МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования «Казанский национальный исследовательский технический
университет им. А.Н. Туполева-КАИ»
(КНИТУ-КАИ)
Чистопольский филиал «Восток»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

по дисциплине

ОПТИМИЗАЦИЯ В ПРОГРАММНО- АППАРАТНЫХ СИСТЕМАХ

Индекс по учебному плану: **Б1.В.ДВ.05.01**

Направление подготовки: **09.03.01 Информатика и вычислительная техника**

Квалификация: **Бакалавр**

Профиль подготовки: **Вычислительные машины, комплексы, системы и сети**

Типы задач профессиональной деятельности: **проектный, производственно-технологический**

Рекомендовано УМК ЧФ КНИТУ-КАИ

Чистополь
2023 г.

Лабораторная работа № 1 Потоки

Создание потока

Для оптимизации вычислительных процессов и создания потока необходимо создать объект класса Thread.

Один из конструкторов этого класса имеет следующий вид:

public Thread (ThreadStart *start*)

Объект *start* типа ThreadStart должен содержать адрес точки входа в функцию, которая будет выполняться в потоке.

Тип ThreadStart объявлен в пространстве имен System.Threading с помощью ключевого слова delegate и имеет следующую сигнатуру:

public delegate void ThreadStart();

Согласно, сигнатуре объявления делегата, потоковая функция должна возвращать значение типа void и не принимать параметров.

Пусть, например, заголовок потоковой функции - void funcThread(), тогда создание объекта start:

ThreadStart start = new ThreadStart(funcThread);

Конструктору делегата ThreadStart при создании объекта передается имя функции, которая будет выполняться в потоке.

Объект start после своего создания хранит адрес точки входа в функцию funcThread.

Следующим шагом необходимо создать потоковый объект, передав конструктору класса Thread объект start в качестве параметра:

Thread thrd = new Thread(start);

После создания потокового объекта необходимо запустить потоковую функцию на выполнение путем вызова метода Start, используя ссылку thrd:

```
thrd.Start();
```

Потоковому объекту с помощью свойства Name можно присвоить некоторое имя:

```
thrd.Name = “ ThrName”;,
```

или прочесть:

```
strig nm = thrd.Name;
```

Поток считается активным, не зависимо от того находится ли он в состоянии останова или нет, до тех пор, пока не завершится запущенная в нем функция. После завершения потоковой функции поток уничтожается. Такое завершение работы потока считается нормальным. Узнать, находиться ли поток в активном состоянии или нет можно с помощью свойства IsAlive, доступного только для чтения. Свойство возвращает значение true, если поток активен и ложь (false), если нет. Это свойство можно использовать для определения момента окончания работы потока. Приостановить работу потока на определенное время можно с помощью статического метода:

```
public static  void Sleep(int time)
```

класса Thread, где параметр time задает время в миллисекундах.

Для закрепления материала рассмотрим пример, демонстрирующий изложенный материал.

Пример 1:

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Threading;
namespace ConsoleApplication4
{
   class MyThread
   {
      private int count; //счетчик
                  //объявление ссылки на потоковый объект
                  public Thread thrd;
      public MyThread(string name)
      {
         count = 0;
//здесь создается потоковый объект для функции FnThr
         thrd = new Thread(new ThreadStart(this.FnThr));
         thrd.Name = name; // Устанавливаем имя потока.
         thrd.Start(); // Запускаем поток на выполнение.
      }
//определение потоковой функции
      public void FnThr()
      {
         Console.WriteLine(thrd.Name + " стартовал.");
         do
```

```
        {
            Thread.Sleep(300); //”засыпаем” на 0,3 сек
            Console.WriteLine("В потоке " + thrd.Name + ", count - " + count);
            count++;
        } while (count <= 4);
        Console.WriteLine(thrd.Name + " завершен.");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Главный поток стартовал.");
        // Создаем массив ссылок на потоковые объекты
            MyThread[] mt = new MyThread[5];
          // Создаем объекты класса MyThread.
            for (int j = 0; j < mt.Length; j++)
        mt[j] = new MyThread("Потомок #" + j.ToString());
        bool live;
        do
        {
            live=false;
```

```
            Console.Write(".");
            Thread.Sleep(100); //”спит” главный поток
            for (int j = 0; j < mt.Length; j++)
            {
               live = live | mt[j].thrd.IsAlive;
            }

        } while (live);
        Console.WriteLine("Главный поток завершен.");

    }

}
```

В этом приложении объявлен класс MyThread. Конструктор класса MyThread создает потоковый объект с помощью строки кода:

```
thrd = new Thread(new ThreadStart(this.FnThr));
```

Затем потоковому объекту присваивается имя, которое конструктор принимает в качестве параметра и запускается с помощью метода Start() поток, что приводит к вызову потокового метода FnThr().

В методе FnThr () организован цикл, который "считает" от 0 до 4. До входа в цикл метод выводит на консоль имя стартовавшего потока, а после окончания цикла – имя завершившего свою работу потока. Вызов метода

Sleep(), в теле цикла метода FnThr () заставляет поток, из которого он был вызван, приостановить выполнение на период времени, заданный в миллисекундах. В методе Main() при выполнении следующих инструкций создается пять объектов класса Thread:

```
// Создаем массив ссылок на потоковые объекты
    MyThread[] mt = new MyThread[5];
// Создаем объекты класса MyThread.
    for (int j = 0; j < mt.Length; j++)
mt[j] = new MyThread("Потомок #" + j.ToString());
```

После создания потоков и запуска их на выполнение в конструкторе класса MyThread , в приложении выполняются основной поток, а, именно, метод Main() и пять дочерних потоков.

С помощью цикла:

```
for (int j = 0; j < mt.Length; j++)
{
   live = live | mt[j].thrd.IsAlive;
}

} while (live);
```

метод Main() дожидается завершения работы потоков.

Остановка и возобновление работы потоков

Запущенный на выполнение поток может быть остановлен путем вызова метода Suspend(), а возобновление выполнения – путем вызова метода Resume():

```
thr1.Suspend(); // останов потока
thr1.Resume();   // восстановление работы потока
```

Для аварийного завершения работы потока служит метод Abort(). Этот метод имеет два формата:

```
public void Abort()
public void Abort(object info)
```

Метод Abort ()завершает работу потока и генерирует исключение типа ThreadAbortException для потока.

Исключение может быть перехвачено программным кодом. С помощью параметра *info* в программный блок catch, обрабатывающий данное исключение, может быть передана информация о причине завершения потока.

Пример 2:

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

```
namespace ThreadAbort
{
   // Класс для установки и чтения текущей температуры
   class Grd
   {
      int grade =0;// температура в градусах
      //  установка и чтение текущей температуры
      public int Grade
      {
         get
         {
            return grade;
         }
         set
         {
            grade = value;
         }

      }
   }
   class MyThread
   {
```

```
Grd gr;          // Ссылка на объект "разогрева"
public Thread th; // Поток  для "разогрева" объекта
public MyThread(Grd grd)
{
    gr = grd;// Получаем ссылку на объект "разогрева"
    th = new Thread(new ThreadStart(FnThr));
    th.Start();
}
// Функция которая греет объект
void FnThr()
    {   //здесь мы греем и можем "перегреться"
        try //блок,где может быть вызвано исключение
        {
            for (int j = 0; j < 10; j++)
            {
                Console.WriteLine("Температура={0}", gr.Grade);
                lock(gr) gr.Grade += 10;
                Thread.Sleep(100); // Передаем управление другому потоку
            }
        }
            // Перехватываем исключение и печатаем причину
        catch (ThreadAbortException abortException)
```

```
            {
                Console.WriteLine((string)abortException.ExceptionState);
            }

        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            Grd gr = new Grd(); //Создаем объект для разогрева
            MyThread thr = new MyThread(gr);//Запускаем процесс обогрева - поток
            int grd;
            lock(gr) grd = gr.Grade;
            // Следим за процессом нагрева
            while ( grd <= 70)
                lock (gr) grd = gr.Grade;
            thr.th.Abort("Перегрелись");// хватит греть
        }
```

```
    }
}
```

В программе, для синхронизации работы потоков использован метод lock().

Синхронизация работы потоков

Синхронизацией работы потоков называется обеспечение корректной работы нескольких потоков с общими (разделяемыми) данными или ресурсами.

Синхронизация обеспечивается путем организации монопольного доступа одного из потоков на время работы с разделяемыми ресурсами, и блокирования доступа к разделяемым ресурсам на это время со стороны других потоков.

В основе синхронизации лежит понятие *блокировки,* т.е. управление доступом к некоторому блоку кода в объекте. На то время, когда объект заблокирован одним потоком, никакой другой поток не может получить доступ к блоку кода для работы с этим объектом. Когда поток снимет блокировку, объект станет доступным для использования другим потоком.

Средство блокировки встроено в язык C#, поэтому доступ ко всем объектам может быть синхронизирован. Синхронизация поддерживается ключевым словом lock.

Формат использования инструкции lock таков:

```
lock(object)
 {
 // Инструкции, подлежащие блокировки.
 }
```

Здесь параметр *object* представляет собой ссылку на синхронизируемый объект.

Инструкция lock гарантирует, что указанный блок кода, защищенный блокировкой для данного объекта, может быть использован только потоком, который получает эту блокировку. Инструкция lock может применяться не только к объектам, а и к статическим функциям класса. Предположим, что в классе CA имеется статическая функция, объявленная следующим образом:

```
static void Min(int a, int b)
    {
        :
    },
```

Тогда обращение к этой функции с применением блокировки:

```
lock(typeof(CA))  CA.Min(x,y);
```

Рассмотрим пример многопоточного приложения под Windows. В момент запуска приложения в окне приложения появляется десять “шаров”. Координата X всех шаров в момент появления равна нулю, а координата Y рассчитывается как величина равная 10*j, где j порядковый номер шарика. Шарики сразу же начинают двигаться вниз под углом 45 градусов с разными скоростями. Скорость первого шарика принимается за единицу, скорости всех последующих шаров равны скорости первого шарика умноженной на номер шарика. Шарики движутся в области ограниченной координатами 200 на 200 пикселей, отражаясь от границ области по законам физики. Пересчет координат шариков должен осуществляться потоковой функцией. Окно приложения содержит кнопку, при щелчке на которую потоки завершают свою работу, а шарики исчезают с экрана.

Пример 3:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
namespace Balls
{
   public partial class Form1 : Form
   {
      class Ball
      {
         int x, y;   // координаты
         int dx, dy; //приращение координат-определяет скорость
         int w, h;   //ширина высота шарика
         public bool live;   // признак жизни
         public delegate void DlTp();// Объявление типа (делегат) и
         //создание пока что пустой ссылки для организации в последующем
         // с помощью ее вызова функции Invalidate()для главного потока
         public DlTp dl;
```

```
public Thread thr;  //Создание ссылки на потоковый объект
// потоковая функция
void FnThr()
{
   while (live)
   {   //здесь отражемся от границ области
      if (x < 0 || x > 200) dx = -dx;
      if (y < 0 || y > 200) dy = -dy;
      //здесь пересчитываем координаты
      x += dx;
      y += dy;
      Thread.Sleep(100);//спим
      dl();   //вызываем с помощью делегата Invalidate()
   }
   w = h = 0;  //схлопываем шарик
   dl();      //вызываем с помощью делегата Invalidate()
}
//функция рисования шарика
public void DrawBall(Graphics dc)
{
   dc.DrawEllipse(Pens.Magenta, x, y, w, h);
}
```

```
    //конструктор класса
    public Ball(int xn, int yn, int wn, int hn, int dxn, int dyn)
    {
     x = xn; y = yn; w = wn; h = hn; dx = dxn; dy = dyn;//инициализируем
     thr = new Thread(new ThreadStart(FnThr)); //создаем потоковый объект
     live = true;    //устанавливаем признак жизни
     thr.Start();    //запускаем поток
    }
}
Ball[] bl = new Ball[10];//массив пустых ссылок типа Ball
public Form1()
{
    InitializeComponent();
    for (int j = 0; j < bl.Length; j++)
    {
        //создаем потоковые объекты
        bl[j] = new Ball(j, j * 10, 10, 10, j + 1, j + 1);
        //подписываемся на событие
        bl[j].dl += new Ball.DlTp(Invalidate);
    }
}
private void Form1_Paint(object sender, PaintEventArgs e)
```

```
{
    for (int j = 0; j < bl.Length; j++)
    {
        bl[j].DrawBall(e.Graphics);//рисуем
    }

}
private void button1_Click(object sender, EventArgs e)
{
    for (int j = 0; j < bl.Length; j++)
    {
        bl[j].live = false;// Уничтожаем потоки
    }
}

private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    for (int j = 0; j < bl.Length; j++)
    {
        bl[j].live = false;//уничтожаем потоки
    }
}
```

```
    }
}
```

Скомпилируйте и выполните приложение. Проанализируйте работу приложения.

