriteLine("3 - Сортировка файла ");
Console.WriteLine("4 - Запись массива в файл");
Console.WriteLine("5 - Прочитать массив из файла и напечатать
его");
Console.WriteLine("6 - Выход из программы");
Console.WriteLine("Введите пункт меню программы");
buf = Console.ReadLine();
    k = Convert.ToInt32(buf);
switch (k)
    {
case 1:
using (Stream fmi = newFileStream("FiMaIn.txt",
FileMode.Create))
     { fmi.Write(a, 0, a.Length); fmi.Close(); }; break;
case 2: SozdMas(a); break;
case 3: Process.Start("ConsoleApplication2"); break;
case 4:
using (Stream fmi = newFileStream("FiMaIn.txt", FileMode.Open))
     { fmi.Write(a, 0, a.Length); fmi.Close(); }; break;
```

```
case 5:
using (Stream fmi = newFileStream("FiMaIn.txt", FileMode.Open))
     { fmi.Read(a, 0, a.Length); fmi.Close(); }; ReadMas(a);
break;
default: break;
    }}}
publicstaticvoid SozdMas(byte[] ma)
  {
Random rnd = newRandom();
Console.WriteLine("Массивсоздан !!");
for (int i = 0; i < 10; i++)
   {
    ma[i] = (byte)(rnd.Next() % 101);
Console.Write(ma[i]+"\t");
   }
Console.WriteLine();
  }
publicstaticvoid ReadMas(byte[] ma)
  {
for (int i = 0; i < 10; i++)
Console.Write(ma[i] + "\t");
Console.WriteLine();
  } }}
```

Дополнительная программа сортировки массива:

```
using System;
using System.IO;
using System.Diagnostics;

namespace ConsoleApplication2
{
classProgram
 {
staticvoid Main()
  {
byte b;
byte[] a = newbyte[10];
Console.WriteLine("Cortirovka nacinaetcj");
Console.ReadLine();
using (Stream fmi = newFileStream("FiMaIn.txt", FileMode.Open))
   {
    fmi.Read(a, 0, a.Length);
for (int i = 0; i < 10; i++)
Console.Write(a[i] + "\t");
Console.WriteLine();
for (int i = 0; i < 9; i++)
for (int j = i+1; j < 10; j++)
if(a[i]<a[j])
      {
       b = a[i]; a[i] = a[j]; a[j] = b;
      }
for (int i = 0; i < 10; i++)
Console.Write(a[i] + "\t");
```