Задания:

1. В приложение по примеру 3 добавить редактор TextBox и две кнопки. При щелчке на первую кнопку шарик с номером, указанным в TextBox останавливатся, а при щелчке на вторую продолжает движение.
2. В приложение по примеру 3 добавить редактор TextBox и две кнопки. При щелчке на первую кнопку шарик с номером, указанным в TextBox изменяет цвет на красный, а при щелчке на вторую цвет восстанавливается.
3. В приложение по примеру 3 добавить редактор TextBox и две кнопки. При щелчке на первую кнопку шарик с номером, указанным в TextBox исчезает с экрана, а при щелчке на вторую восстанавливается.
4. В приложение по примеру № 3 добавить вертикальную область. Внутри области может находиться только один шарик, другие шарики ожидают, пока область не освободится. Применить синхронизацию потоков.
5. В приложении по примеру № 3 выделить функцию пересчета координат в отдельный класс. Создать единственный объект этого класса. Все потоковые объекты должны обращаться к функции этого объекта для пересчета координат. Применить синхронизацию потоков.
6. Создать консольное приложение. В нем два потока обращаются к одному и тому же объекту. Метод этого объекта принимает в качестве параметра массив и возвращает сумму элементов массива. В процессе расчета суммы печатается промежуточная сумма и имя потока, для которого

производится расчет. Потоки по окончанию работы печатают полученный результат работы (сумму) и свое имя.

7. Создать консольное приложение. В нем два потока обращаются к одному и тому же объекту. Метод этого объекта распечатывает внутренний массив объекта и имя потока, для которого производится печать. Обеспечить синхронизацию работы потоков.
8. Создать консольное приложение. В нем запускаются два независимых потока. Первый поток в цикле осуществляют инкремент внутренней переменной, которая изначально была равна нулю. Второй поток ожидает, пока значение этой переменной не станет равно 1000 и завершает работу первого потока, а затем и свою. В момент завершения работы оба потока печатают сообщение.
9. Создать Windows приложение. Два потока обращаются к одному объекту. Объект принимает координаты начала и конца линии и рисует их на экране. Каждый поток должен с помощью этого объекта нарисовать замкнутую фигуру, каждый в своей области экрана.
10. Создать Windows приложение. Четыре потока обращаются к статическому методу класса, для рисования линии. Метод принимает координаты начала и конца линии. . Каждый поток должен с помощью этого объекта нарисовать замкнутую фигуру, каждый в своей области экрана.

Лабораторная работа № 2 .

Знакомство с многопоточной обработкой для оптимизации вычислительных процессов

Задания

1. Реализуйте оптимизированную последовательную обработку элементов вектора, например, умножение элементов вектора на число. Число элементов вектора задается параметром N.
2. Реализуйте оптимизированную многопоточную обработку элементов вектора, используя разделение вектора на равное число элементов. Число потоков задается параметром M.
3. Выполните анализ эффективности оптимизации с помощью многопоточной обработки при разных параметрах N (10, 100, 1000, 100000) и M (2, 3, 4, 5, 10). Результаты представьте в табличной форме.
4. Выполните анализ эффективности при усложнении обработки каждого элемента вектора.
5. Исследуйте эффективность оптимизации с помощью разделения по диапазону при неравномерной вычислительной сложности обработки элементов вектора.
6. Исследуйте эффективность оптимизации с помощью параллелизма при круговом разделении элементов вектора. Сравните с эффективностью разделения по диапазону.

В работе исследуется эффективность оптимизации с помощью распараллеливания независимой обработки элементов вектора. В первом задании в качестве обработки можно выбрать то или иное математическое преобразование элементов вектора:

```
for(int i=0; i<a.Length; i++)
```

```
b[i] = Math.Pow(a[i], 1.789);
```

Многопоточная обработка реализуется с помощью объектов Thread. На многоядерной системе многопоточная обработка приводит к параллельности выполнения. Классы для работы с потоками расположены в пространстве имен System.Threading.

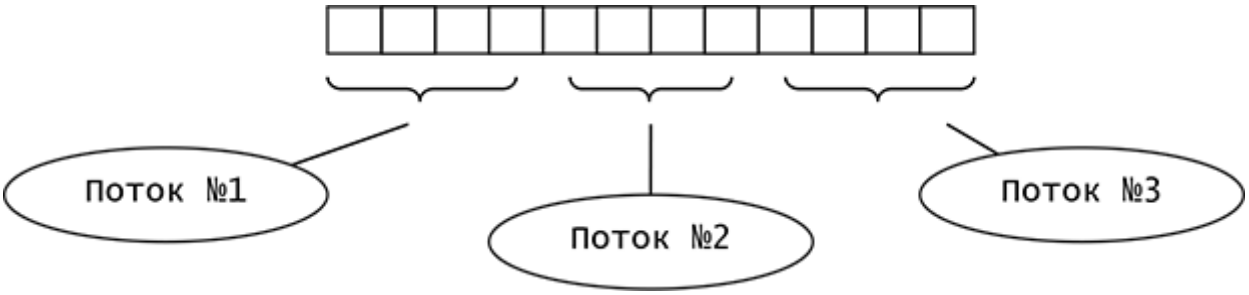
Для создания потока необходимо указать имя рабочего метода потока, который может быть реализован в отдельном классе, в главном классе приложения как статический метод или в виде лябда-выражения. Метод потока либо не принимает никаких аргументов, либо принимает аргумент типа object. Запуск потока осуществляется вызовом метода Start.

```
class Program
{
  static void Run(object some_data)
  {
    int m = (int) some_data;
    ..
  }
  static void Main()
  {
    ..
    Thread thr = new Thread(Run);
    thr.Start(some_data);
  }
}
```

Дождаться завершения работы потоков можно с помощью метода Join:

```
thr1.Join(); thr2.Join();
```

В функции потока необходимо предусмотреть возможность разбиения диапазона 0.. (N-1) на число потоков nThr. При запуске потока в качестве аргумента передается либо "индекс потока", определяющий область массива, который обрабатывается в данном потоке, либо начальный и конечный индексы массива.



Многопоточное выполнение будет оптимальным и параллельным при наличии в вычислительной системе нескольких процессоров (ядер процессора). Число процессоров можно узнать с помощью свойства:

System.Environment.ProccessorCount;

Параллельное выполнение вычислений также можно реализовать с помощью классов библиотеки TPL (Task Parallel Library). Классы библиотеки располагаются в пространстве имен System.Threading.Tasks. Параллельное вычисление операций над элементами цикла выполняется с помощью метода Parallel.For:

```
Parallel.For(0, a.Length, i =>
  { b[i] = Math.Pow(a[i], 1.789); });
```

Для анализа производительности последовательного и параллельного выполнения можно использовать переменные типа DateTime. Например,

```
DateTime dt1, dt2;
dt1 = DateTime.Now;
```

```
// Вызов_вычислительной_процедуры;
dt2 = DateTime.Now;
TimeSpan ts = dt2 – dt1;
Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
```

Также можно использовать объект Stopwatch пространства System. Diagnostics:

```
Stopwatch sw = new Stopwatch();
sw.Start();
// Вызов_вычислительной_процедуры;
sw.Stop();
TimeSpan ts = sw.Elapsed;
Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
```

При оценке производительности необходимо учесть, что время выполнения алгоритма зависит от множества параметров. Поэтому желательно оценивать среднее время выполнения при нескольких прогонах алгоритма, исключая первый разогревающий прогон.

Эффективность оптимизации вычислительного процесса параллельного алгоритма существенно зависит от элементов массива, числа потоков, сложности математической функции и т.д. Следует учитывать, что при малом объеме элементов массива, накладные расходы, связанные с организацией многопоточной обработки, превышают выигрыш от параллельности обработки. При последовательном выполнении примитивной циклической обработки быстродействие достигается за счет оптимального использования кэш-памяти.

Выполняя анализ зависимости быстродействия от числа потоков, следует учитывать число ядер процессора. Увеличение числа потоков сверх возможностей вычислительной системы приводит к конкуренции потоков и ухудшению быстродействия.

Усложнение обработки элементов массива предлагается реализовать с помощью внутреннего цикла. Например,

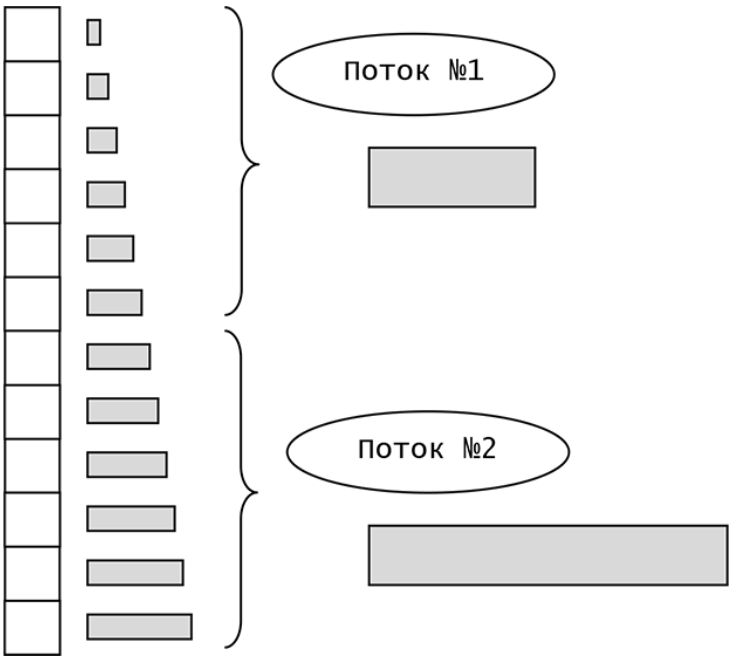
```
for(int i=0; i<a.Length; i++)
 {
 // Обработка i-элемента
 for(int j=0; j < K; j++)
   b[i] += Math.Pow(a[i], 1.789);
}
```

K – параметр "сложности". Увеличивая параметр K, наблюдаем повышение эффективности оптимизации вычислительного процесса параллельной обработки при меньшем объеме массива чисел.