```
Console.WriteLine();
     fmi.Close();
    }
using (Stream fmi = newFileStream("FiMaIn.txt",
FileMode.Create))
    {
     fmi.Write(a, 0, a.Length);
     fmi.Close();
    }
Console.WriteLine("Cortirovka zakoncena");
Console.ReadLine();
  } }}
```

Работа программ:

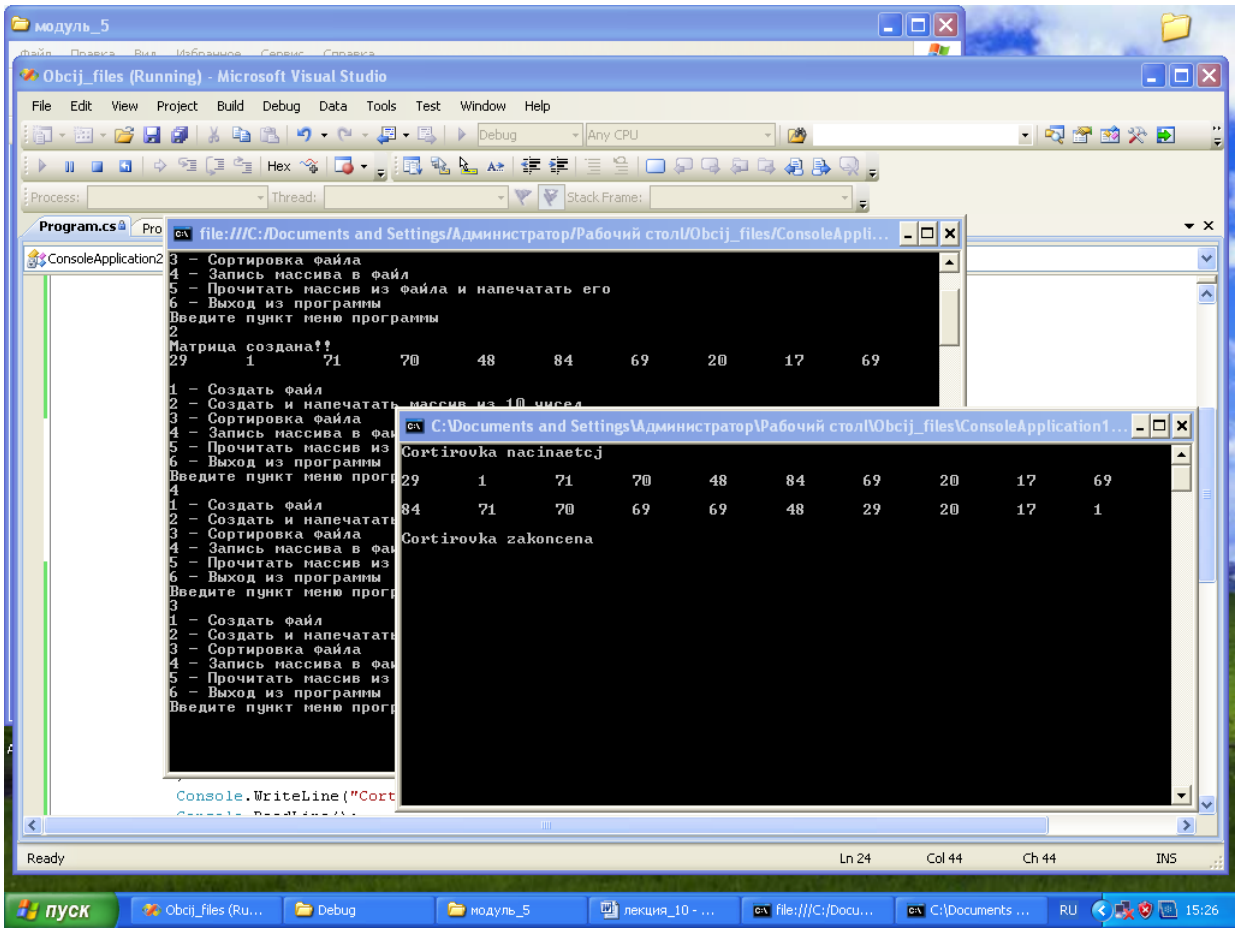


Рисунок 6.1.2 Обмен данными между процессами с помощью файла

В примерах, перед записью данных в файл они как бы создаются заново. Это нужно, чтобы выполнить очистку файла. Естественно, эту операцию можно выполнить и с помощью метода очистки.

6.2 Работа с каналами в языке C#

В лекции использован материал книги Побегайло А.П. Системное программирование в Windows и книги Албахари Д. и Албахари Б. C# 3.0 Справочник.

6.2.1 Каналы в языке C#

Как мы отмечали в предыдущей лекции, обмен данными между процессами в системе Windows осуществляется через каналы. Естественно в языке C# имеется специальный класс, реализующий технологию передачи данных через канал – это класс PipeStream (дословно поток-канал).

Класс PipeStream предоставляет набор методов, реализующих взаимодействие процессов с помощью протоколов (правил обмена данными) каналов в системе Windows.

Существу два вида каналов в языке C# это анонимный и именованный каналы.

Именованный канал предполагает создание объекта канал с заданием ему имени для работы в сети Windows. Основное его назначение передача данных между процессами (обычно их называют клиентом и сервером), работающими на разных компьютерах.

Анонимный канал предназначен для передачи данных между процессами, работающими на одном компьютере, что позволяет обходиться без транспортного протокола.

Класс PipeStream состоит из четырех подклассов. Два первых подкласса обеспечивают работу анонимного канала (AnonymousPipeServerStream и AnonymousPipeClientStream), а два вторых обеспечивают работу именованного канала (NamePipeServerStream и NamePipeClientStream).

Знакомство с работой каналов начнем с именованного канала передачи данных.

6.2.2 Именованные каналы

Именованным каналом называется объект языка C#, который обеспечивает передачу данных между процессами в локальной сети.

Обычно именованные каналы используются для передачи данных между процессами, выполняющимися на компьютерах в одной локальной сети, но они могут использоваться для передачи данных между процессами на одном компьютере.

К сожалению, администрация корпоративной сети нашего университета запрещает все «несанкционированные» передачи между компьютерами в сети, поэтому передачи будут организованы между процессами на одном компьютере.

Процесс, который создает именованный канал, называется сервером именованного канала. Процессы, которые связываются с именованным каналом, называются клиентами именованного канала.

Перечислим основные характеристики именованных каналов:
- канал имеет имя, которое используется клиентами для связи с именованным каналом;
- передача данных в канале может осуществляться как потоком, так и сообщениями;
- обмен данными может быть как синхронным, так и асинхронным.

Теперь приведем порядок работы с именованными каналами, который и будет использоваться в дальнейшем:
-создание именованного канала сервером;
- ожидание сервера подключения клиента к экземпляру именованного канала;
- создание именованного канала клиентом;
- соединение клиента с экземпляром именованного канала;
- обмен данными по именованному каналу;
- закрытие именованного канала клиентом и сервером.
Рассмотрим подробнее перечисленные пункты работы с именованными каналами.

Именованные каналы создаются процессом-сервером при помощи конструктора NamedPipeServerStream, который имеет 10 различных вариантов записи:

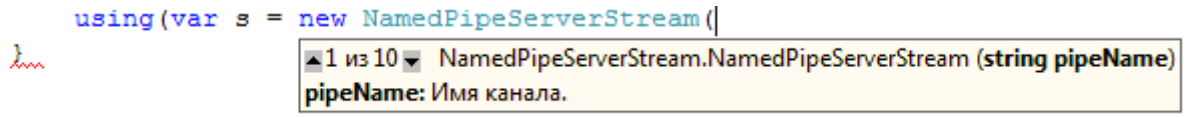


Рисунок 6.2.1 Варианты создания объекта именованного канала

Все варианты создания объекта именованного канала можно посмотреть в справке VisualStudio. Мы рассмотрим один, имеющий максимальное количество параметров:
NamedPipeServerStream
(String,
PipeDirection,
Int32,
PipeTransmissionMode,
PipeOptions,
Int32,
Int32,
PipeSecurity,
HandleInheritability,
PipeAccessRights)

Первый параметр предложенного варианта конструктора содержит имя канала. Это имя канала будет использоваться и сервером и клиентом в локальной сети, поэтому оно должно быть уникальным.

Второй параметр PipeDirection, определяет направление передачи данных. Возможны варианты In, InOut иOut. Именованные каналы, по умолчанию, являются двунаправленными. Серверу и клиенту разрешено передавать или читать данные в канал. Для этого сервер и клиент «должны договориться» о каком-нибудь протоколе, координирующем их действия (например, обмен сообщениями поочередно), чтобы исключить одновременного обращения к каналу.

Третий параметр определяет максимальное число экземпляров сервера с одинаковыми именами, которым могут подключаться клиенты. В нашем случае будет существовать одна пара сервер – клиент.

Параметр PipeTransmissionMode определяет режим передачи данных, например, PipeTransmissionMode.Message определяет передачу сообщениями.

Параметр PipeOptions определяет способ создания или открытия канала.

Следующие параметры задают размеры входного и выходного буферов обмена данными канала. Обычно они определяются по умолчанию.

Режим безопасности работы канала определяется параметром PipeSecurity. У вас будет дисциплина «Защита информации», в которой будут рассмотрены значения подобных параметров объектов.

Параметр HandleInheritability определяет, может ли базовый дескриптор канала наследоваться «дочерними» процессами.

Параметр PipeAccessRights определяет права доступа к каналу.

Таким образом, конструктор рассматриваемого варианта создает и инициализирует новый экземпляр класса NamedPipeServerStreamc заданным именем канала, направлением канала, максимальным количеством экземпляров сервера, режимом передачи, параметрами канала, рекомендуемыми размерами входного и выходного буферов, режимом безопасности канала, режимом наследования и правами доступа к каналу.

После создания именованного канала сервер ждет подключения клиента с помощью метода WaitForConnection();.

Процесс клиента в языке C# также создает объект именованного канала с помощью конструктора, имеющего 8 вариантов записей.

Все варианты создания объекта именованного канала процессом клиентом можно посмотреть в справке VisualStudio. Мы рассмотрим один, имеющий максимальное количество параметров:

```
NamedPipeClientStream
(String,
String,
PipeDirection,
PipeOptions,
TokenImpersonationLevel,
HandleInheritability)
```

Как сказано в Help (F1) среды данная запись инициализирует новый экземпляр класса NamePipeClientStream с заданными именами канала и

сервера, направлением передачи данных в канале, параметрами канала, уровнем безопасности и режимом наследования.

Следующими этапами по протоколу работы именованного канала клиент должен соединиться с экземпляром именованного канала. Далее выполняется обмен данными между сервером и клиентом. По окончании обмена осуществляется закрытие именованного канала клиентом и сервером.

В качестве учебного примера рассмотрим работу именованного канала, через который сервер передает клиенту массив байтов, сформированный случайным образом. Клиент сортирует его и возвращает серверу. Сервер распечатывает полученный массив. Задача реализована двумя программами – основной программой (сервером) и дополнительной программой (клиентом).
Исходный код основной программы:

```
usingSystem;
using System.Collections.Generic;
using System.Linq;
using System.IO.Pipes;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main(string[] args)
  {
byte[] a = newbyte[10];
using (var s = newNamedPipeServerStream("Pipe_lab6"))
   {
Random rnd = newRandom();
Console.WriteLine("Массивсоздан: ");
for (int i = 0; i < 10; i++)
    {
     a[i] = (byte)(rnd.Next() % 101);
Console.Write(a[i] + "\t");
    }
Console.WriteLine();

    s.WaitForConnection();
for (int i = 0; i < 10; i++)
         s.WriteByte(a[i]);
for (int i = 0; i < 10; i++)
a[i] = (byte)(s.ReadByte());
Console.WriteLine("Массив после сортировки: ");
for (int i = 0; i < 10; i++)
Console.Write(a[i] + "\t");
Console.WriteLine();
   }
Console.WriteLine("Сеанссерверазакончен");
Console.ReadLine();
  }
 }
}
```

Исходный код дополнительной программы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO.Pipes;

namespace ConsoleApplication2
{
classProgram
 {
staticvoid Main(string[] args)
  {
byte[] a = newbyte[10];
using (var s = newNamedPipeClientStream("Pipe_lab6"))
{
Console.WriteLine("Начинает работать клиент");
byte b;
    s.Connect();
for (int i = 0; i < 10; i++)
a[i] = (byte)(s.ReadByte());
Console.WriteLine("Полученный массив байтов:");
for (int i = 0; i < 10; i++)
Console.Write(a[i] + "\t");
Console.WriteLine();
for (int i = 0; i < 9; i++)
for (int j = i + 1; j < 10; j++)
if (a[i] < a[j])
         { b = a[i]; a[i] = a[j]; a[j] = b; }
Console.WriteLine("Массивпослесортировки:");
for (int i = 0; i < 10; i++)
Console.Write(a[i] + "\t");
Console.WriteLine();

for (int i = 0; i < 10; i++)
        s.WriteByte(a[i]);
   }
Console.WriteLine("Сеансклиентазакончен");
Console.ReadLine();
  }}}
```

Работа программы представлена на рисунке 6.2.2.