В рассмотренных вариантах обработки вычислительная нагрузка на каждой итерации относительно одинакова. В ситуациях, когда вычислительная нагрузка зависит от индекса элемента, разделение массива по равным диапазонам может быть не эффективно. Рассмотрим следующий вариант обработки:

```
for(int i=0; i<a.Length; i++)
{
 // Обработка i-элемента
 for(int j=0; j <i; j++)
   b[i] += Math.Pow(a[i], 1.789);
}
```

Вычислительная нагрузка при обработке i-элемента зависит от индекса i. Обработка начальных элементов массива занимает меньшее время по сравнению с обработкой последних элементов. Разделение данных по диапазону приводит к несбалансированной загрузке потоков и снижению эффективности распараллеливания.



Одним из подходов к выравниванию загрузки потоков является применение круговой декомпозиции. В случае двух потоков получаем такую схему: первый поток обрабатывает все четные элементы, второй поток обрабатывает все нечетные элементы. Реализуйте круговую декомпозицию для нескольких потоков (больше двух).

Вопросы:

1. Почему эффект оптимизации вычислительного процесса от распараллеливания наблюдается только при большем числе элементов?
2. Почему увеличение сложности обработки повышает эффективность оптимизации вычислительного процесса многопоточной обработки?
3. Какое число потоков является оптимальным для конкретной вычислительной системы?
4. Почему неравномерность загрузки потоков приводит к снижению эффективности оптимизации вычислительного процесса многопоточной обработки?
5. Какие другие варианты декомпозиции позволяют увеличить равномерность загрузки потоков?

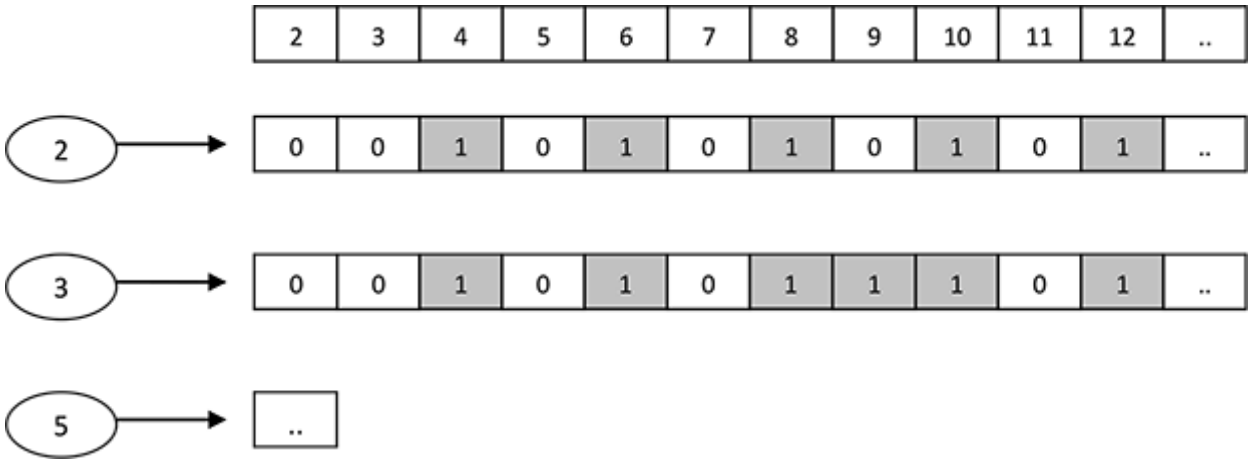
6. В какой ситуации круговая декомпозиция не обеспечивает равномерную загрузку потоков?

Лабораторная работа № 3 Поиск простых чисел

Реализовать последовательный и параллельные алгоритмы поиска простых чисел; выполнить анализ оптимизации вычислительного процесса и быстродействия алгоритмов при разном объеме данных, разном числе потоков; рассчитать ускорение и эффективность оптимизации выполнения алгоритмов; сделать выводы о целесообразности применения параллельных алгоритмов и необходимости использования синхронизации.

Последовательный алгоритм "Решето Эратосфена".

Алгоритм заключается в последовательном переборе уже известных простых чисел, начиная с двойки, и проверке разложимости всех чисел диапазона $(m, n]$ на найденное простое число m. На первом шаге выбирается число $m = 2$, проверяется разложимость чисел диапазона $(2, n]$ на 2-ку. Числа, которые делятся на двойку, помечаются как составные и не участвуют в дальнейшем анализе. Следующим непомеченным (простым) числом будет $m = 3$, и так далее.



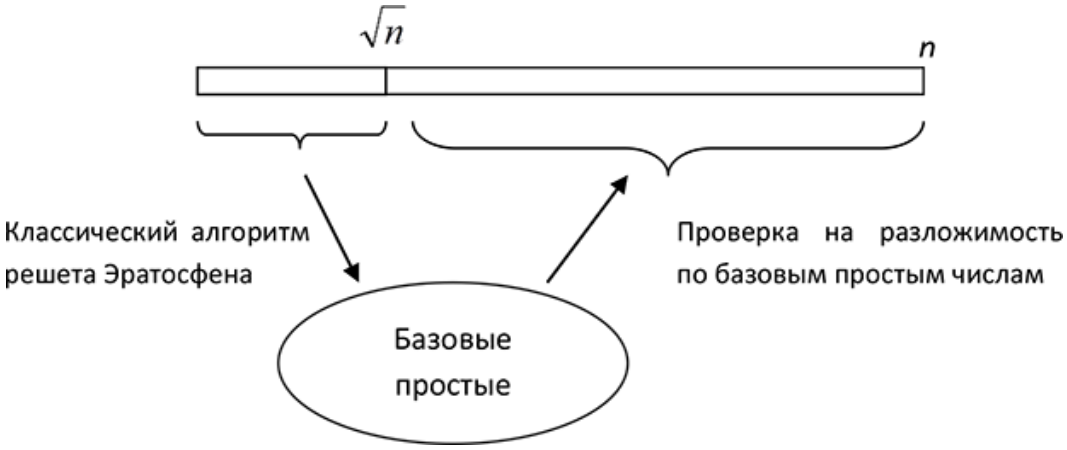
При этом достаточно проверить разложимость чисел на простые числа в интервале $(2, \sqrt{n}]$. Например, в интервале от 2 до 20 проверяем все числа на разложимость 2, 3. Составных чисел, которые делятся только на пятерку, в этом диапазоне нет.

Модифицированный последовательный алгоритм поиска

В последовательном алгоритме "базовые" простые числа определяются поочередно. После тройки следует пятерка, так как четверка исключается при обработке двойки. Последовательность нахождения простых чисел затрудняет распараллеливание алгоритма. В модифицированном алгоритме выделяются два этапа:

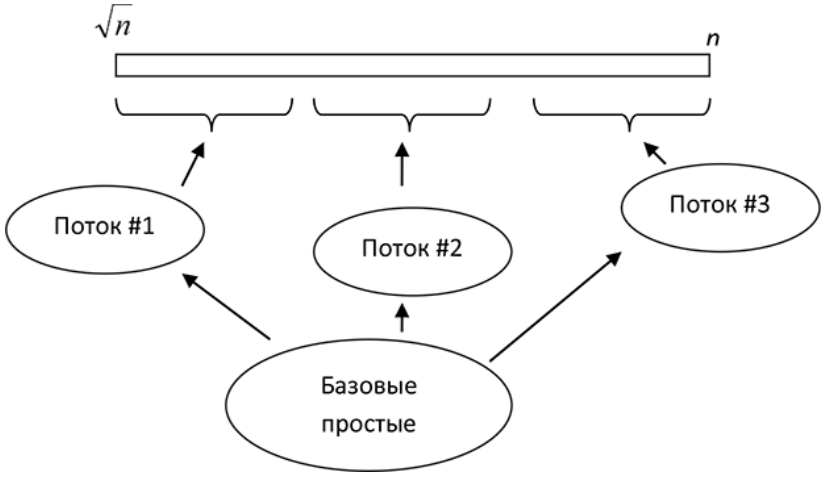
1-ый этап: поиск простых чисел в интервале от $2 \dots \sqrt{n}$ с помощью классического метода решета Эратосфена (базовые простые числа).

2-ой этап: поиск простых чисел в интервале от $\sqrt{n} \dots n$, в проверке участвуют базовые простые числа, выявленные на первом этапе.



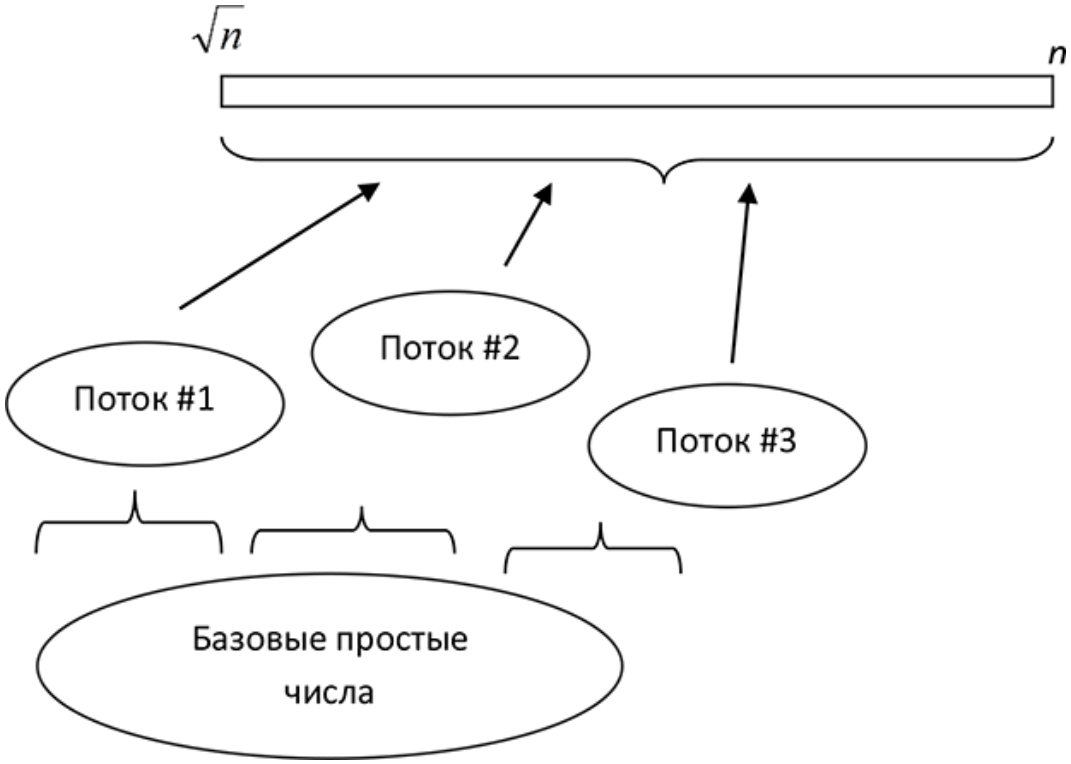
На первом этапе алгоритма выполняется сравнительно небольшой объем работы, поэтому нецелесообразно распараллеливать этот этап. На втором этапе проверяются уже найденные базовые простые числа. Параллельные алгоритмы разрабатываются для второго этапа.

Параллельный алгоритм №1: декомпозиция по данным



Идея распараллеливания заключается в разбиении диапазона $\sqrt{n} \ldots n$ на равные части. Каждый поток обрабатывает свою часть чисел, проверяя на разложимость по каждому базовому простому числу.

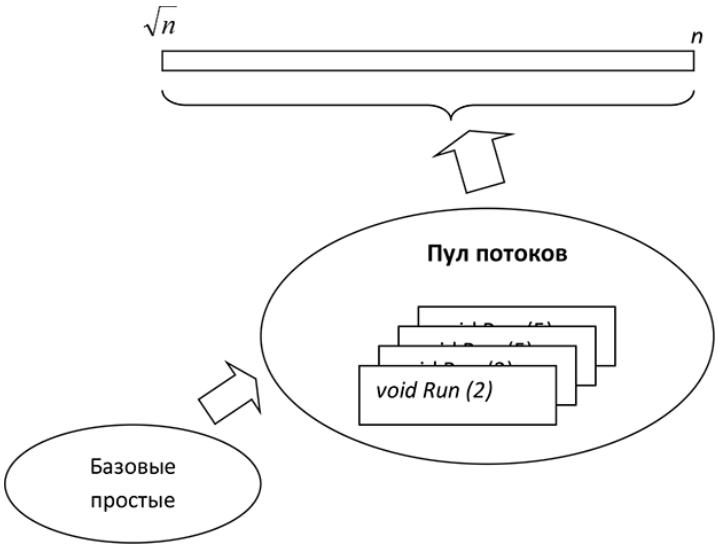
Параллельный алгоритм №2: декомпозиция набора простых чисел



В этом алгоритме разделяются базовые простые числа. Каждый поток работает с ограниченным набором простых чисел и проверяет весь диапазон $\sqrt{n} \ldots n$.

Параллельный алгоритм №3: применение пула потоков

Применение пула потоков позволяет автоматизировать обработку независимых рабочих элементов. В качестве рабочих элементов предлагается использовать проверку всех чисел диапазона от $\sqrt{n} \ldots n$ на разложимость по одному базовому простому числу.



Для применения пула потоков необходимо загрузить рабочие элементы вместе с необходимыми параметрами в очередь пула потоков:

```
for(int i=0; i<basePrime.Length; i++)
 {
    ThreadPool.QueueUserWorkItem(Run, basePrime[i]);
 }
```

Run – метод обработки всех чисел диапазона $\sqrt{n} \ldots n$ на разложимость простому числу basePrime[i].

Выполнение рабочих элементов осуществляется автоматически после добавления в пул потоков. Не существует встроенного механизма ожидания завершения рабочих элементов, добавленных в пул потоков. Поэтому вызывающий поток (метод Main) должен контролировать завершение либо с помощью средств синхронизации (например, сигнальных сообщений), либо с помощью общих переменных и цикла ожидания в методе Main.

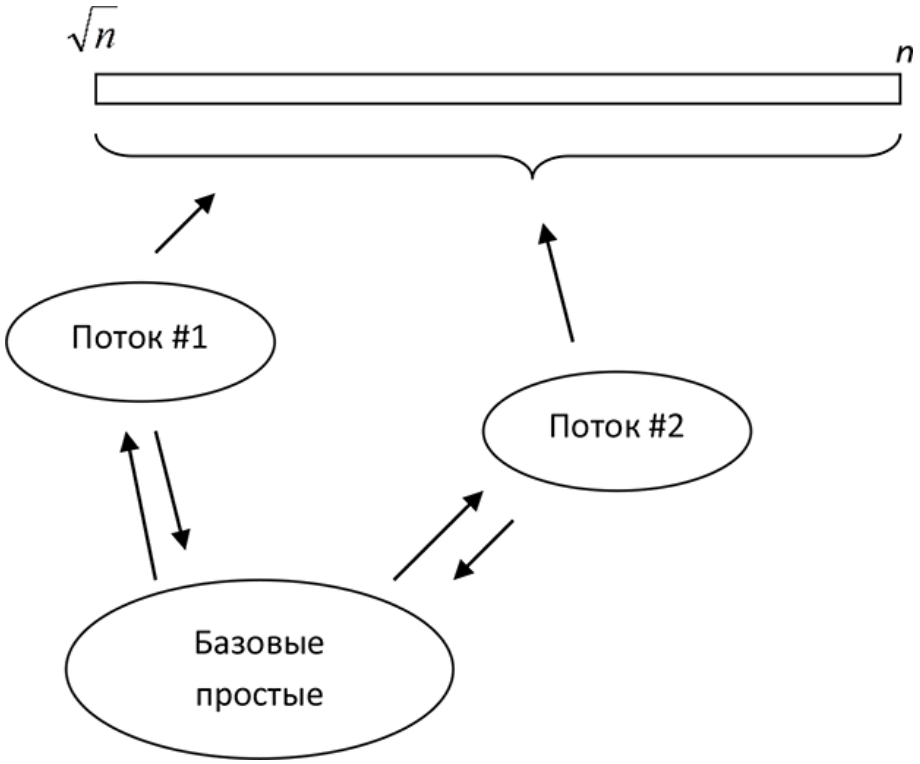
Применение сигнальных сообщений может быть реализовано следующим образом:

```
static void Main()
{
 // Поиск базовых простых
```

```
  ..
  int[] basePrime = ..

  // Объявляем массив сигнальных сообщений
  ManualResetEvent [] events =
    new ManualResetEvent [basePrime.Length];

   // Добавляем в пул рабочие элементы с параметрами
  for(int i=0; i<basePrime.Length; i++)
  {
    events[i] = new ManualResetEvent(false);
    ThreadPool.QueueUserWorkItem(Run,
    new object[] {basePrime[i], events[i]})
  }
  // Дожидаемся завершения
  WaitHandle.WaitAll(events);

  // Выводим результаты
  ..
}
static void F(object o)
{
  int prime = (int)((object[])o)[0];
  ManualResetEvent ev = ((object[])o)[1] as ManualResetEvent;
  // Обработка чисел на разложимость простому числу prime
  ..
  ev.Set();
}
```

Параллельный алгоритм №4: последовательный перебор простых чисел

Идея алгоритма заключается в последовательном переборе базовых простых чисел разными потоками. Каждый поток осуществляет проверку всего диапазона на разложимость по определенному простому числу. После обработки первого простого числа поток не завершает работу, а обращается за следующим необработанным простым числом.



Для получения текущего простого числа поток выполняет несколько операторов:

```
while(true)
{
 if (current_index >= basePrime.Length)
   break;
 current_prime = basePrime[current_index];
 current_index++;
 // Обработка текущего простого числа
 ..
}
```

В этой реализации существует разделяемый ресурс – массив простых чисел. При одновременном доступе к ресурсу возникает проблема гонки данных. Следствием этой проблемы являются: лишняя обработка, если несколько потоков одновременно получают одно и то же число; пропущенная задача - потоки, получив одно число, последовательно увеличивают текущий индекс; исключение "Выход за пределы массива", когда один поток успешно прошел проверку текущего индекса, но перед обращением к элементу массива, другой потокувеличивает текущий индекс.

Для устранения проблем с совместным доступом необходимо использовать средства синхронизации (критические секции, атомарные операторы, потокобезопасные коллекции).

Критическая секция позволяет ограничить доступ к блоку кода, если один поток уже начал выполнять операторы секции:

```
lock (sync_obj)
{
  критическая_секция
}
```

где sync_obj – объект синхронизации, идентифицирующий критическую секцию (например, строковая константа).

Задания на лабораторную работу

1. Какими достоинствами и недостатками обладает каждый вариант распараллеливания для оптимизации вычислительного процесса ?
2. Какие средства синхронизации можно использовать вместо конструкции lock? Какой вариант будет более эффективным?
3. Какой вариант ожидания завершения работ, запущенных пулом потоков, более эффективный с точки зрения оптимизации вычислительного процесса и почему?

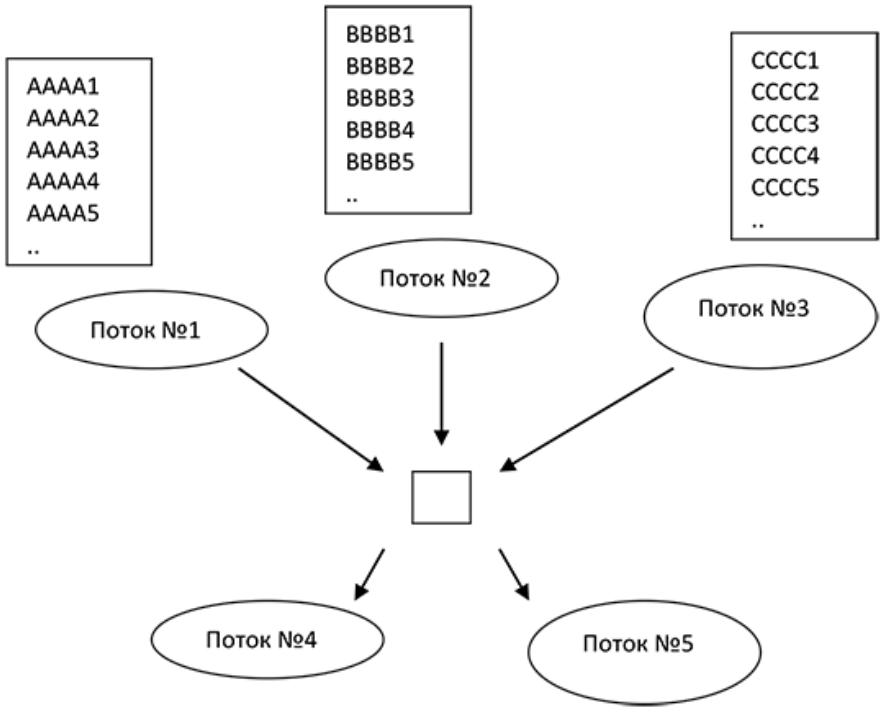
4. Реализуйте один или несколько вариантов распараллеливания с помощью объектов Task и с помощью метода Parallel.For. Выполните анализ эффективности алгоритмов оптимизации вычислительного процесса .

5. Реализуйте алгоритм поиска простых чисел как LINQ-запрос к массиву чисел.

Лабораторная работа № 4.

Синхронизация доступа к одноэлементному буферу для оптимизации вычислительного процесса

Несколько потоков работают с общим одноэлементным буфером. Потоки делятся на "писателей", осуществляющих запись сообщений в буфер, и "читателей", осуществляющих извлечение сообщений из буфера. Только один поток может осуществлять работу с буфером. Если буфер свободен, то только один писатель может осуществлять запись в буфер. Если буфер занят, то только один читатель может осуществлять чтение из буфера. После чтения буфер освобождается и доступен для записи. В качестве буфера используется глобальная переменная, например, типа string. Работа приложения заканчивается после того, как все сообщения писателей через общий буфер будут обработаны читателями.



Задание на лабораторную работу

1. Реализуйте взаимодействие потоков-читателей и потоков-писателей с общим буфером без каких-либо средств синхронизации для оптимизации

вычислительного процесса. Проиллюстрируйте проблему совместного доступа. Почему возникает проблема доступа?