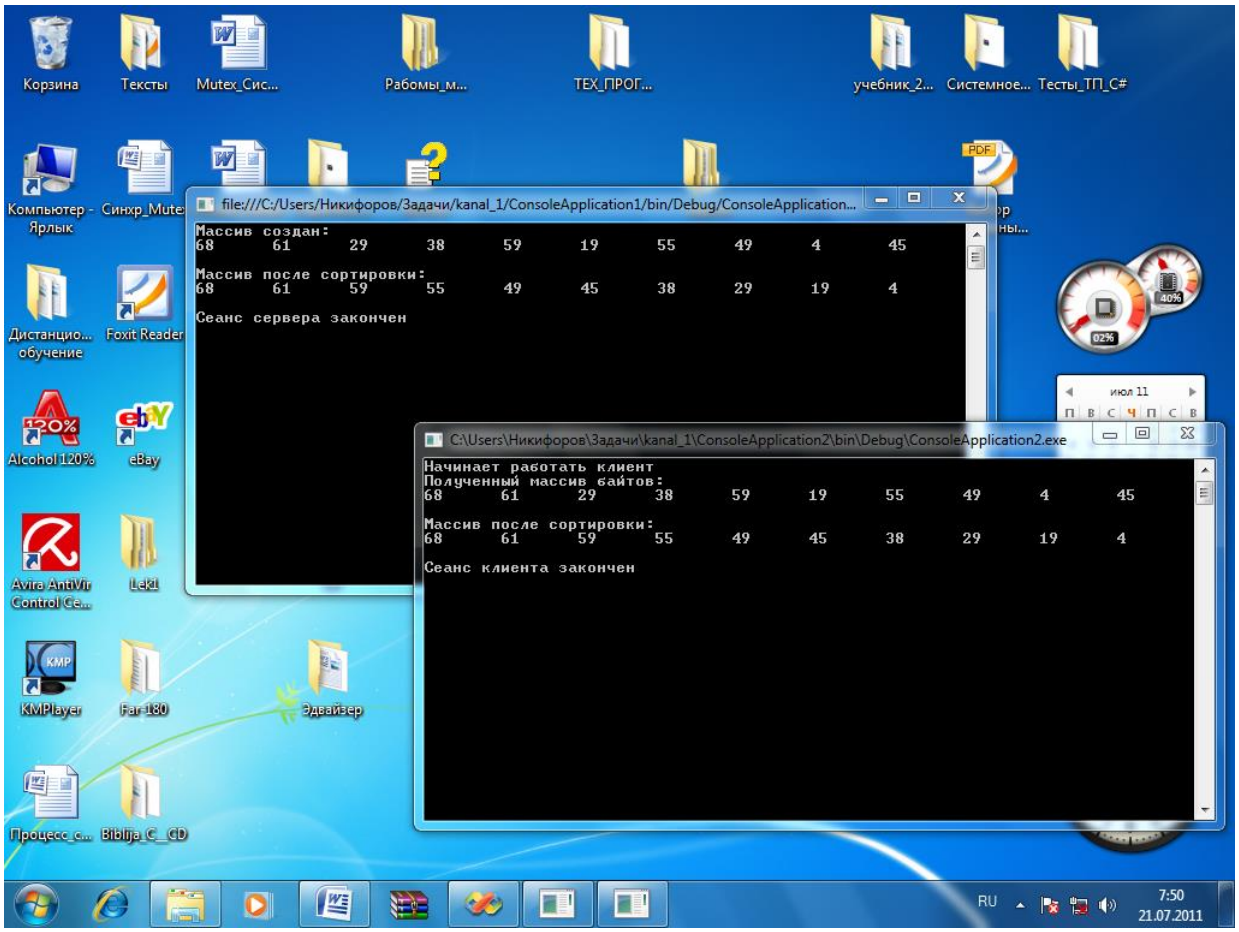


Рисунок 6.2.2 – Работа именованного канала при передаче массива байтов

6.2.3 Использование именованного канала для передачи сообщений

Рассмотрим пример программ, в которых сервер и клиент поочередно обмениваются текстовыми сообщениями. Процесс обмена прекращается, когда один из участников отправит сообщение “ДОМОЙ”.

Исходный код сервера:

```
usingSystem;
using System.Threading;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;

namespace Server_Name_1
{
classProgram
 {
staticvoid Main(string[] args)
  {
string clientfileExe = "Klient_Name_1";
Process p = Process.Start(clientfileExe);
// Созданиеименованногоканаласервером.
```

```
NamedPipeServerStream pipestream =
newNamedPipeServerStream("Kanal_Name_1");
// Ждем соединения с клиентом.
Console.WriteLine("Ждем соединения с клиентом");
pipestream.WaitForConnection();

StreamReader reader = newStreamReader(pipestream);
StreamWriter writer = newStreamWriter(pipestream);
string ss = "";
int fl = 0;
while (ss != "ДОМОЙ")
   {
if (fl == 0)
    {
Console.WriteLine("Вашесообщение ?");
     ss = Console.ReadLine();
     writer.WriteLine(ss);
     writer.Flush();
     fl = 1;
    }
else
    {
     ss = reader.ReadLine();
Console.WriteLine("Получено сообщение от клиента : " + ss);
fl = 0;
    }
   }
Console.WriteLine("Сеанс связи закончен");
Console.ReadLine();
// Закрываемканал.
   pipestream.Close();
}}}
```

Исходный код клиента:

```
usingSystem;
using System.Threading;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;

namespace Klient_Name_1
{
classProgram
 {
staticvoid Main(string[] args)
{
stringss;
//Создаем именованный канал клиентом.
NamedPipeClientStream pipestream =
newNamedPipeClientStream("Kanal_Name_1");
//Соединяемся с каналом.
pipestream.Connect();
```

```
StreamReader reader = newStreamReader(pipestream);
StreamWriter writer = newStreamWriter(pipestream);
int fl = 0;
do
   {
if (fl == 0)
    {
     ss = reader.ReadLine();
Console.WriteLine("Полученосообщениеотсервера: " + ss);
fl = 1;
    }
else
    {
Console.WriteLine("Сообщение от клиента ?");
ss = Console.ReadLine();
     writer.WriteLine(ss);
     writer.Flush();
     fl = 0;
    }
   }
while (ss != "ДОМОЙ");
Console.WriteLine("Сеанссвязизакончен");
Console.ReadLine();
// Закрываемканал.
   pipestream.Close();
}}}
```

Работа программы представлена копией экрана консольных приложений сервера и клиента, представленной на рисунке 6.2.3.

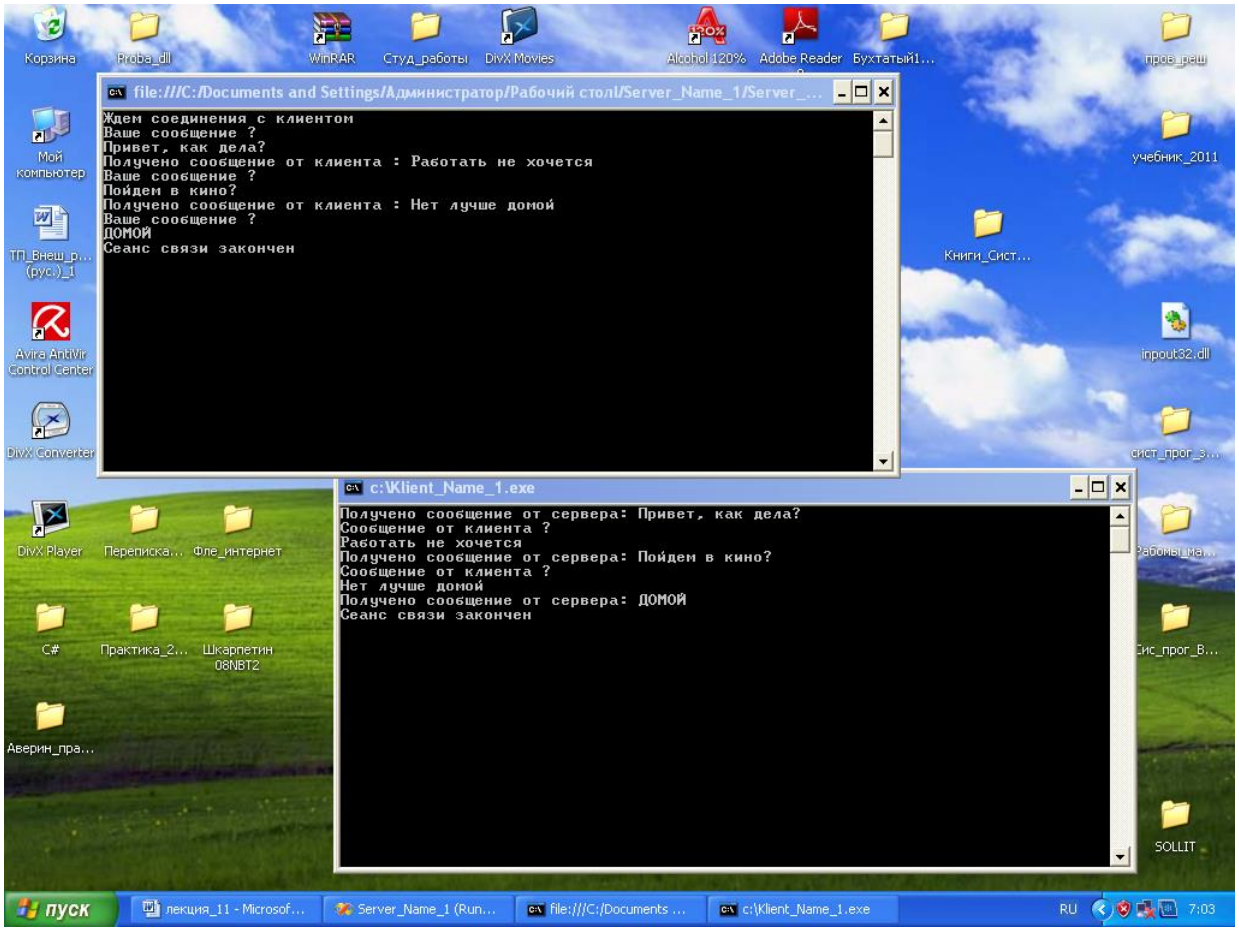


Рисунок 6.2.3 – Работа именованного канала при передаче сообщений

Для нормальной работы программ файл `Klient_Name_1.exe`, должен находиться в папке с файлом `Server_Name_1.exe`.

Работа программ:

Созданный канал открыт для чтения и записи, поэтому для исключения ситуации одновременного обращения к каналу и сервером и клиентом использован протокол поочередного доступа к каналу, который реализуется с помощью специальной переменной «флага»fli операторов условного перехода.

Для передачи текстовых сообщений по именованному каналу использованы потоковые адаптеры:

```
StreamReader reader = newStreamReader(pipestream);
StreamWriter writer = newStreamWriter(pipestream);
```

которые будут рассмотрены в следующей лекции.

6.3 Потоковые адаптеры и анонимные каналы

6.3.1 Понятие потокового адаптера

Базовым классом всех потоков является класс Stream, который оперирует только с байтами данных.

Однако, существует множество задач, в которых предпочтительнее является работа не с байтами данных, а с более сложными типами, например, строками или XML-документами.

Поэтому в языке C# были разработаны специальные классы, включающие специализированные методы для работы со сложными типами данных. Эти классы не являются потоками, а предоставляют потокам специализированные методы, выполняющие роль преобразователей типов данных в потоках, как бы расширяют возможности потоков.

Потоковые адаптеры это специальные классы, предназначенные для работы с потоками и содержащие специальные методы для работы со сложными типами данных.