2. Реализуйте доступ "читателей" и "писателей" к буферу с применением следующих средств синхронизации:

   a. блокировки (lock);

   b. сигнальные сообщения (ManualResetEvent, AutoResetEvent, ManualResetEventSlim);

   c. семафоры (Semaphore, SemaphoreSlim).

   d. атомарные операторы (Interlocked)

3. Исследуйте производительность средств синхронизации при разном числе сообщений, разном объеме сообщений, разном числе потоков.

4. Сделайте выводы об эффективности применения средств синхронизации для оптимизации вычислительного процесса.

В случае одноэлементного буфера достаточно использовать флаг типа bool для контроля состояния буфера. Читатели обращаются к буферу, только если он свободен:

```
// Работа читателя
while (!finish)
{
if (!bEmpty)
{
MyMessages.Add(buffer);
bEmpty = true;
}
}
```

Писатели обращаются к буферу, только если он пуст:

```
// Работа писателя
```

```
while(i < n)
{
if (bEmpty)
{
buffer = MyMessages[i++];
bEmpty = false;
}
}
```

Писатели работают, пока не запишут все свои сообщения. По окончании работы писателей основной поток может изменить статус переменной finish, который является признаком окончания работы читателей.

```
static void Main()
{
// Запускаем читателей и писателей
..
// Ожидаем завершения работы писателей
for(int i=0; i< writers.Length; i++)
writers[i].Join();
// Сигнал о завершении работы для читателей
finish = true;

// Ожидаем завершения работы читателей
for(int i=0; i< readers.Length; i++)
readers[i].Join();

}
```

Отсутствие средств синхронизации при обращении к буферу приводит к появлению гонки данных – несколько читателей могут прочитать одно и то же сообщение, прежде чем успеют обновить статус буфера; несколько писателей могут одновременно осуществить запись в буфер. В данной задаче следствием гонки данных является потеря одних сообщений и дублирование других. Для фиксации проблемы предлагается выводить на экран число повторяющихся и потерянных сообщений.

Самый простой вариант решения проблемы заключается в использовании критической секции (lock или Monitor).

```
// Работа читателя
while (!finish)
{
lock ("read")
{
if (!bEmpty)
{
MyMessage[i++] = buffer;
bEmpty = true;
}
}
}
```

Для писателей существует своя критическая секция:

```
// Работа писателя
while(i < n)
{
lock("write")
{
```

```
if (bEmpty)
{
buffer = MyMessage[i++];
bEmpty = false;
}
}
}
```

Данная реализация не является оптимальной. Каждый из читателей поочередно входит в критическую секцию и проверяет состояние буфера, в это время другие читатели блокируются, ожидая освобождения секции. Если буфер свободен, то синхронизация читателей избыточна. Более эффективным является вариант двойной проверки:

```
// Работа читателя
while (!finish)
{  if (!bEmpty)
{
lock ("read")
{
if (!bEmpty)
{
bEmpty = true;
MyMessage[i++] = buffer;
}
}
}
}
```

Если буфер свободен, то читатели "крутятся" в цикле, проверяя состояние буфера. При этом читатели не блокируются. Как только буфер заполняется, несколько читателей, но не все, успевают войти в первый if-блок, прежде чем самый быстрый читатель успеет изменить статус буфера bEmpty = true.

Применение сигнальных сообщений позволяет упростить логику синхронизации доступа. Читатели ожидают сигнала о поступлении сообщения, писатели – сигнала об опустошении буфера. Читатель, освобождающий буфер, сигнализирует об опустошении. Писатель, заполняющий буфер, сигнализирует о наполнении буфера. Сообщения с автоматическим сбросом AutoResetEvent обладают полезным свойством – при блокировке нескольких потоков на одном и том же объекте AutoResetEvent появление сигнала освобождает только один поток, другие потоки остаются заблокированными. Порядок освобождения потоков при поступлении сигнала не известен, но в данной задаче это не существенно.

```
// Работа читателя
void Reader(object state)
{
var evFull = state[0] as AutoResetEvent;
var evEmpty = state[1] as AutoResetEvent;
while(!finish)
{
evFull.WaitOne();
MyMessage.Add(buffer);
evEmpty.Set();
}

// Работа писателя
```

```
void Writer(object state)
{
var evFull = state[0] as AutoResetEvent;
var evEmpty = state[1] as AutoResetEvent;
while(i < n)
{
evEmpty.WaitOne();
buffer = MyMessage[i++];
evFull.Set();
}
}
```

Данный фрагмент приводит к зависанию работы читателей. Писатели закончили работу, а читатели ждут сигнала о наполненности буфера evFull. Для разблокировки читателей необходимо сформировать сигналы evFull.Set() от писателей при завершении работы или от главного потока. Чтобы отличить ситуацию завершения можно осуществлять проверку статуса finish непосредственно после разблокировки.

```
// Рабочий цикл читателей
while(true)
{
evFull.Wait();
// Сигнал о завершении работы
if(finish) break;
MyMessage.Add(buffer);
evEmpty.Set();
}
```

Применение семафоров (Semaphore, SemaphoreSlim) в данной задаче аналогично использованию сигнальных сообщений AutoResetEvent. Кроме предложенного варианта обмена сигналами между читателями и писателями, семафоры и сигнальные сообщения могут использоваться в качестве критической секции читателей и писателей.

```
void Reader(object state)
{
var semReader = state as SemaphoreSlim;
while(!finish)
{
if(!bEmpty)
{
semReader.Wait();
if(!bEmpty)
{
bEmpty = true;
myMessages.Add(buffer);
}
semReader.Release();
}
}
}

void Writer(object state)
{
var semWriter = state as SemaphoreSlim;
while(i < myMessages.Length)
{
if(bEmpty)
```

```
{
semWriter.Wait();
if(bEmpty)
{
bEmpty = false;
buffer = myMessages[i];
}
semWriter.Release();
}
}
}
```

Вопросы и упражнения

1. Почему проблема гонки данных проявляется не при каждом прогоне?

2. Какие факторы увеличивают вероятность проявления проблемы гонки данных?

3. Возможно ли в данной задаче при отсутствии средств синхронизации возникновение исключения и аварийное завершение программы?

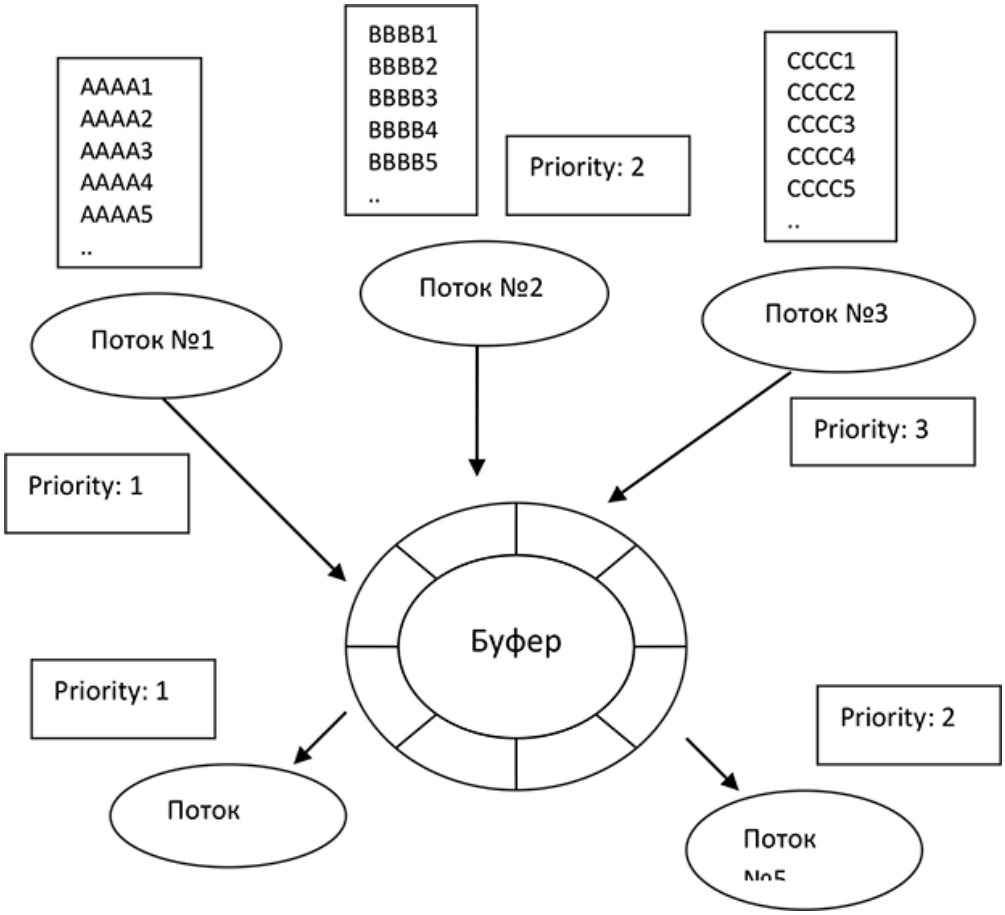
4. Можно ли в данной задаче использовать атомарные операторы для обеспечения согласованности доступа? Необходимы ли при этом дополнительные средства синхронизации?

5. Можно ли в данной задаче использовать потокобезопасные коллекции для обеспечения согласованного доступа?

6. Какие средства синхронизации обеспечивают наилучшее быстродействие в данной задаче для оптимизации вычислительного процесса? Объясните с чем это связано.

Лабораторная работа № 5 Синхронизация приоритетного доступа к многоэлементному буферу для оптимизации вычислительного процесса

Несколько потоков работают с общим многоэлементным буфером. Потоки делятся на "читателей" и "писателей", каждый поток обладает приоритетом. Писатели осуществляют запись в буфер, если есть свободные ячейки. Читатели извлекают содержимое буфера, если есть заполненные ячейки. Работа приложения заканчивается после того, как все сообщения писателей будут обработаны читателями через общий буфер. В качестве буфера используется "кольцевой массив".



Задание на лабораторную работу

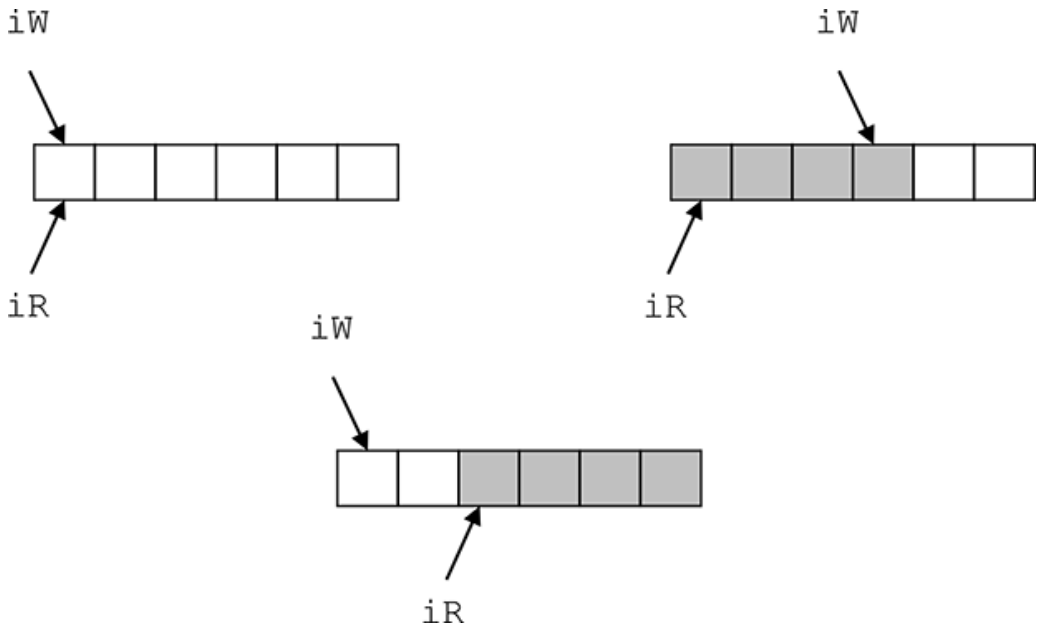
Реализуйте синхронизированное взаимодействие читателей и писателей с учетом приоритета для оптимизации вычислительного процесса. Аргументируйте выбор средств синхронизации.