В настоящее время платформа .NET, точнее ее библиотека Framework, содержит три группы потоковых адаптеров:

- текстовые адаптеры для строковых и символьных типов данных;
- двоичные адаптеры для работы со стандартными базовыми типами, такими как bool, byte, char, int, float, doubleи т.д.
- XML-адаптеры, позволяют значительно упростить работу потоков с XML-документами.

В этой лекции мы познакомимся с текстовыми и двоичными потоковыми адаптерами.

6.3.2 Текстовые и двоичные потоковые адаптеры

Текстовые потоковые адаптеры платформы .NET базируются на абстрактных классах TextReader иTextWriter.

Класс TextReader, в свою очередь, является абстрактным базовым классом для классов StreamReader и StringReader, которые обеспечивают чтение символов либо из потока, либо из обычной строки.

Класс TextWriter, так же является абстрактным базовым классом для классов StreamWriter и StringWriter, которые обеспечивают запись символов либо в поток, либо в обычную строку.

Таким образом, эти классы обеспечивают передачу данных в символьном виде как из потока в строку, так из строки в поток.

Каждый из перечисленных классов имеет богатый набор методов, свойств и полей, в дополнение к которым обычно около 10 перегружаемых конструкторов. Копия экрана справки для StreamReader приведена в конце лекции.

В предыдущей лекции для чтения текстовых сообщений из именованного канала использован потоковый адаптер `StreamReaderreader = newStreamReader(pipestream);`, который позволил читать данные из потока в строковом виде `ss = reader.ReadLine();`и сразу отображать их в консольном окне экрана монитора `Console.WriteLine("Полученосообщениеотсервера: " + ss);`.

Для записи текстовых сообщений в именованный канал передачи данных, в предыдущей лекции использован потоковый адаптер `StreamWriterwriter = newStreamWriter(pipestream);`, который позволил записывать данные из строковой переменной в поток. Например,

```
Console.WriteLine("Сообщение от клиента ?");
ss = Console.ReadLine();
writer.WriteLine(ss);
writer.Flush();
```

Методwriter.Flush(); очищает содержимое буфера обмена.

Двоичные потоковые адаптеры представлены классами BinaryReaderи BinaryWriter, которые читают и, соответственно, записывают данные базовых типов в поток. Из первого семестра вы знаете, что к базовым типам относятся все значимые типы языка C#, а это bool, byte, int, float, double и т.д.

В отличие от текстовых адаптеров двоичные адаптеры хранят данные базовых типов в том виде, в каком они представлены в памяти компьютера – под данные целого типа отводится 4 байта, а под данные типа double – 8 байт.

6.3.3 Понятие анонимного канала передачи данных

Как мы уже отмечали, анонимный канал предназначен для передачи данных между процессами, работающими на одном компьютере.

Классы, обеспечивающие работу анонимного канала, имеют названия AnonymousPipeServerStream и AnonymousPipeClientStream.

Анонимный канал может иметь только однонаправленный поток – только чтение или только запись. Поэтому для обмена данными между процессами необходимо создавать два анонимных канала.

Скорость работы анонимного канала значительно быстрее, чем у именованного канала, так как он не выходит за границы операционной системы компьютера (говорят, что он не работает с транспортными протоколами сети).

При создании сервером анонимного канала операционная система присваивает каналу «закрытый дескриптор», идентификатор которого необходимо передавать клиенту. Обычно это осуществляется с помощью аргументов метода Main программы клиента. Получив идентификатор дескриптора анонимного канала сервера, клиент может создать свой анонимный канал для передачи данных, но с противоположным направлением – если сервер настраивается на чтение данных, то клиент должен записывать данные в канал и наоборот.

В общем случае взаимодействия сервера и клиента должны осуществляться в следующей очередности:

- сервер создает анонимный канал передачи данных, определяя направление передачи (In илиOut);
- с помощью метода GetClientHandleAsString() сервер получает идентификатор созданного анонимного канала;
- сервер запускает процесс клиента и передает ему в виде аргумента метода Main полученный идентификатор;
- клиент создает экземпляр анонимного канала для полученного идентификатора, но с противоположным направлением передачи данных;
- сервер освобождает «закрытый дескриптор» полученный при создании анонимного канала;
- сервер и клиент обмениваются данными.

Рассмотрим работу анонимного канала передачи данных.

6.3.4 Анонимный канал передачи вещественных данных

Рассмотрим чисто учебный пример передачи вещественного чисел от сервера клиенту и целого числа от клиента серверу.

Исходный код сервера:

```
usingSystem;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main(string[] args)
  {
string clientExe = "ConsoleApplication2.exe";
AnonymousPipeServerStream PipeServer1 =
newAnonymousPipeServerStream(PipeDirection.Out,
HandleInheritability.Inheritable);
AnonymousPipeServerStream PipeServer2 =
newAnonymousPipeServerStream(PipeDirection.In,
HandleInheritability.Inheritable);
   string txID = PipeServer1.GetClientHandleAsString();
string rxID = PipeServer2.GetClientHandleAsString();
var startInfo = newProcessStartInfo(clientExe, txID + " " +
rxID);
   startInfo.UseShellExecute = false;//водноконсольноеокно
Process p = Process.Start(startInfo);
   PipeServer1.DisposeLocalCopyOfClientHandle();
   PipeServer2.DisposeLocalCopyOfClientHandle();
BinaryWriter dd = newBinaryWriter(PipeServer1);
   dd.Write(15.5);
BinaryReader ii = newBinaryReader(PipeServer2);
Console.WriteLine("сервер получил сообщение:  " +
ii.ReadInt32());
```

```
   p.WaitForExit();
Console.WriteLine("сервер - сеанс закончен");
Console.ReadLine();
  }
 }
}
```

Исходный код клиента:

```
using System;
using System.IO;
using System.IO.Pipes;

namespace ConsoleApplication2
{
classProgram
 {
staticvoid Main(string[] args)
  {
string rxID = args[0];
string txID = args[1];
using (var rx = newAnonymousPipeClientStream(PipeDirection.In,
rxID))
using (var tx = newAnonymousPipeClientStream(PipeDirection.Out,
txID))
   {
BinaryReader dd = newBinaryReader(rx);
Console.WriteLine(" клиентполучилсообщение: " +
dd.ReadDouble());
BinaryWriter ii = newBinaryWriter(tx);
    ii.Write(1234);
   }
Console.WriteLine("клиент - сеансзакончен");
Console.ReadLine();
  }}}
```

Для нормальной работы программ файл ConsoleApplication2.exe, должен находиться в папке с файлом ConsoleApplication1.exe. Рассматривая работу программы необходимо отметить, что были созданы два анонимных канала сервером PipeServer1и PipeServer2, для которых были получены идентификаторы каналов txIDи rxID. Эти идентификаторы были переданы программе клиент (при ее запуске).

В программе сервер были использованы два двоичных потоковых адаптера – для передачи вещественного числа и для приема целого числа.

Программа клиент с помощью аргументов метода Main приняла идентификаторы созданных каналов и сформировала два анонимных канала rx итх для чтения и записи данных в канал. Одновременно программа клиент использовала два двоичных потоковых адаптера – для приема вещественного числа и для отправки целого числа серверу.

Работа программ отображена на рисунке 6.3.1.

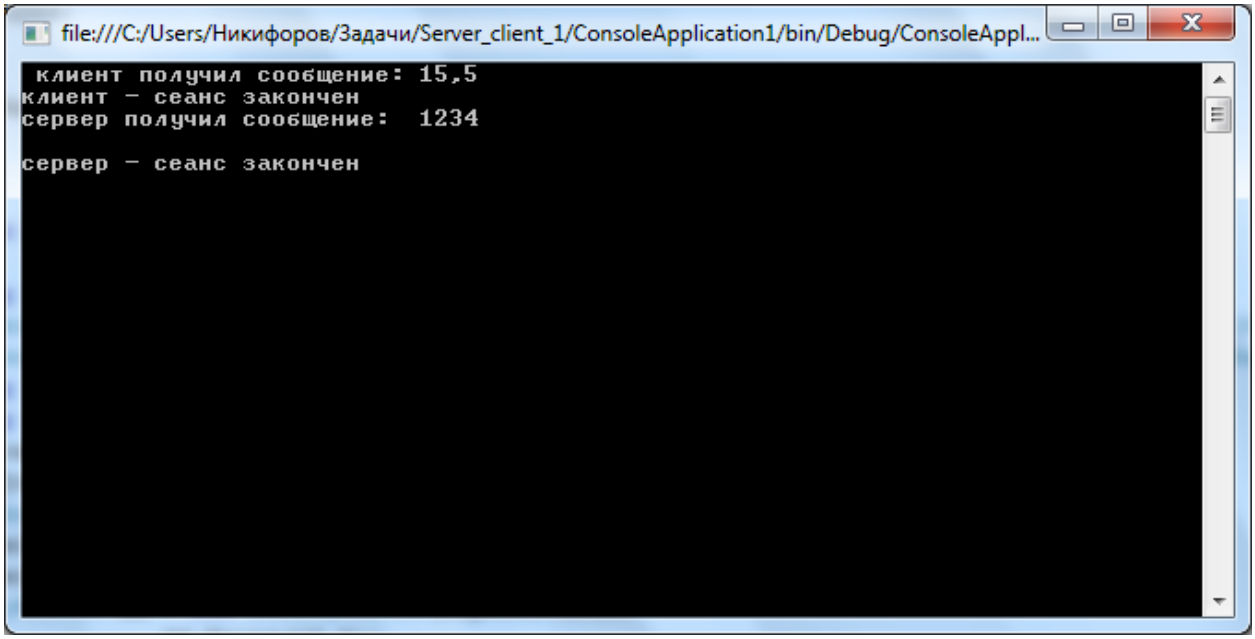


Рисунок 6.3.1 – Работа анонимного канала

6.3.5 Анонимный канал передачи текстовых данных

В чисто учебных целях рассмотрим работу анонимного канала передачи текстовых данных от сервера к клиенту и от клиента к серверу. Естественно и сервер и клиент должны создавать по два анонимных канала передачи данных и дважды использовать текстовые потоковые адаптеры. Перед запуском программ, но после их компиляции файл `ConsoleApplication2.exe` необходимо разместить в корневом каталоге диска с.