Вывод программы включает: время работы каждого писателя и читателя; число сообщений, обработанных каждым писателем и читателем.

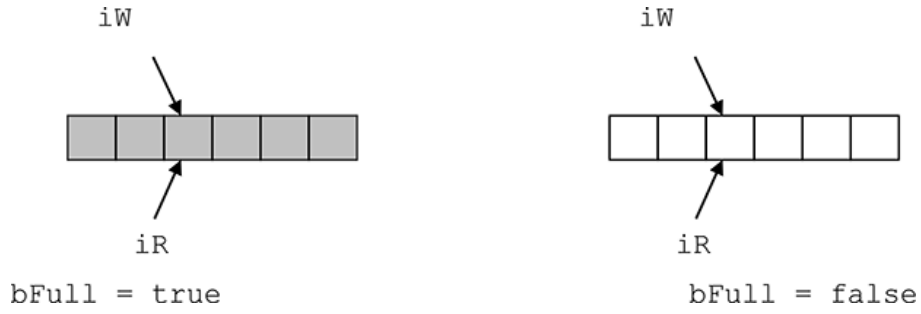
Выполните прогон программы при разных параметрах: разном числе писателей и читателей, разном объеме сообщений, разных приоритетах потоков. Результаты прогонов представьте в табличной форме.

В качестве многоэлементного буфера используется кольцевой массив. Он представляет собой обычный массив размера n. Буфер называется кольцевым, так как при смещении текущего индекса после крайнего элемента следует первый. Для доступа к буферу используются два индекса: один для чтения и один для записи. Такая организация обеспечивает независимость операций чтения и записи – если в массиве есть свободные элементы и есть заполненные элементы, то операции чтения и записи могут производиться одновременно без каких-либо средств синхронизации.

В начале работы буфер является пустым – оба индекса указывают на первый элемент. При осуществлении операций чтения или записи соответствующие индексы смещаются.



Операция чтения блокируется, если буфер пуст, запись при этом разрешена. Операция записи блокируется, если буфер полностью заполнен, чтение при этом разрешено. Равенство индексов чтения и записи является признаком и занятости буфера, и пустоты. Чтобы различать эти ситуации необходимо контролировать, какая операция привела к равенству индексов. Если операция записи, то буфер заполнен.



Операции чтения и записи могут быть реализованы следующим образом:

```
bool Write(string Msg)
{
 if(bFull)
  return false;
 buffer[iW] = Msg;
 iW = (iW + 1) % n;
 // Если индексы совпали после записи,
 // буфер заполнен
  if(iW == iR)
  bFull = true;
 return true;
}
bool Read(ref string Msg)
    {
     // Если индексы совпадают, но не после операции записи
```

```
    // буфер пуст
    if(iW == iR && !bFull)
  return false;
    Msg = buffer[iR];
    iR = (iR + 1) % n;
    // Если буфер был заполнен, то снимаем отметку
    if(bFull)
   bFull = false;
    return true;
   }
```

Главный поток контролирует статус завершения операций чтения и записи. Если операция чтения не выполнена, то поток читателя блокируется.

Ситуация усложняется, если доступ к буферу осуществляют несколько читателей и несколько писателей. Один из вариантов решения проблемы – добавить конструкции критической секции в функции чтения и записи.

Другой подход заключается в реализации схемы "управляющий-рабочие", где управляющий контролирует все операции, требующие синхронизации. Рабочие потоки (читатели и писатели) обращаются к управляющему (основной поток) с сигналом о готовности осуществлять операцию чтения или записи. Управляющий поток фиксирует обращения читателей и писателей, вычисляет текущие индексы для чтения и записи, контролирует состояние буфера (полностью заполнен или полностью пуст), выбирает читателя и писателя, которым разрешает доступ. Операции чтения и записи по корректным индексам, полученным от управляющего потока, осуществляются читателями и писателями уже без контроля.

Взаимодействие рабочих и управляющего удобно организовать с помощью сигнальных сообщений типа ManualResetEventSlim.

Сигналы о готовности evReadyToRead, evReadyToWrite генерируют читатели и писатели, готовые осуществлять операции с буфером. Управляющий контролирует состояние сигналов у каждого рабочего.

Сигналы о возможности операций чтения и записи evStartReading, evStartWriting генерируются управляющим потоком конкретным читателям и писателям. Перед генерацией сигналов управляющий вычисляет индекс чтения или записи и сохраняет его в индивидуальной ячейке конкретного рабочего.

Такая организация взаимодействия позволяет достаточно легко изменять правила доступа: вводить приоритеты читателей и писателей, учитывать время обращения к управляющему потоку и обеспечивать "справедливость" доступа в плане очередности.

```
void ReaderThread(int iReader,
  ManualResetEventSlim evReadyToRead,
  ManualResetEventSlim evStartReading)
{
 // Инициализация внутреннего буфера
 var Messages = new List<string>();
 // Рабочий цикл чтения
 while(true)
  {
    // Сигнализирует о готовности
    evReadyToRead.Set();
    // Ждем сигнала от менеджера
    evStartReading.Wait();
    // Разрешено чтение по текущему индексу
    int k = ReadIndexCopy[iReader];
    Messages.Add(buffer[k]);
   // Сбрасываем сигнал о чтении
```

```
    evStartReading.Reset();
    // Проверяем статус завершения работы
    if (finish) break;
  }
}
// Код писателя практически идентичен коду читателя
void WriterThread(int iWriter,
 ManualResetEventSlim evReadyToWrite,
 ManualResetEventSlim evStartWriting)
{
 // Инициализация массива сообщений писателя
 Messages = ..
 // Рабочий цикл записи
 while(true)
  {
    // Сигнализируем о готовности менеджеру
    evReadyToWrite.Set();
    // Ждем сигнала от менеджера
    evStartWriting.Wait();
    // Разрешена запись по текущему индексу
    k = WriteIndexCopy[iWriter];
    buffer[k] = Messages[j];
    // Проверяем статус завершения работы
    if (finish || j >= Messages.Length)
   break;
    j++
    }
  }
  // Код менеджера
```

```
void Manager(int nReaders, int nWriters)
{
 // Запуск читателей
 for(int i=0; i<nReaders; i++)
 {
  evReadyToRead[i] =
  new ManualResetEventSlim(false);
  evStartReading[i] =
   new ManualResetEventSlim(false);
  tReaders[i] = new Task( () =>
  Reader(i, evReadyToRead[i], evStartReading[i]));
  tReaders[i].Start();
 }
 // Запуск писателей
 for(int i=0; i < nWriters; i++)
 {
  var evReadyToWrite[i] =
   new ManualResetEventSlim(false);
  var evStartWriting[i] =
   new ManualResetEventSlim(false);
  tWriters[i] = new Task( () =>
  Writer(i, evReadyToWrite[i], evStartWriting[i]));
  tWriters[i].Start();
 }
 // Рабочий цикл
 while(true)
 {
  // Если в буфере есть свободные ячейки
  // пытаемся обработать готовых писателей
```

```
if(!bFull)
{
 // Получаем текущий индекс записи
 iW = GetBufferWriteIndex();
 if(iW != -1)
 {
  // Устанавливаем писателя,
  // которому разрешаем работать
  iWriter = GetWriter();
  if (iWriter != -1)
  {
  // Сбрасываем сигнал готовности
   // выбранного писателя
   evReadyToWrite[iWriter].Reset();
   // Сохраняем копию индекса для записи
   ReadIndexCopy[iWriter] = iW;
   // Разрешаем писателю начать работу
  evStartWriting[iWriter].Set();
 }
 }
 else
 bFull = true;
}
// Если буфер не пуст, пытаемся
// обработать готовых писателей
if(!bEmpty)
{
 // Получаем текущий индекс для чтения
 iR = GetBufferReadIndex();
```

```
  if(iR != -1)
  {
   //Устанавливаем готового читателя
   iReader = GetWriter();
   if (iReader != -1)
   {
    evReadyToRead[iReader].Reset();
    WriteIndexCopy[iReader] = iR;
   evStartReading[iReader].Set();
   }
  }
  else
   bEmpty = false;
 }
}
// Код функции получения номера готового писателя
// с учетом приоритетов
int GetWriter()
{
 // Устанавливаем готовых писателей
 var ready = new List<int>();
 for(int i=0; i<nWriter; i++)
  if(evReadyToWrite[i].IsSet())
   ready.Add(i);
 if(ready.Count == 0)
  return -1;
 return ready.OrderBy(i => WriterPriority[i]).First();
}
```

Вопросы и упражнения

1. Можно ли вместо объектов ManualResetEventSlim использовать другие типы сигнальных сообщений: AutoResetEvent илиManualResetEvent?

2. Какие особенности задачи не позволяют использовать объект ReaderWriterSlim?

3. Почему структура кольцевого буфера не требует синхронизации при работе одного читателя и одного писателя?

4. Почему в предложенной реализации не используются критические секции?

5. Реализуйте учет времени обращения рабочих потоков к буферу.

6. Реализуйте решение задачи с использованием конкурентных коллекций в качестве буфера для оптимизации вычислительного процесса.

Лабораторная работа № 6. Клеточная модель "Игра Жизнь" Дж.Конвея

Клеточная модель представляет собой множество клеток (ячеек таблицы), принимающих одно из нескольких состояний. Состояние каждой клетки определяется состоянием её ближайших соседей. Одной из известных моделей является "Игра Жизнь" математика Дж. Конвея.

В модели Конвея каждая клетка может находиться в двух состояниях: живая или неживая. Состояние клетки на следующем шаге определяется потенциалом клетки (числом живых соседних клеток):

если потенциал клетки равен двум, то клетка сохраняет свое состояние;

если потенциал равен трем, то клетка оживает;

если потенциал меньше двух или больше трех, то клетка погибает.

Правила изменения состояния клетки можно описать следующим лямбда-выражением:

```
var lifeRules = new Func<int, bool, bool>((p, state) =>
 {
  if(p == 3)
   return true;
  else if (p == 2)
   return state;
  else
   return false;
 });
```

Последовательный алгоритм расчета представляет собой расчет состояния каждой клетки

```
LifeTable tableNew = new LifeTable;
```

```
for(int i=0; i < Height; i++)
  for(int j=0; j < Width; j++)
  {
   p = CalcPotential(table[i,j]);
     tableNew[i, j] = lifeRules(p, table[i, j]);
  }
table = tableNew;
```

Потенциал клетки вычисляется по восьми ближайшим соседям клетки.

```
int CalcPotential(int i, int j)
{
 int p=0;
 for(int x = i-1; x <= i + 1; x++)
  for(int y = j-1; y <= j + 1;y++)
  {
   if(x < 0 || y < 0 || x >= Height || y >= Width
      ||  (x == i && y == j))
    continue;

   if(table[x,y])  p++;

  }
return p;
}
```

Вычисления новых состояний для каждой клетки являются независимыми и могут выполняться параллельно.

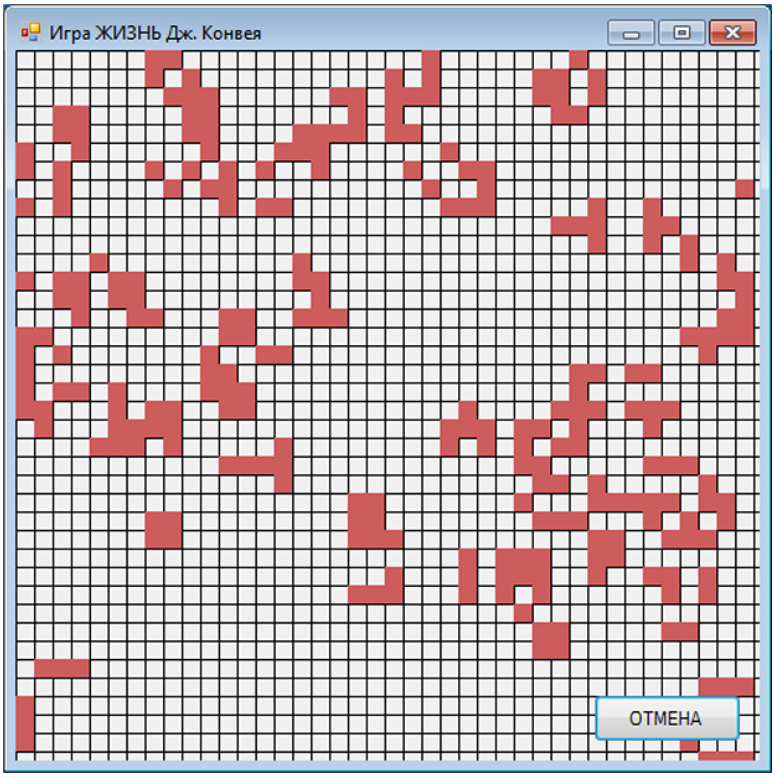
```
Parallel.For(0, Height, (i) => {
```

```
 for(int j=0; j < Width; j++)
  {
   p = CalcPotential(table[i,j]);
     tableNew[i, j] = lifeRules(p, table[i, j]);
  }
});
```

Также можно изменить порядок расчета и распараллелить внешний цикл по столбцам:

```
Parallel.For(0, Width, (j) => {
 for(int i=0; i < Height; i++)
  {
   p = CalcPotential(table[i,j]);
     tableNew[i, j] = lifeRules(p, table[i, j]);
  }
}
```

Вывод таблицы состояний клеточной модели может быть следующим:



Задания

1. Реализуйте Windows-приложение, которое последовательно отображает состояния клеточной модели "Игра жизнь".
2. Реализуйте последовательный алгоритм расчета состояний модели.
3. Реализуйте параллельные алгоритмы расчета состояний модели. Для распараллеливания используйте задачи (tasks) или метод Parallel.For.
4. Реализуйте возможность отмены расчета с помощью объекта CancellationToken.
5. Выполните анализ эффективности разработанных алгоритмов.

Вопросы и упражнения

1. Имеет ли смысл распараллеливание внутреннего цикла расчета? Почему?

```
for(int i=0; i < Height; i++)
{
 Parallel.For(0, Width, j =>
  {
   p = CalcPotential(table[i,j]);
     tableNew[i, j] = lifeRules(p, table[i, j]);
  });
}
```

2. Как вариант расчета – по строкам или по столбцам – более эффективен и с чем это связано?
3. Продумайте вариант блочной декомпозиции, где блок выступает матрицей размера KxK. В чем достоинства и недостатки блочной декомпозиции для этой задачи? Какое значение параметра K следует выбирать?

Лабораторная работа № 7.

Анализ оптимальности выполнения вычислительного процесса с помощью "Визуализатора параллелизма" в Visual Studio

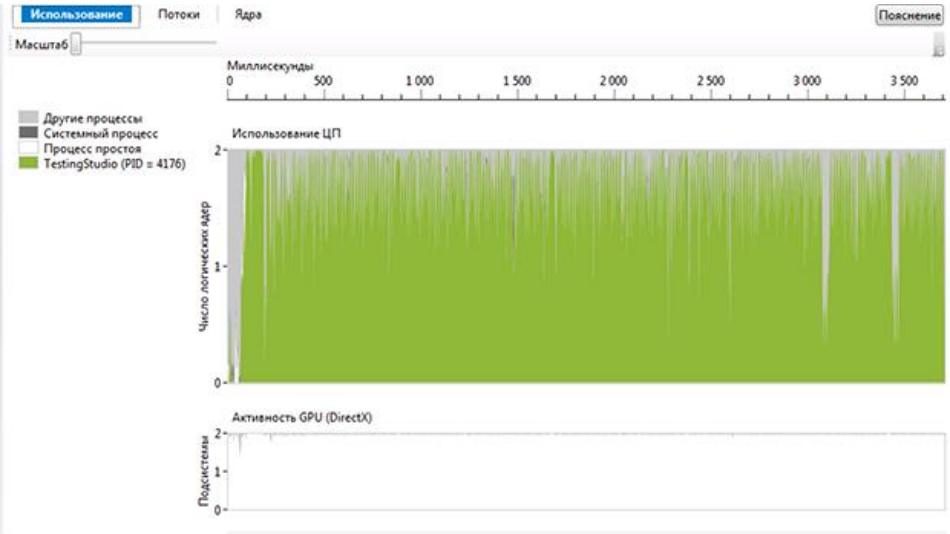
Задачи

1. Выполните анализ загруженности вычислительной системы при разном режиме ожидания потока.
2. Выполните анализ автоматического распараллеливания циклической обработки шаблоном Parallel.For для оптимизации вычислительного процесса.
3. Выполните анализ распараллеливания PLINQ-запросов для оптимизации вычислительного процесса.

Среда разработки Visual Studio содержит полезный инструмент для анализа эффективности оптимизации вычислительного процесса – "Визуализатор параллелизма". Для запуска инструмента на вкладке "Анализ" запускаем "Визуализатор параллелизма" -> "Выполнить анализ с текущим процессом". Инструмент запускает программу и собирает информацию о её фактическом исполнении на данной вычислительной системе. Основные вкладки результатов: "Использование", "Потоки", "Ядра".

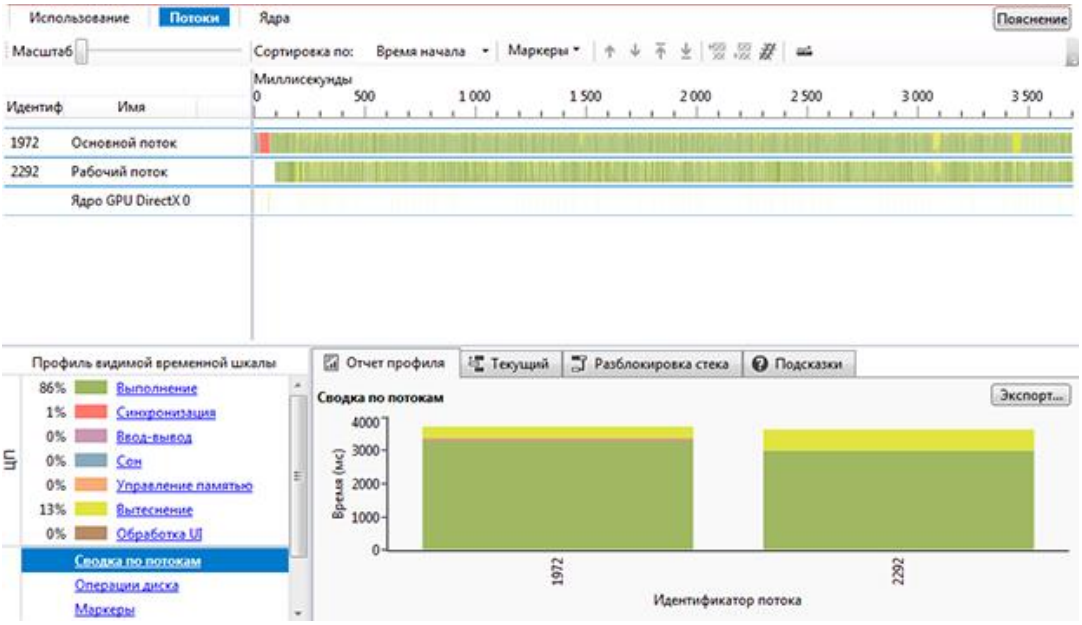
Использование ЦП

Вкладка содержит информацию об общей загрузке ЦП приложением в течение всего интервала выполнения.



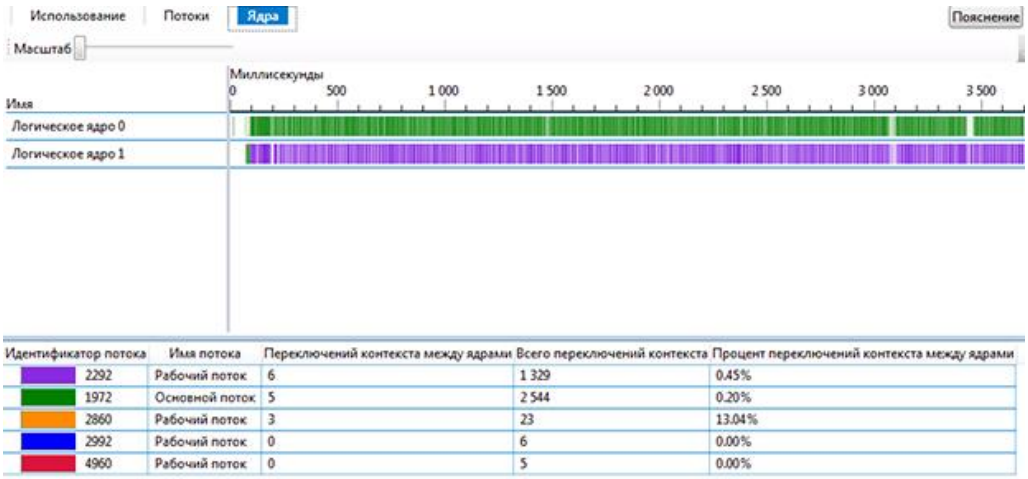
Потоки

Вкладка "Потоки" выводит информацию о выполнении потоков, как пользовательских, так и рабочих потоков пула. Если в программе не используются объекты ThreadPool, Parallel, Task, то рабочие потоки простаивают. Для удобства восприятия информацию об отдельных потоках, например, простаивающих, можно скрыть. График "Сводка по потокам" суммирует время нахождения потоков в том или ином состоянии.



Ядра

Вкладка "Ядра" позволяет проанализировать перемещение потоков по фактическим исполнителям - ядрам процессора.



Идентификатор потока	Имя потока	Переключений контекста между ядрами	Всего переключений контекста	Процент переключений контекста между ядрами
2292	Рабочий поток	6	1 329	0.45%
1972	Основной поток	5	2 544	0.20%
2860	Рабочий поток	3	23	13.04%
2992	Рабочий поток	0	6	0.00%
4960	Рабочий поток	0	5	0.00%

Анализ блокировки потоков

Применение "Визуализатора" позволяет исследовать особенности разных средств синхронизации для оптимизации вычислительного процесса.