Исходный код программы сервера:

```
usingSystem;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace ConsoleApplication1
{
classProgram
 {
staticvoid Main(string[] args)
  {
string clientExe = "ConsoleApplication2.exe";
AnonymousPipeServerStream PipeServer1 =
newAnonymousPipeServerStream(PipeDirection.Out,
HandleInheritability.Inheritable);
```

```
AnonymousPipeServerStream PipeServer2 =
newAnonymousPipeServerStream(PipeDirection.In,
HandleInheritability.Inheritable);
string txID = PipeServer1.GetClientHandleAsString();
string rxID = PipeServer2.GetClientHandleAsString();
var startInfo = newProcessStartInfo(clientExe, txID + " " +
rxID);
   startInfo.UseShellExecute = false;
Process p = Process.Start(startInfo);
StreamReader reader = newStreamReader(PipeServer2);
StreamWriter writer = newStreamWriter(PipeServer1);
   PipeServer1.DisposeLocalCopyOfClientHandle();
   PipeServer2.DisposeLocalCopyOfClientHandle();
Console.WriteLine("Вашесообщение ?");
string ss = Console.ReadLine();
   writer.WriteLine(ss);
   writer.Flush();
Console.WriteLine("серверполучилсообщение:  " +
reader.ReadLine());
   p.WaitForExit();
Console.WriteLine("сервер - сеансзакончен");
Console.ReadLine();
  }
 }
}
```

Исходный код программы клиента:

```
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication2
{
classProgram
 {
staticvoid Main(string[] args)
  {
string rxID = args[0];
string txID = args[1];
using (var rx = newAnonymousPipeClientStream(PipeDirection.In,
rxID))
using (var tx = newAnonymousPipeClientStream(PipeDirection.Out,
txID))
   {
StreamReader reader = newStreamReader(rx);
StreamWriter writer = newStreamWriter(tx);
Console.WriteLine(" клиентполучилсообщение: " +
reader.ReadLine());
Console.WriteLine("Вашесообщение ?");
```

```
string ss = Console.ReadLine();
    writer.WriteLine(ss);
    writer.Flush();
   }
Console.WriteLine("клиент - сеансзакончен");
Console.ReadLine();
  }
 }
}
```

Для нормальной работы программ файл `ConsoleApplication2.exe`, должен находиться в папке с файлом `ConsoleApplication1.exe`. Работа программы представлена копией экрана на рисунке 6.3.2

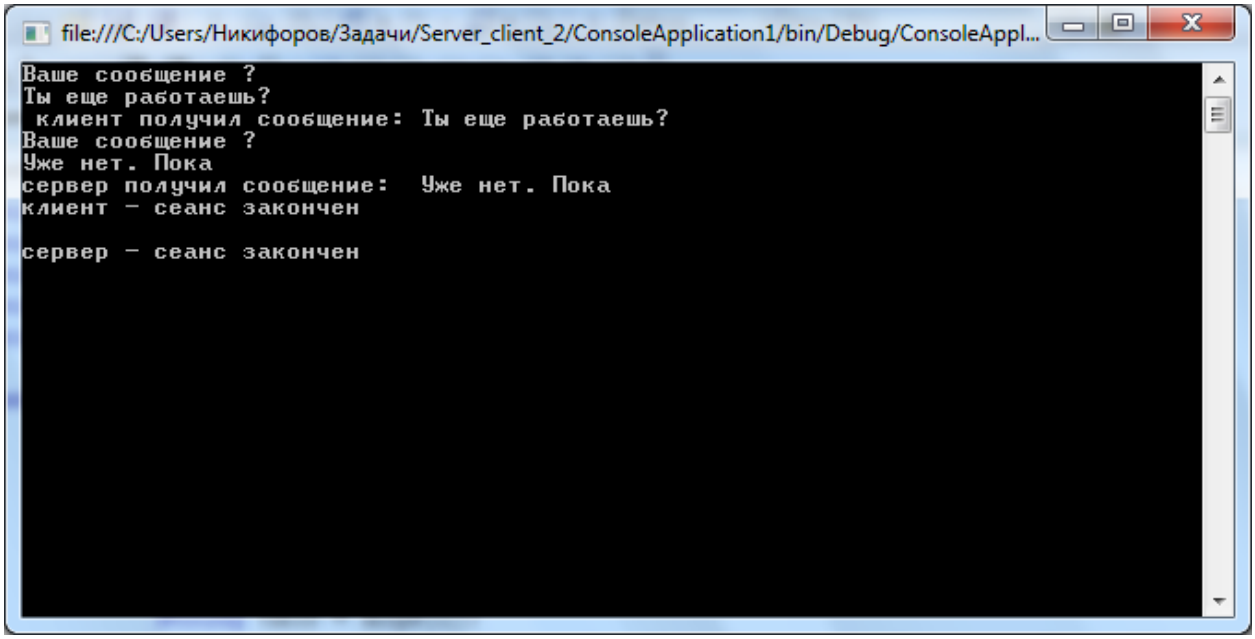


Рисунок 6.3.2 – Работа анонимного канала при передаче текстовых данных

7 СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

7.1 Основная литература

1 В.В. Фаронов «Создание приложений с помощью C#» Руководство программиста. - М.: “Эксмо”, 2008г.

2 Т.А. ПавловскаяC#, Программирование на языке высокого уровня. Учебник для вузов, СПб,: Питер, 2009г.

3 Д. Албахари, Б. Албахари «C# 3.0 справочник» СПб,:«БХВ - Петербург» 2009г

4 В.М. Рябенький и др. Компьютерное управление внешними устройствами через стандартные интерфейсы, Учебное пособие, Олди-плюс, Херсон, 2008г.

5 Презентации лекций по дисциплине «Системное программирование» для магистрантов специальности 6В070400 – смотри портал кафедры ИС http: \\ www.do.ektu.kz

6 Методические указания к лабораторным работам, СРС и СРСП дисциплины «Системное программирование» специальности 6В070400 Портал кафедры ИС http: \\ www.do.ektu.kz

7.2 Дополнительная литература

7 Э. Йодан Структурное программирование и конструирование программ. М.: ”Мир”, 1989г.

8 Д. Кнут. Искусство программирования для ЭВМ. Т.1./ Основные алгоритмы / - М.:Мир,1976.