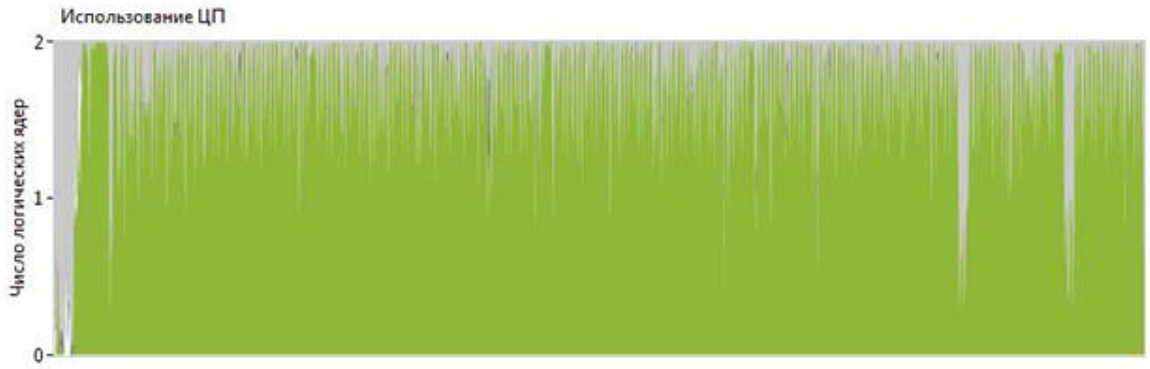
Следующие фрагменты осуществляют блокировку основного потока с помощью *цикла* ожидания (активная *блокировка*) и с помощью встроенного механизма Join с выгружением контекста потока.

```
// Фрагмент 1
 Thread t = new Thread(() => {
  for (int i=0; i<10000; i++)
   for(int j =0; j < 100; j++)
    a[i] += Math.Sin(j) + i;
  });

t.Start();
// Цикл ожидания
 while(t.IsAlive) ;

 // Фрагмент 2
 Thread t = new Thread(() => {
  for (int i=0; i<10000; i++)
   for(int j =0; j < 100; j++)
```

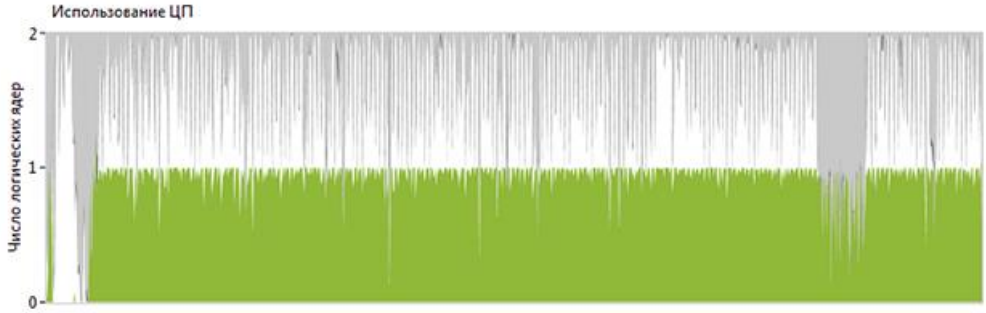
```
    a[i] += Math.Sin(j) + i;


  });
 t.Start();
 t.Join();
```

Окно "Использование ЦП" иллюстрирует основной недостаток циклической блокировки – полезную работу осуществляет один *поток*, а вычислительные ресурсы заняты двумя потоками

Циклическая *блокировка*

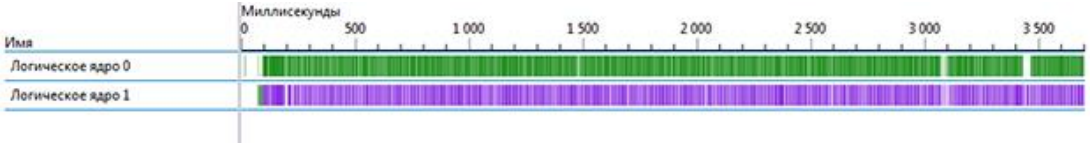


Блокировка Join

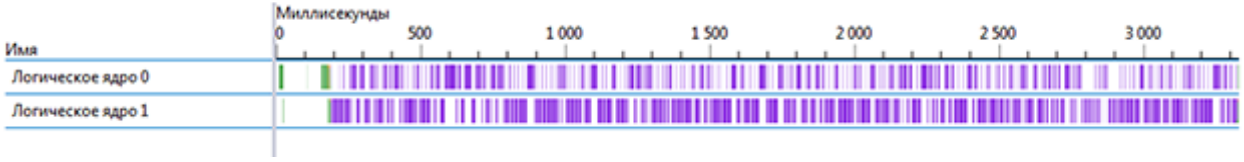


При Join-блокировке основной *поток* №4808 почти все время находится в состоянии "Синхронизация", ожидая завершения потока №4848.

Вкладка "ядра" позволяет зафиксировать интересную особенность блокировок. При циклическом ожидании работают два потока (рабочий и основной), на двуядерной системе потоки практически не перемещаются между ядрами.



При блокировке основного потока методом Join, *вычислительная система* с двумя ядрами полностью предоставлена одному рабочему потоку. *Операционная система* активно перемещает *поток* с одного ядра на другое.



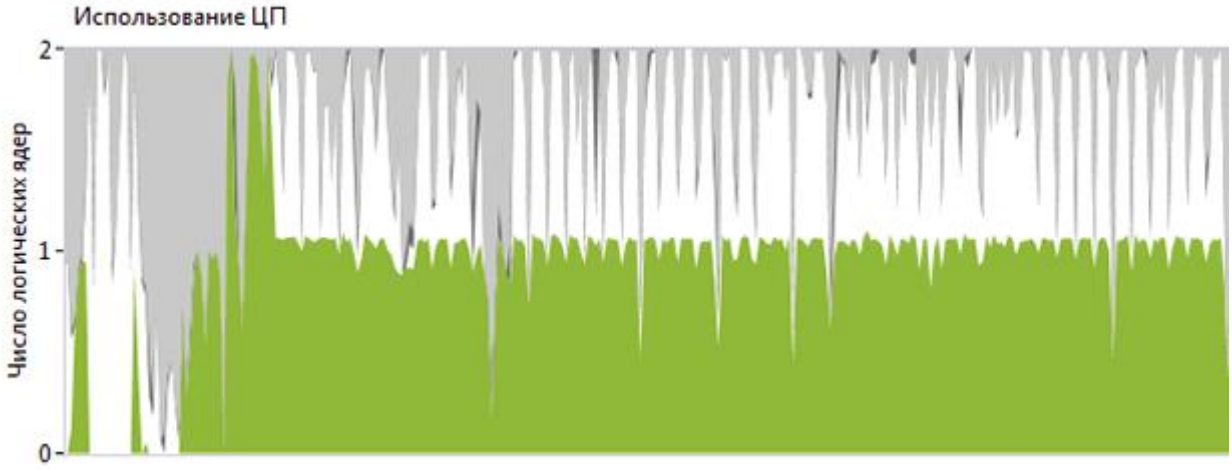
Идентификатор потока	Имя потока	Переключений контекста между ядрами	Всего переключений контекста	Процент переключений контекста между ядрами
4848	Рабочий поток	929	1 929	48.16%
4808	Основной поток	10	84	11.90%
4948	Рабочий поток	2	30	6.67%
4724	Рабочий поток	2	7	28.57%
4276	Рабочий поток	1	8	12.50%

При таком выполнении существуют дополнительные накладные *расходы*, связанные с переключением контекста между ядрами (процент переключений между ядрами 48.16% от общего числа переключений).

При выполнении полезной обработки двумя и большим числом потоков загруженность системы увеличится, и доля переключений между ядрами будет минимальной.

Блокировка SpinWait комбинирует два типа ожидания.

```
// Фрагмент 1
 Thread t = new Thread(() => {
  for (int i=0; i<10000; i++)
   for(int j =0; j < 100; j++)
    a[i] += Math.Sin(j) + i;
  });

t.Start();
// Цикл ожидания
 SpinWait.SpinUntil(() => !t.IsAlive);
```



В начале *цикла* ожидания выполняется несколько прокруток, после чего ожидающий *поток* выгружается, освобождая вычислительные ресурсы.

Анализ выполнения объектов библиотеки TPL

Применение объектов *TPL* для распараллеливания скрывает от программиста работу по созданию и синхронизации потоков при

оптимизации вычислительного процесса. "Визуализатор параллелизма" позволяет получить основную информацию о действиях среды выполнения по распараллеливанию задач.

Метод Parallel.For автоматически распределяет итерации *цикла* по рабочим потокам. Вкладка "Потоки" раскрывает особенности выполнения параллельной циклической обработки.

Специальная строка System.Threading.Tasks указывает на вызов и работу объекта TPL. В окне "Сводка" отображается итоговая информация о выполнении цикла и оптимизации вычислительного процесса. Рассматривая работу потоков, видим, что в каждый момент времени работают только два потока, другие потоки вытесняются. Основной поток также участвует в обработке цикла. Сначала для обработки используются два дополнительных рабочих потока, спустя какое-то время в работу включается еще один поток.

Основная литература

Туральчук, К. А. Параллельное программирование с помощью языка C# : учебное пособие / К. А. Туральчук. — 2-е изд. — Москва : ИНТУИТ, 2016. — 189 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100360 (дата обращения: 27.06.2023). — Режим доступа: для авториз. пользователей.

Танвар, Ш. Параллельное программирование на C# и .NET Core / Ш. Танвар ; редактор В. Н. Черников ; перевод с английского А. Д. Ворониной. — Москва : ДМК Пресс, 2022. — 272 с. — ISBN 978-5-97060-851-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/241118 (дата обращения: 27.06.2023). — Режим доступа: для авториз. пользователей.

Богачёв, К.Ю. Основы параллельного программирования [Электронный ресурс] : учеб. пособие — Электрон. дан. — Москва : Издательство "Лаборатория знаний", 2015. — 345 с. — Режим доступа: https://e.lanbook.com/book/70745. — Загл. с экрана.

Дополнительная литература

1. Соснин, В.В. Введение в параллельные вычисления. [Электронный ресурс] / В.В. Соснин, П.В. Балакшин. — Электрон. дан. — СПб. : НИУ ИТМО, 2015. — 51 с. — Режим доступа: https://e.lanbook.com/reader/book/91486/#1— Загл. с экра-на.

2. Энтони, У. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ [Электронный ресурс] : учеб.

пособие — Электрон. дан. — Москва : ДМК Пресс, 2012. — 672 с. — Режим доступа: https://e.lanbook.com/book/4813. — Загл. с экрана.

3. Кокоса, К. Управление памятью в .NET для профессионалов : практиче-ское руководство / К. Кокоса. - Москва : ДМК Пресс, 2020. - 800 с. - ISBN 978-5-97060-800-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1210679 (дата обращения: 08.11.2020). – Режим доступа: по подписке.

4. Федотов, И. Е. Модели параллельного программирования : учебное посо-бие / И. Е. Федотов. — Москва : СОЛОН-Пресс, 2012. — 384 с. — ISBN 978-5-91359-102-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/13807 — Режим доступа: для авториз. пользо-вателей.